DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

COMPUTER ARCHITECTURES

Cache Memory

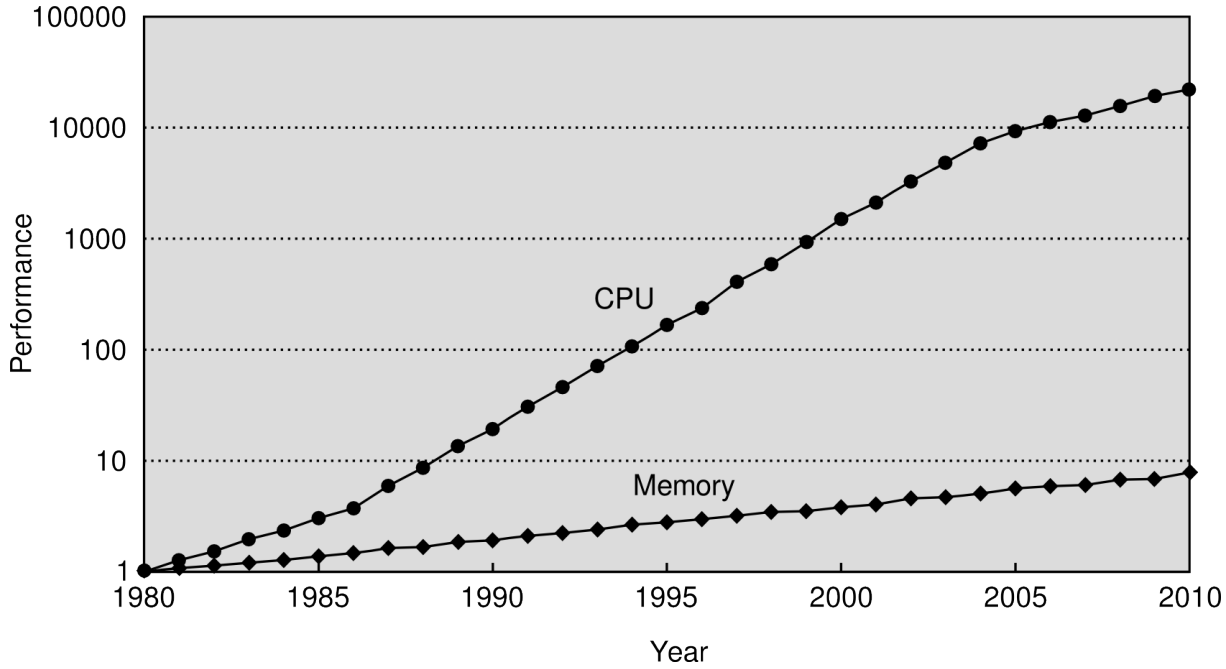
Gábor Horváth, ghorvath@hit.bme.hu

Budapest,
2025. 03. 17.

MŰEGYETEM 1782

SPEED OF MEMORY OPERATIONS

- The memory is a serious bottleneck of Neumann computers
- Because it is slow



- Using virtual memory makes it even worse
  - 1 memory operation generates several memory accesses due to page table lookup

REMEDY: LOCALITY

- Programs do not refer to memory addresses randomly
- Memory addresses referenced by the programs show a special pattern

  → we can utilize it!
- Locality of reference:
  - **Temporal**: a memory content referenced will be referenced again in the near future
  - **Spatial**: if a memory content has been referenced, its neighborhood will be referenced as well in the near future
  - **Algorithmic**: some algorithms (like traversing linked lists) refer to the memory in a systematic way
- Examples:
  - Media players:
    - Spatial locality: yes, temporal locality: no
  - „For” loop in the C language:
    - Both temporal and spatial locality hold

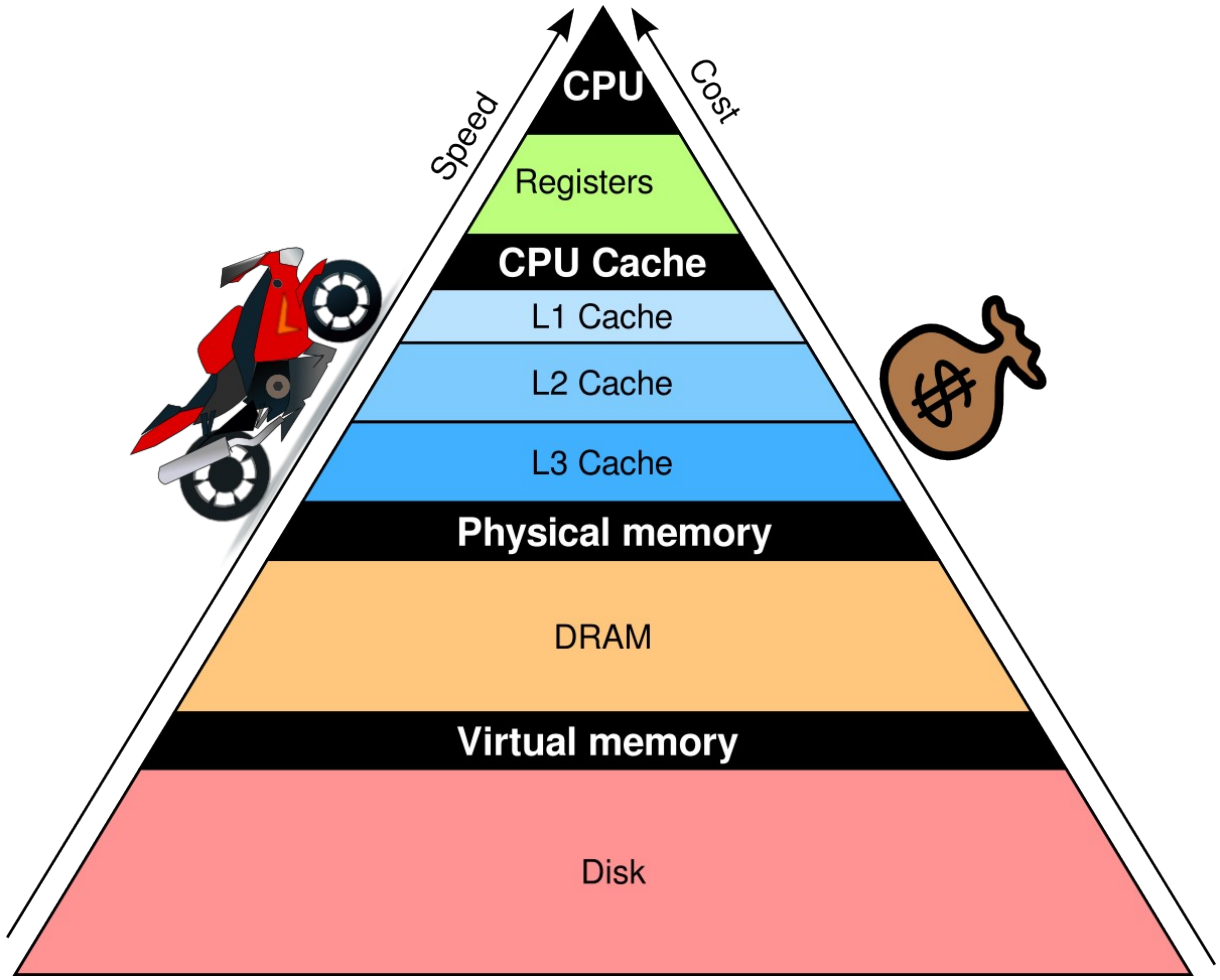
MEMORY HIERARCHY

- If the programs show this „locality of reference” behavior:
  - Let us move the frequently used data close the CPU, and store it in a special, fast memory
- Why, isn’t all kinds of memory slow?
  - No. The slowest is the hard disk drive...
  - … and we have SRAM, that is much faster than DRAM
  - SRAM is the fastest, but also much more expensive

Storage technology	Access time	Price/GB
SRAM	0.5 – 2.5 ns	2000 – 5000 $
DRAM	50 – 70 ns	20 – 50 $
HDD	5 – 20 * 10^6 ns	0.2 $

(data from 2008)

MEMORY HIERARCHY



CACHE IMPLEMENTATIONS

- Classification of the various cache implementations:
- According to „*addressing scheme*” the cache can be
  - Transparent
    - The cache memory stores a part of the system memory
  - Non-transparent
    - A part of the address space is covered by cache memory (SRAM), the rest is covered by the system memory (DRAM)
- According to „*management scheme*” the cache may have:
  - Implicit management:
    - The content of the cache is managed by the hardware (CPU)
  - Explicit management:
    - The content of the cache is managed by the applications

CACHE IMPLEMENTATIONS

	Addressing scheme	Management scheme
Transparent cache	Transparent	Implicit
Software managed cache	Transparent	Explicit
Self-managed scratch-pad	Non-transparent	Implicit
Scratch-pad memory	Non-transparent	Explicit

- Transparent cache: classical CPU cache
- Scratch-pad memory: DSPs, microcontrollers, PlayStation 3
- Software managed cache:
  - No cache hit → it is the task of the application to update the cache
- Self-managed scratch-pad
  - Rare

CACHE IMPLEMENTATION

- What we are going to cover are:
  - Cache organization:
    - How to store data in the cache in an efficient way (e. g. it is fast to find them, low power dissipation, low number of transistors are needed)
  - Cache content management:
    - When to put a data into the cache and when not
    - What shall we throw out from the cache, if we want to put new data there

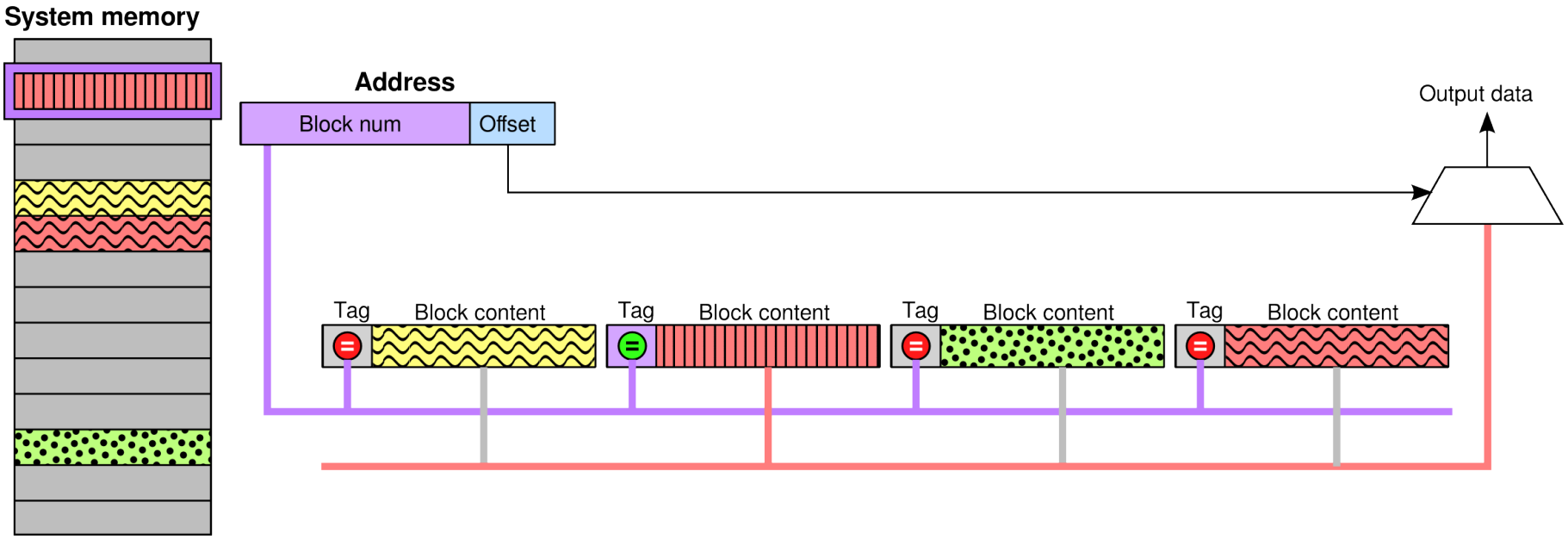
Cache organizations

TRANSPARENT CACHE ORGANIZATIONS

- Data units in the cache are called: ***block***
  - Size of blocks = 2^L
  - Lower L bit of memory addresses: position inside a block
  - Upper bits: number (ID) of the block
- Additional information to be stored in the cache with each block:
  - Cache ***tag*** (which block of the system memory is stored here)
  - ***Valid*** bit: if =1, this cache block stores valid data
  - ***Dirty*** bit: if =1, this cache block has been modified since in the cache
- The principal questions of cache organization are:
  - How to store blocks in the cache
    - To enable finding a block fast
    - To make it simple and relatively cheap

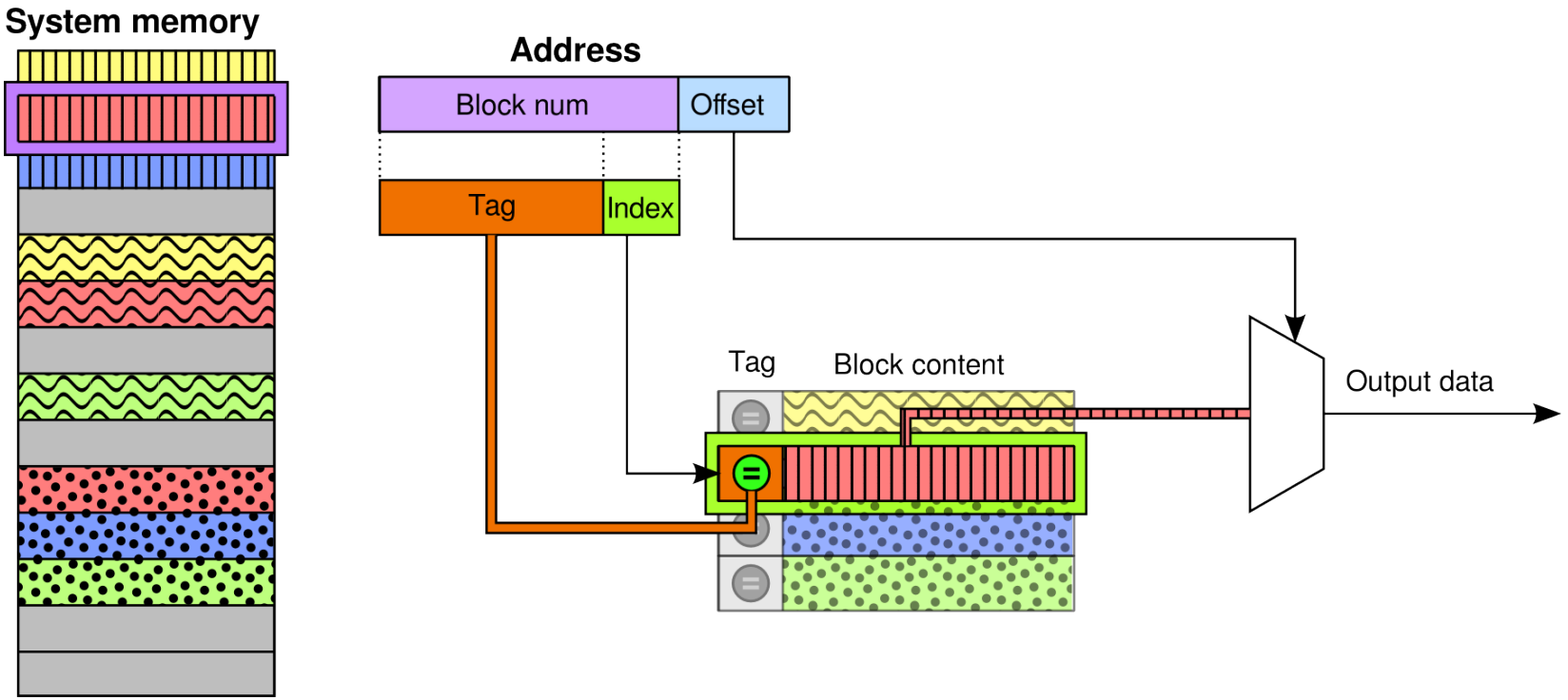
FULLY ASSOCIATIVE CACHE ORGANIZATION

- Blocks can be placed anywhere in the cache
- Cache tag: which block of the system memory is stored here
- It has high energy consumption (thus, heat), since
  - Lookup: block number of the address has to be compared to **all** cache tags
  - The comparators are wide: the number of bits to compare equals the number of bits describing a block number



DIRECT MAPPED ORGANIZATION

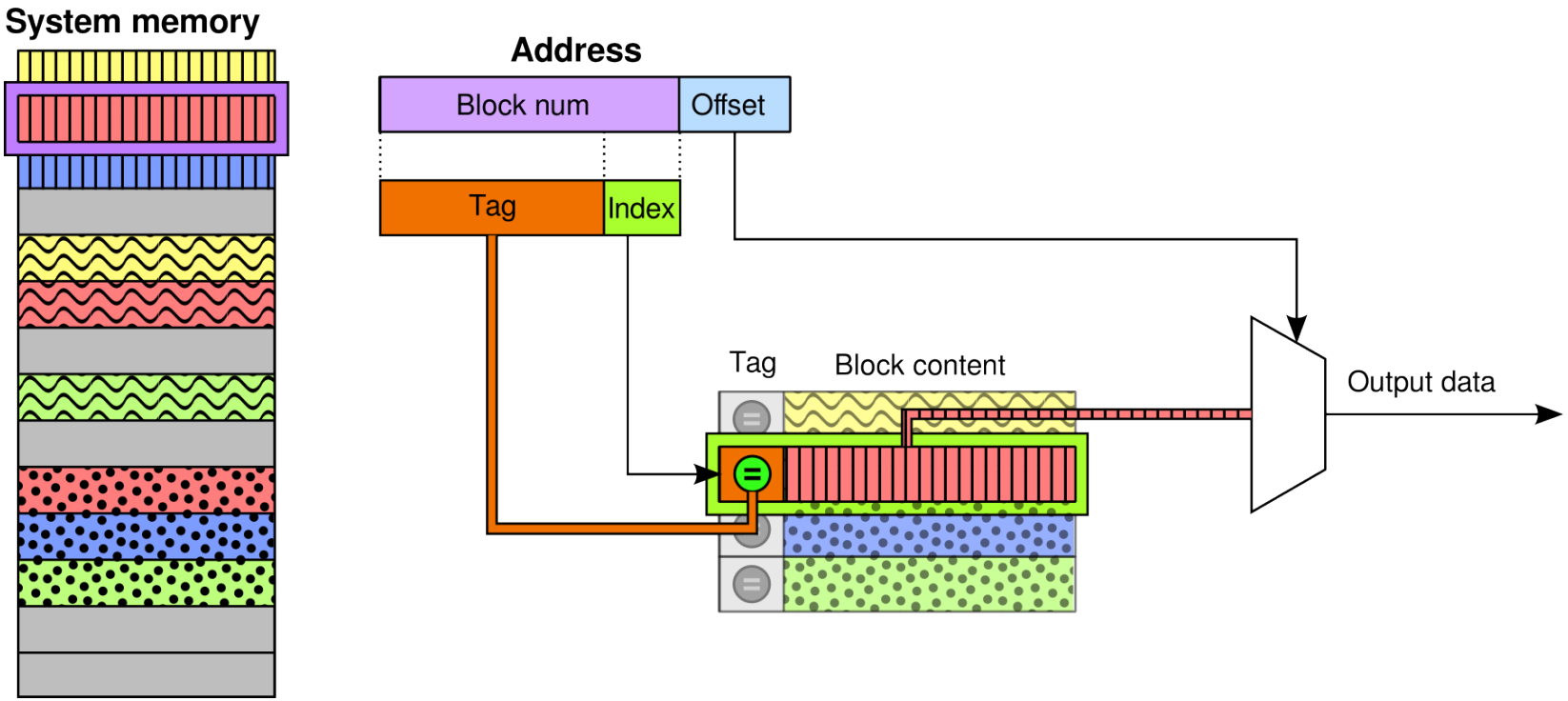
- Each block of the memory can be placed only to a single place in the cache
- The block number determines, where
  - E.g. the lower bits of the block number determines it



DIRECT MAPPED ORGANIZATION

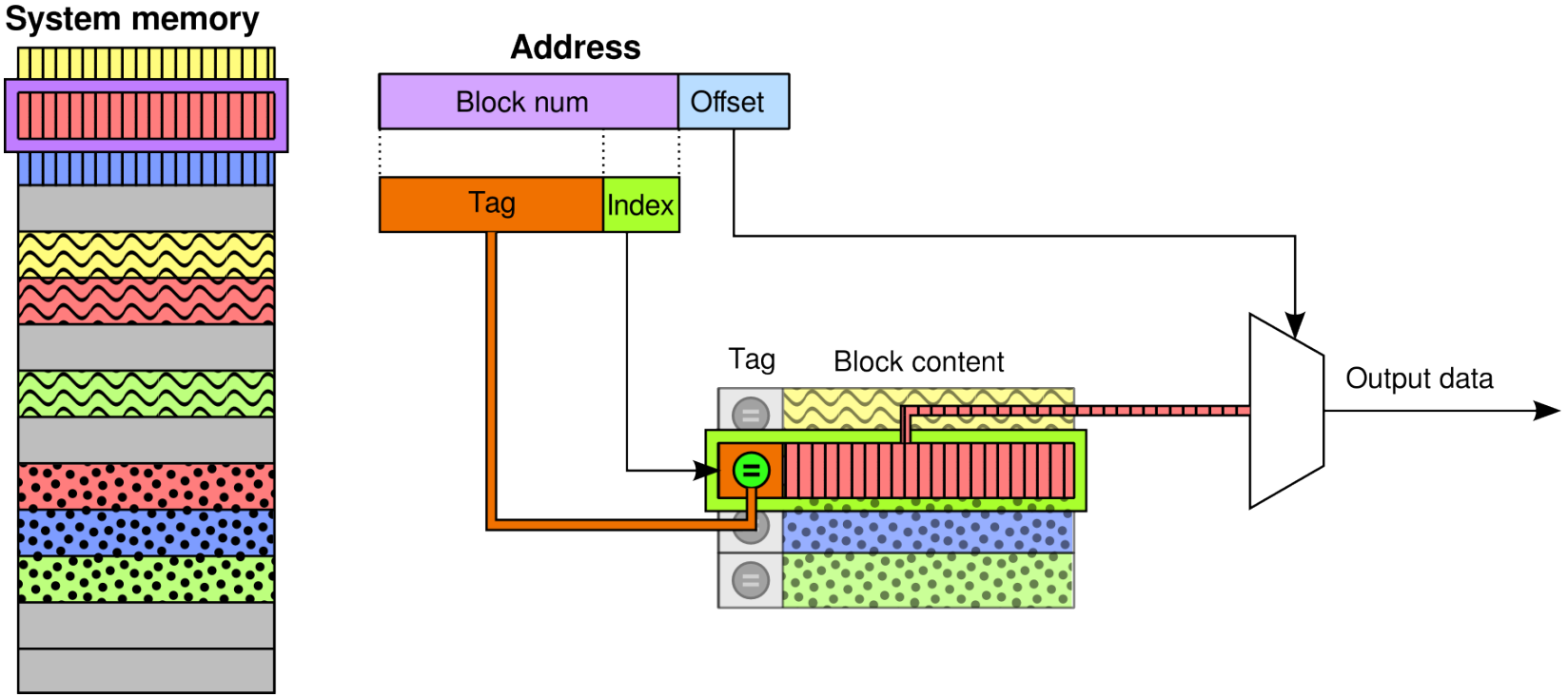
- Example: a cache can store 4 blocks: 1 yellow, 1 red, 1 blue, 1 green
- The blocks of the memory are following each other in this order

  → color = lowest 2 bits of the block number



DIRECT MAPPED ORGANIZATION

- Lookup consists of two steps:
  1) **Indexing**: The color is given by the lower two bits of the memory address: assume it is „red” (this is called: ***index***)
  2) **Comparison**: Is the „red” block of the cache storing our memory block?
     - To decide, we need a single comparison! The number of bits to compare is equal to the number of bits of the ***Tag***.



CONCLUSION

- Fully associative:
  - Block can be placed anywhere
  - At each lookup all of the comparators are in use, and the comparators need to compare a lot of bits

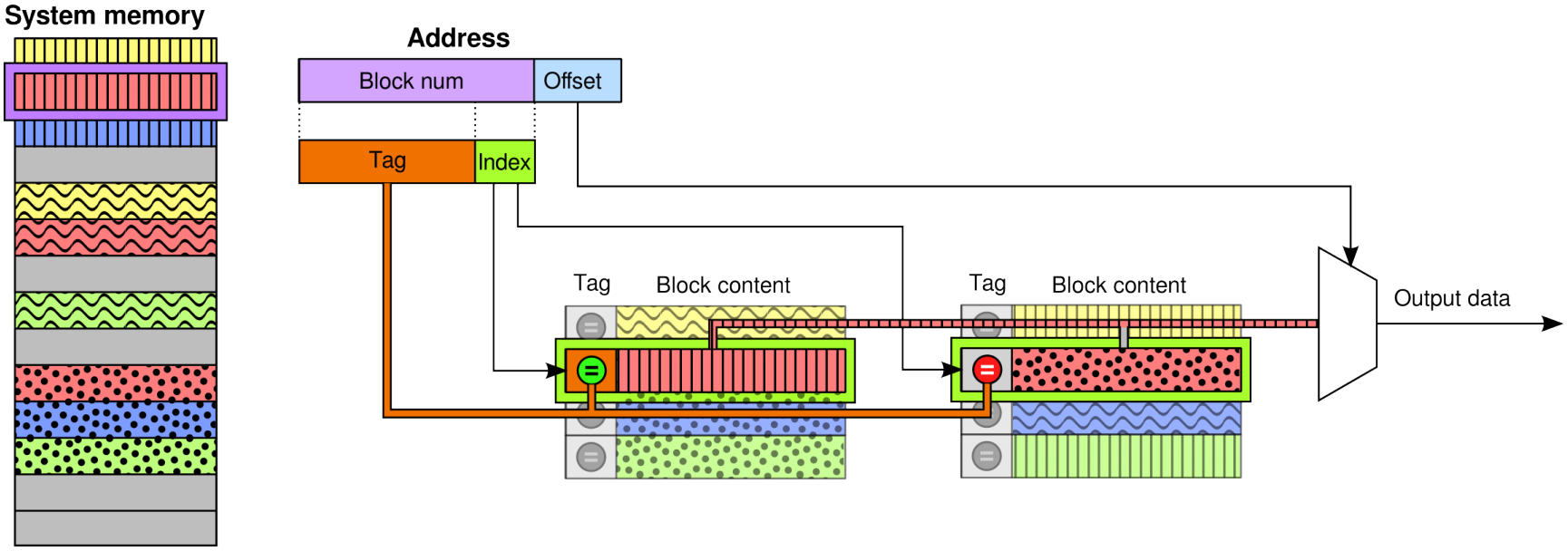
    → which is complex, and consumes a lot of energy
- Direct mapped organization:
  - Placement is restricted:

    → Contention is possible

    E.g. a program might refer only to „red” blocks
  - At each lookup, only a single comparator is in use, and there are fewer bits to compare

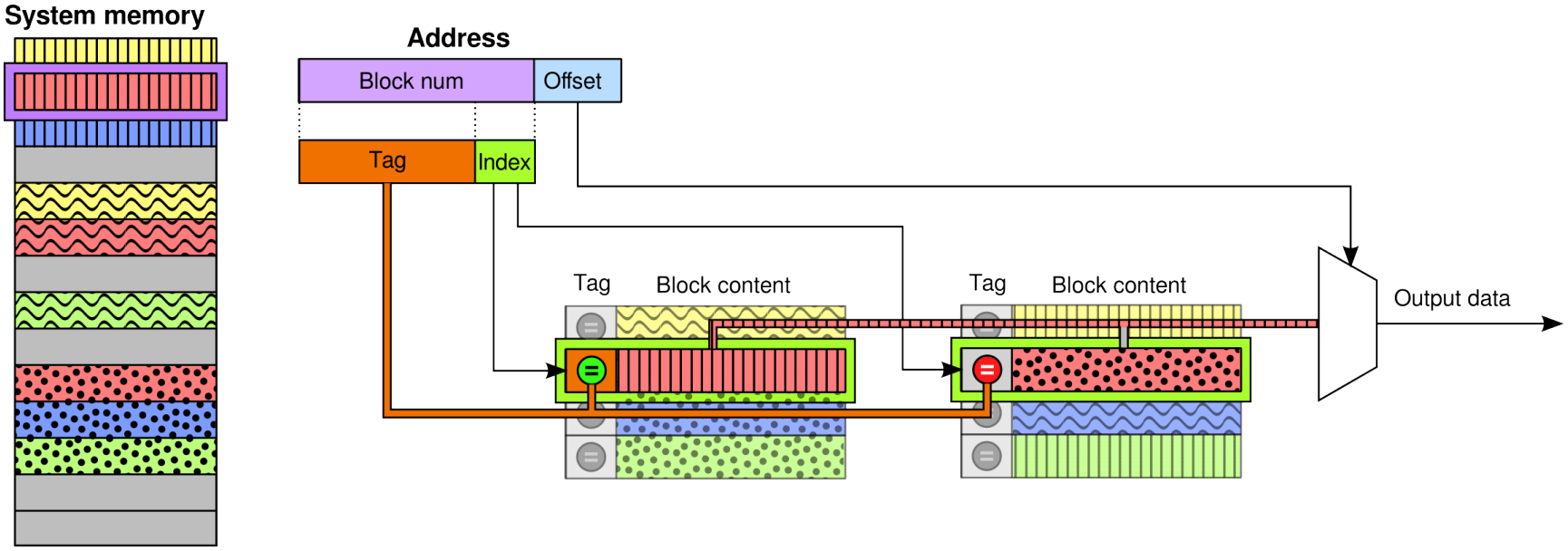
N-WAY SET-ASSOCIATIVE ORGANIZATION

- Combines the advantages of the previous two solutions
- The lower bits of the block number of memory addresses determine where the block can be placed in the cache
  - But not in a unique way!
  - The lower bits select a **set** of places, where the block can be stored
  - Each set consists of n blocks, the block we are looking for can be stored in any of them



N-WAY SET-ASSOCIATIVE ORGANIZATION

- Lookup consists of two steps:
  1) **Indexing**: The color is given by the lower two bits of the memory address: assume it is „red” (this is called: **index**)
  2) **Comparison**: There are n places in the cache that can store „red” blocks. To find out if our data is in the cache, we need to compare the address to the tags of all the n places
     - n comparators are in use, the width of each of them is given by the width of the ***Tag***



CONCLUSION

- N-way set-associative organization:
  - Restricted placement of blocks, with n possibilities
    → less contention
  - Only n comparators are switched on at lookup
    → moderate complexity and energy consumption
- Typical values of n:
  n=2...16

	Core i7 (Kaby Lake)		AMD Ryzen		ARM Cortex A53		ARM Cortex A75	
	n=	size	n=	size	n=	size	n=	size
L1	8	32 KB	4/8	64/32 KB	2/4	16-64 KB	4	64 KB
L2	4	256 KB	8	512 KB	16	128-2048 KB	8	256-512 KB
L3	16	2 MB/core	16	2 MB/core	-	-	16	1-4 MB

Cache content management

PUTTING DATA INTO THE CACHE

- When to put a memory block into the cache?
  - Never
  - When the program refers to it at the first time
  - Before the program refers to it (in the hope that it will be used later)
- **Never**: avoids cache pollution
  - e.g. media players: displays frames only once, frames will never be referred later

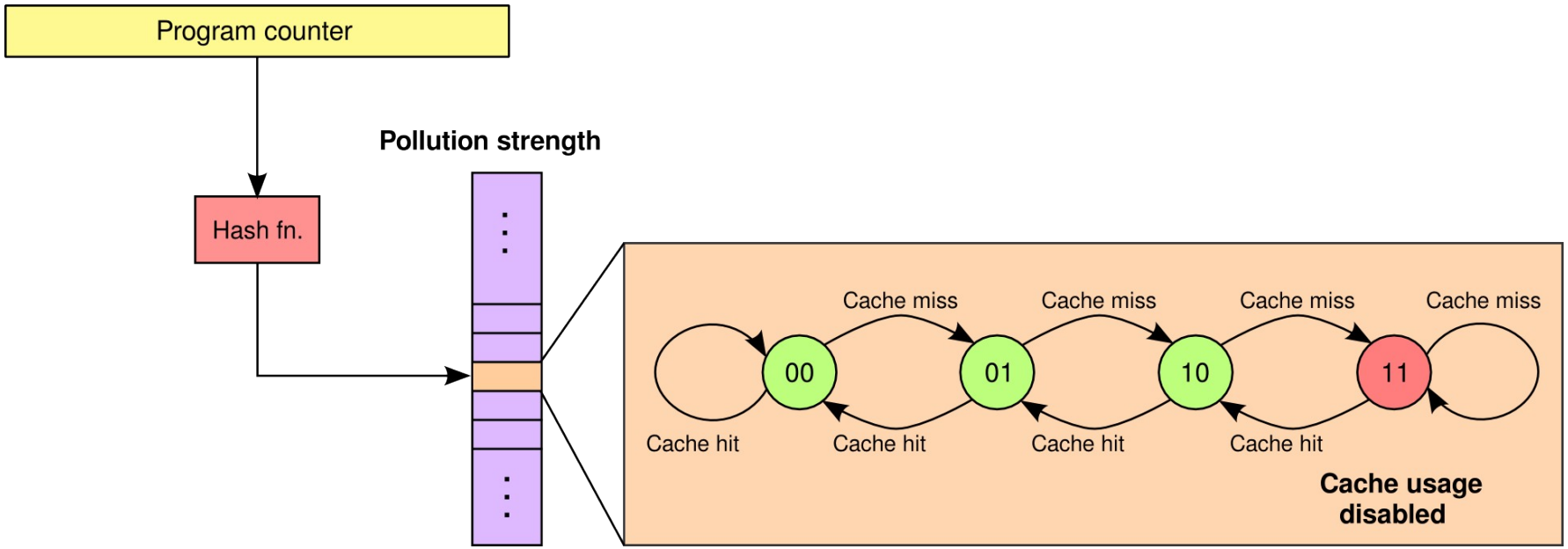
    → it is not worth to put such data into the cache
- **At the first reference**: The first reference is slow (memory → cache transfer), but it will be fast later (accessible from cache)
- **Before the first reference (prefetch)**: needs speculation

AVOIDANCE OF CACHE POLLUTION

- Cache pollution: a block that was not referred to between loading and leaving the cache
  - It was a mistake putting it to the cache
  - Useful data might have been thrown out due to loading it
- Explicit solutions: specific instructions
  - The programmer can tell the processor what **not** to load into the cache
  - **x86**: MOVNTI, MOVNTQ
    - Moves data from an SSE register to the memory, bypassing the cache
  - **PowerPC**: LVXL reads vectors from memory to registers
    - The block will be put to the cache, but it will be marked to be not important
  - **PA-RISC**: Several instructions have a bit:
    - Spatial Locality Cache Control Hint
  - **Itanium**: Data movement instructions have an option: „.nt”
    - Indicates that the data does not exhibit temporal locality
  - **Alpha**: ECB instruction (Evict Data Cache Block)
    - Indicates that the block will not be referred in the near future

AVOIDANCE OF CACHE POLLUTION

- Implicit solutions: CPU tries to detect instructions that pollute the cache automatically
- Several solutions in use. Example: Rivers' algorithm
- Idea: if an instruction causes too many cache misses, but does not have cache hits → the CPU does not allow it to use the cache



DATA PREFETCH

- An important functionality
- Goal: the CPU should not wait for the slow memory at all
- All data has to be loaded into the cache before the first reference occurs to them
- It is simple in case of ***Instruction* cache**:
  - Non-jump instructions are executed sequentially
    → the next instructions to load is known
  - The jump addresses of unconditional jump instructions can be pre-computed
    → the next instructions can be loaded in advance
  - In case of conditional jump instructions
    → the jump address can be estimated

DATA PREFETCH

- For **data cache**:
  - Predicting which data will be referred in the near future is less trivial
  - If the prediction is:
    - Too conservative: the data will not be in the cache, when needed
      → too many cache misses
    - Too aggressive: pollutes the cache with unnecessary blocks
      → useful data get thrown out → too many cache misses
  - Explicit solutions:
    - Special instructions
  - Implicit solutions:
    - The CPU speculates what will be the next referred data

DATA PREFETCH

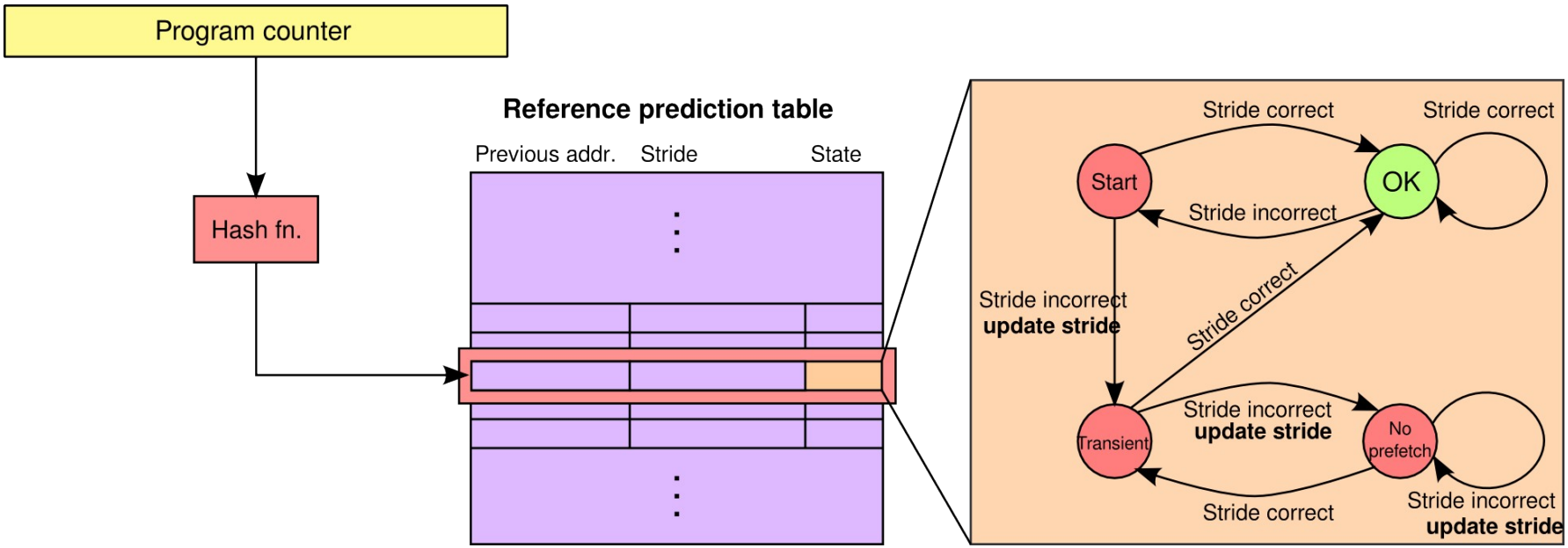
- Explicit prefetch instructions:
  - **x86**: PREFETCHT0/1/2, PREFETCHNTA
    - Load a block into the cache
    - PREFETCHNTA: loads data and puts it the first block in the n-way set-associative cache → avoids cache pollution as well!
  - **PowerPC**: DCBT, DCBTST
    - Loads a block into the cache
  - **PA-RISC**: LDD, LDW
    - Loads a block into the cache
  - **Itanium**: lfetch
    - Loads a block into the cache
  - **Alpha**: If the destination register of a data movement instruction is R31, then the CPU treats it as a prefetch request

- Platform independent solution of the GCC C compiler:

```
__builtin_prefetch (pointer);
```

DATA PREFETCH

- Implicit solution:
  - To detect iterating over data placed equi-distantly in the memory:
    - X, X + stride, X + 2*stride, X + 3*stride, ...
  - Detects stride automatically
- = "Intel Smart Memory Access IP Prefetcher" in Intel Core i7



CACHE REPLACEMENT STRATEGY

- We already know which block to put into the cache and when to do it
- But where to put it?

  → the cache memory is small, therefore it is usually full
- A block needs to be removed from the cache
- Number of candidates = the associativity of the cache
  - There is no choice in case of direct mapped cache

    The new block can be stored to a single place in the cache. The previous content of that place is removed.
- Possible algorithms to select the victim:
  - Random choice
  - Round robin
  - **Least recently used (LRU) – this is the most popular strategy**
  - Not the recently used
  - Least frequently used

CACHE COHERENCE

- We have to be careful with caching write operations!
  - System memory is used by several components in the computer
    - We can have multiple CPUs / more cores
    - I/O peripherals (PCI, DMA, etc.) can use it, too
  - When modifying a cache block, the content of the cache will not be consistent with the memory any more
- Strategies to follow when modifying data in the cache:
  - **Write-through**

    → the modification of the cache immediately implies updating the system memory as well
  - **Write-back**

    → the data is modified in the cache only, the system memory is modified only when the block leaves the cache
    - In this case the content of the memory and the cache will be different!

MULTI-LEVEL CACHES

- Smaller cache size → lower latency (signal paths are shorter)
- Multiple cache levels are used: L1, L2, L3, …
  - Size: increases
  - Speed: decreases
  - In case of L1 miss, the L2 is checked, then L3, etc.
  - They can have different block sizes, organizations, management
- They have different priorities:
  - **The main priority of L1 cache: lowest possible latency**
    - Small size, low associativity
  - **The main priority of Ln cache (n>1): lowest possible miss ratio**
    - Larger size, higher associativity

DEPARTMENT OF
NETWORKED SYSTEMS
AND SERVICES

MŰEGYETEM 1782