Mestrado em Engenharia Informática

2024/25

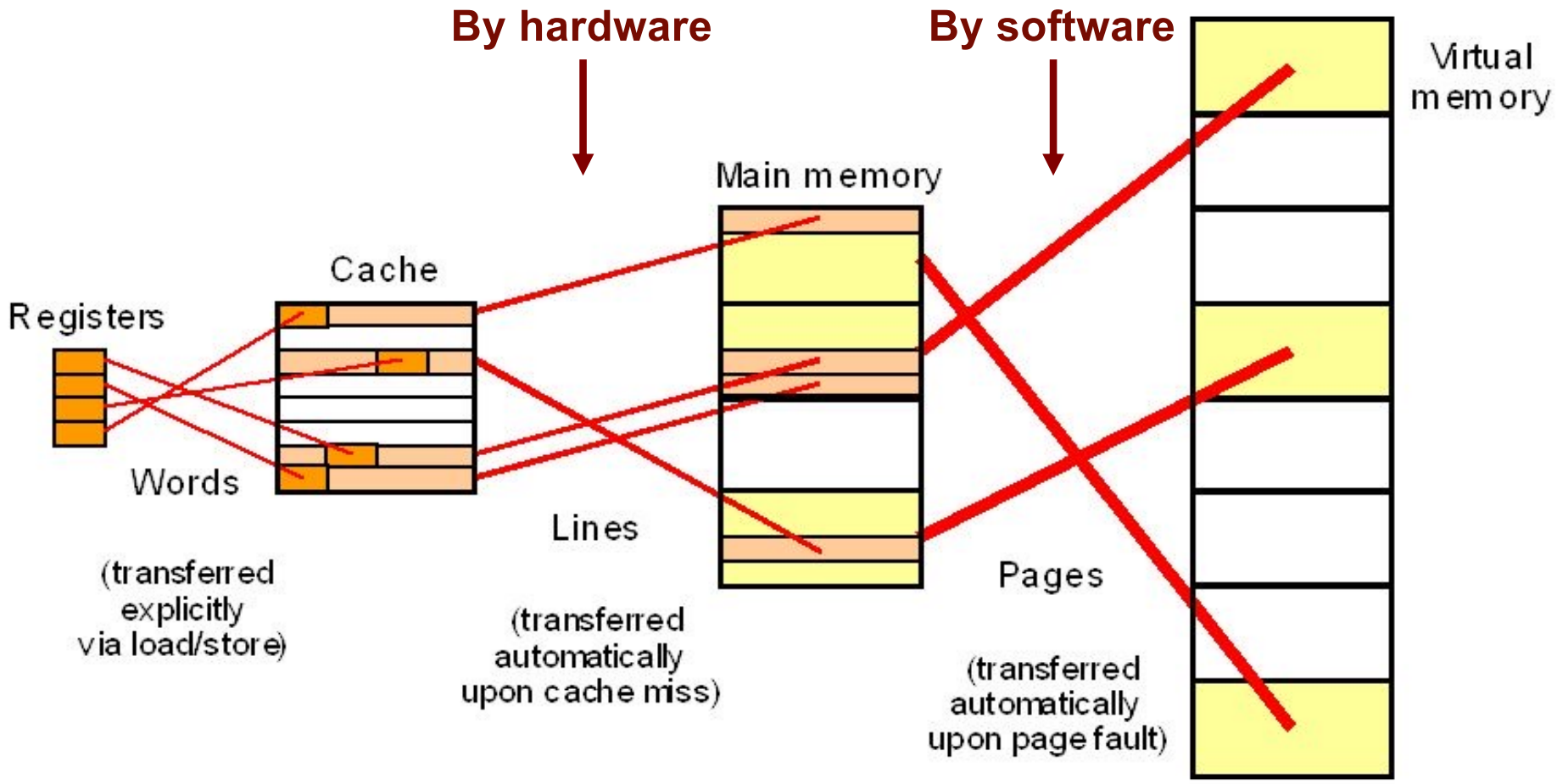
J.L. Sobral

Memory Hierarchy

Memory hierarchy: locality

- **Locality principle**
  - "Programs tend to access a limited portion of memory in a given period of time"
  - allows faster *[local]* memory to be used to store the most frequently/recently used information
  - takes advantage of memory bandwidth, since the information transferred between different levels of the hierarchy is done in blocks
- Spatial locality
  - if a memory element is accessed by the PU, then elements with nearby addresses will most likely be accessed in the near future
- Temporal locality
  - A memory element accessed by the PU will most likely be accessed again in the near future

Memory hierarchy: the big picture



Data movement in a memory hierarchy.

Memory hierarchy: terminology

- Line – The cache is divided into lines. Each line has its address (index) and has the capacity of one block
- **Block** – The amount of information that is transferred at a time from central memory to the cache (or between cache levels). Equals line capacity
- Hit – when the memory element accessed is in the cache.
- **Miss** – A miss is said to have occurred when the memory element accessed by the PU is not in the cache, and it needs to be read from the lower level of the hierarchy.
- Miss rate – Percentage of misses that occurred in relation to the total number of memory accesses.
- **Miss penalty** – The time it takes to load a block from central memory (or a lower level) to the cache when a miss occurs.

Memory hierarchy: performance

$$T_{exec} = \#I * CPI * T_{cc} \quad ou \quad T_{exec} = \#CC * T_{cc}$$

- How does memory hierarchy influence T_{exec}?

- Each memory access will lead to additional clock cycles ($\#CC_{MEM}$) in the execution of the program due to the cache misses:

$$T_{exec} = (\#CC_{CPU} + \#CC_{MEM}) * T_{cc}$$

- Each *miss* implies an increase in the #CC in *miss penalty* cycles, therefore:

$$\#CC_{MEM} = n^{o} miss * miss\ penalty$$

Memory hierarchy: performance

$$T_{exec} = (\#CC_{CPU} + \#CC_{MEM}) * T_{cc}$$

$$\#CC = \#I * CPI$$

$$T_{exec} = \#I * (CPI_{CPU} + CPI_{MEM}) * T_{cc}$$

- **CPI_{CPU} –** number of clock cycles that the processor needs, on average, to execute each instruction
- **CPI_{MEM}** – number of clock cycles that the processor stops, on average, waiting for data from memory, because it did not find this data in the cache.

- Next challenge:
  - how to compare algorithms when $\#CC_{MEM} >> \#CC_{CPU}$?

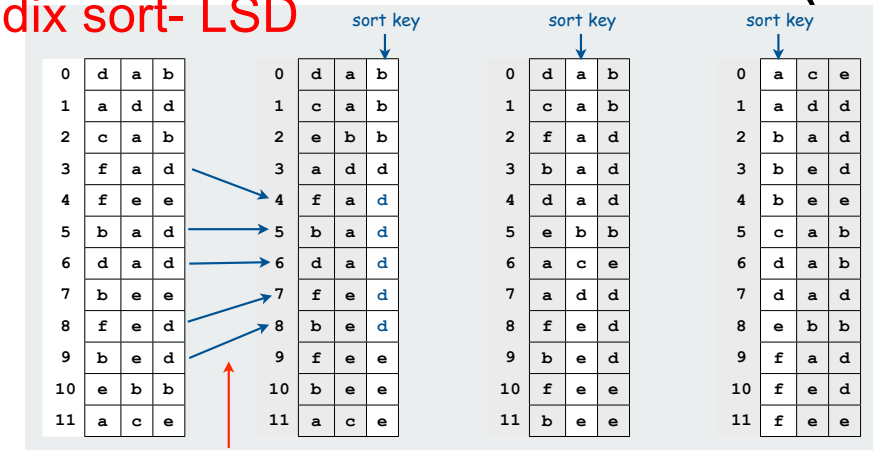
Memory hierarchy: design of algorithms

- How to predict/compare algorithm performance?
- Traditional approach:

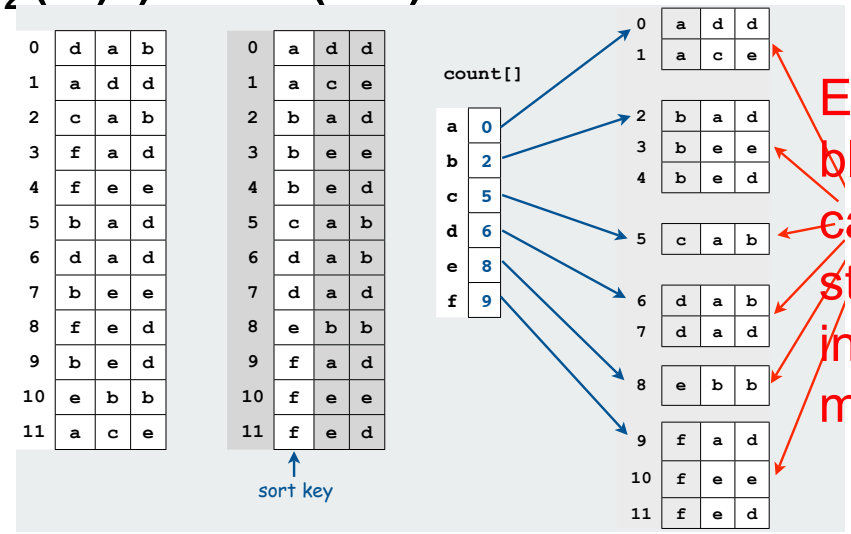
  "big O" notation: - number of operations
  - matrix multiplication: $O(n^3)$
  - sorting: simple approach - $O(n^2)$;

    more efficient - O (n $\log_2(n)$) or O(kn)

Radix sort- LSD



Radix sort- MSD



Each block can be stored in faster memory

Memory hierarchy: design of algorithms

- The traditional complexity model does not consider the impact of the memory hierarchy:
  - is based on a RAM (Random Access Machine) model, where the memory is considered "flat" and with an uniform access time

- **External memory model**
  - Considers two memory levels: external and internal (cache) - External memory is divided into blocks
  - The complexity of the algorithm is expressed in terms of the number of blocks transferred between the two memory levels (i.e. the number of cache misses)

External memory model

example: sum a vector of N contiguous elements in memory

```
1   int sumvec(int v[N])
2   {
3       int i, sum = 0;
4
5       for (i = 0; i < N; i++)
6           sum += v[i];
7       return sum;
8   }
```

implies the transfer of N/B blocks between memory and cache

complexity is O(N/B)

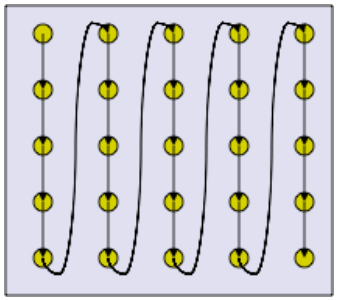
for 16-byte blocks (4x 32-bit integers) we will have N/4 cache misses:

v[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$	$i=7$
Access order, [h]it or [m]iss	1 **[m]**	2 [h]	3 [h]	4 [h]	5 **[m]**	6 [h]	7 [h]	8 [h]

Making Code "Friendly" of the Memory Hierarchy

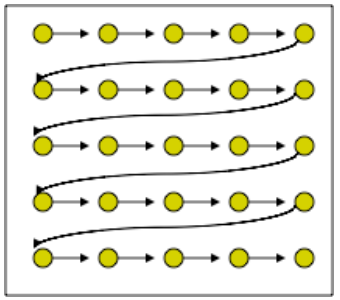
- How to make the implementation more cache-friendly?
  - Minimize the number of blocks transferred between the various memory levels (i.e., the number of cache misses)

- Code/data reorganization to improve spatial locality
  - Reorganization of code blocks
    - Arrange blocks in memory in the order in which they are executed
  - Reorganize data access
    - Access data elements in the order they are stored



Before	Atfer

for (j = 0; j < 100; j = j+1)
  for (i = 0; i < 5000; i = i+1)
  x[i][j] = 2 * x[i][j];
	for (i = 0; i < 5000; i = i+1)
	  for (j = 0; j < 100; j = j+1)
	  x[i][j] = 2 * x[i][j];



Transfers 5000*100 Blocks

Transfers 5000*100/B blocks

Making Code "Friendly" of the Memory Hierarchy

Process the data by blocks (which fit in the cache) to improve the spatial/temporal locality in accessing the data:

Accessing/storing data in blocks, example: transposing an array

Befora	After

for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
  dst[i][j] = src[j][i];
	for (int ii = 0; ii < N; ii += blocksize)
	  for (int jj = 0; jj < N; jj += blocksize)
	  // transpose the block beginning at [ii,ji]
	  for (int i = ii; i < ii + blocksize; i++)
	  for (int j = jj; j < jj + blocksize; j++)
	  dst[i][j] = src[j][i];

src

1	1						
2	2						
3	3						
4	4						
5	5						
6	6						
7	7						
8	8						

dst

1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

src

1	1	1	1				
2	2	2	2				
3	3	3	3				
4	4	4	4				

dst

1	2	3	4				
1	2	3	4				
1	2	3	4				
1	2	3	4				