Master Program (Laurea Magistrale) in Computer Science and Networking

High Performance Computing Systems and Enabling Platforms

Marco Vanneschi

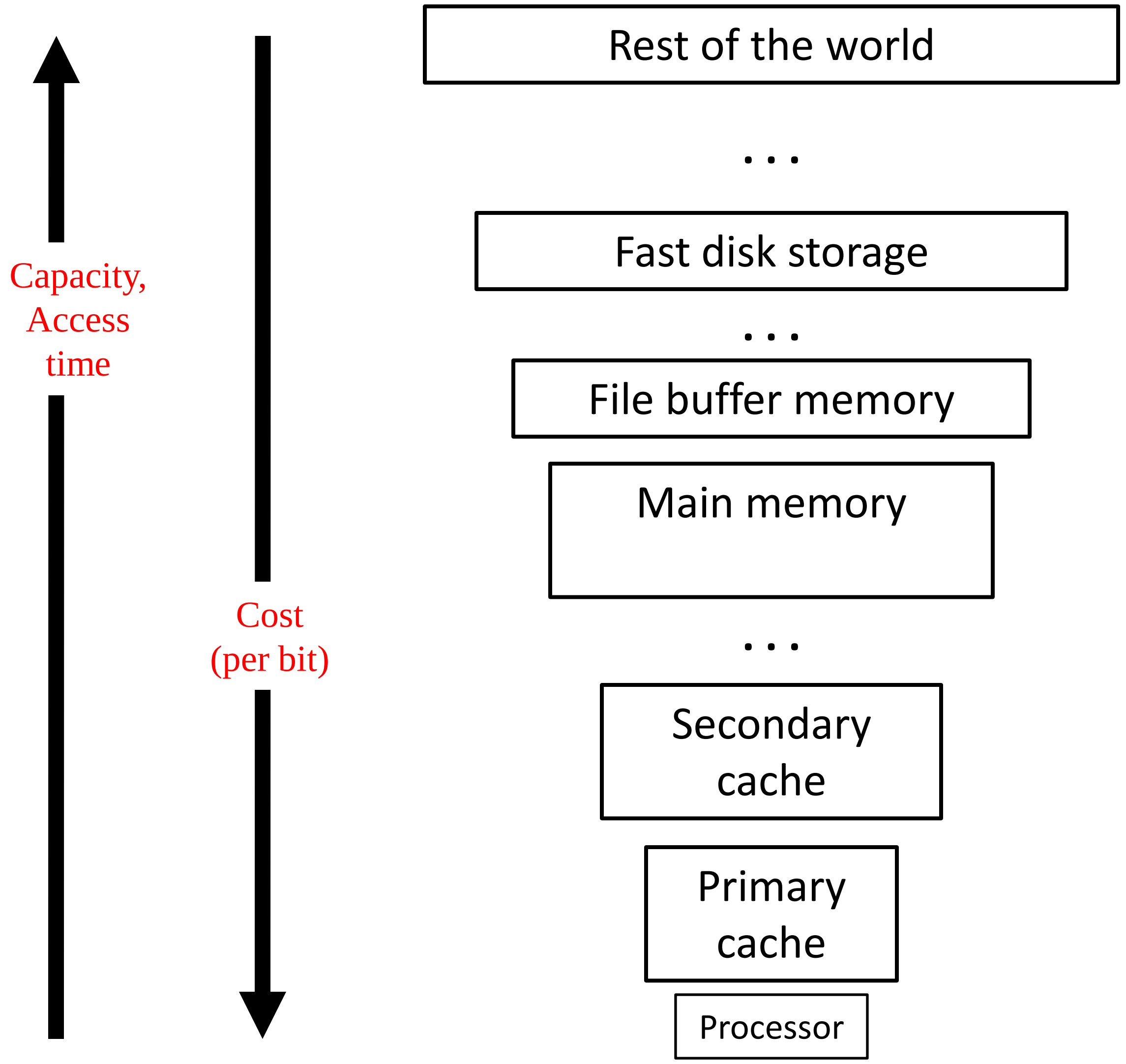
1. Prerequisites Revisited

1.4. Memory Hierarchies and Caching

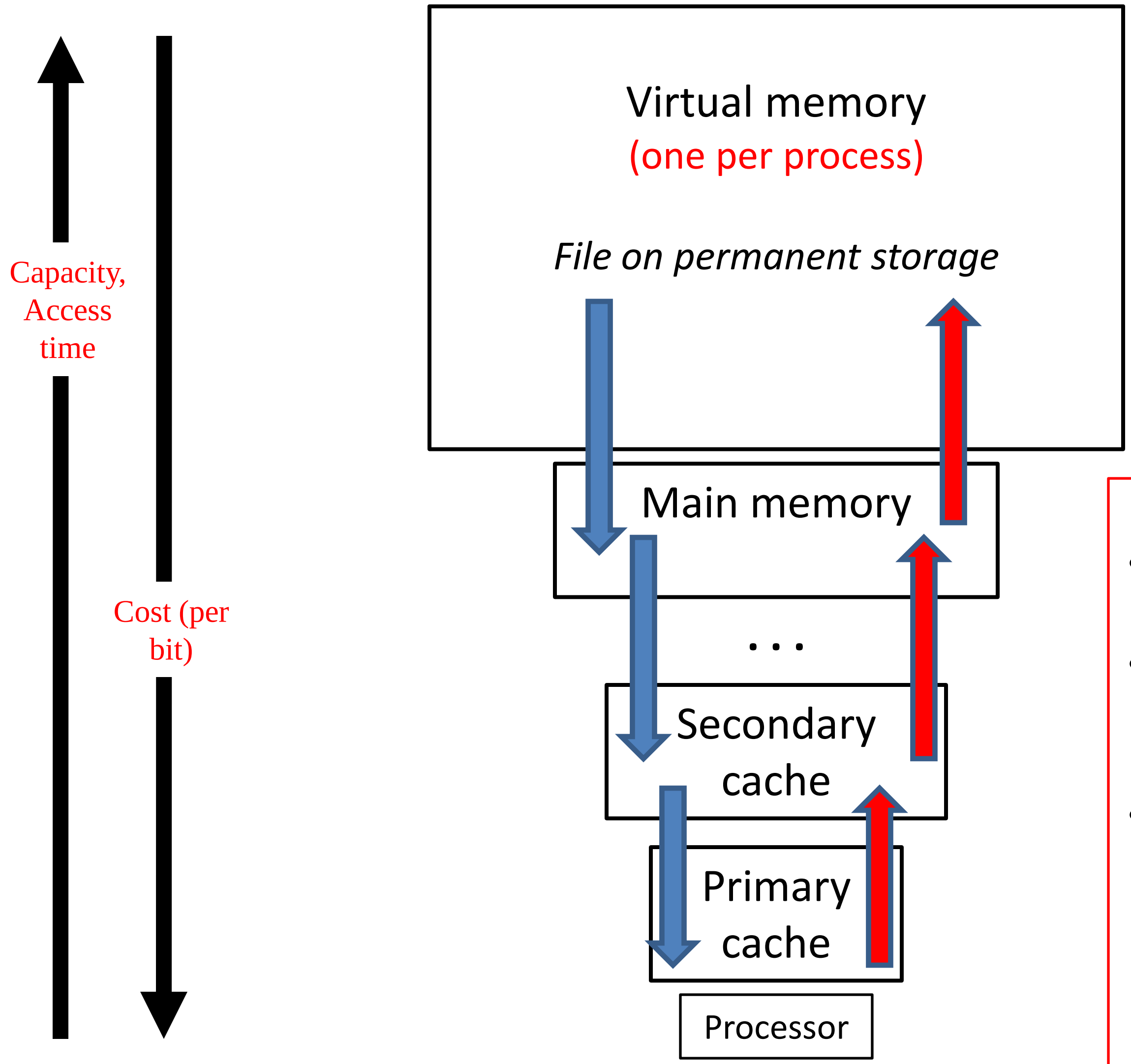
8 March
Women's Day

with best wishes !

Memory hierarchies



Memory hierarchies and performance issues



For our purposes

Virtual Memory: logical memory of the process

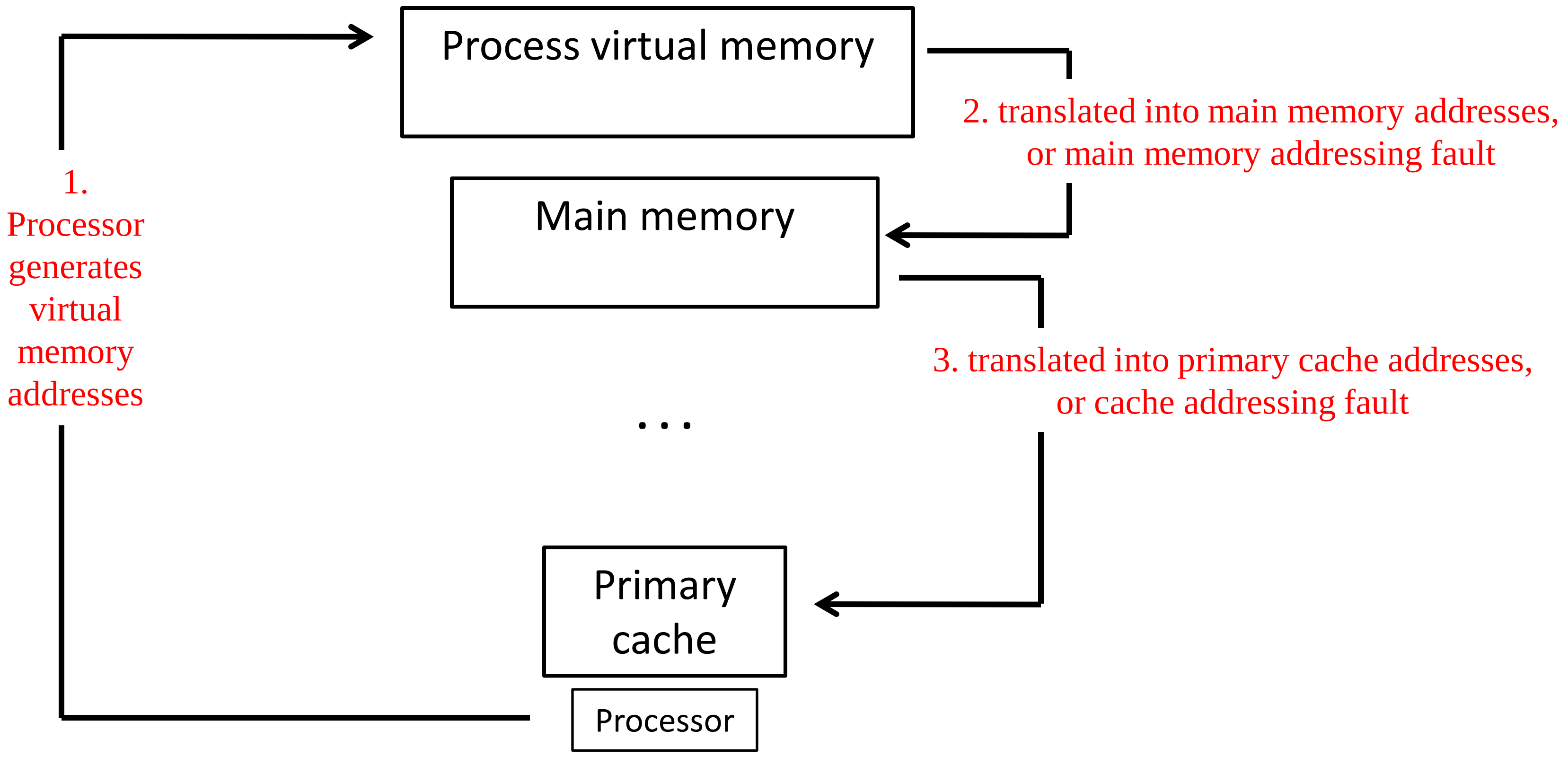
- **Logical address space**
- Allocated , and possibly initialized, **at compile time**
- Possibly re-allocated at run-time (data creation / destruction)

DYNAMIC ALLOCATION

- information transfer between the memory hierarchy levels,
- in order to optimize the access time to the "currently needed" information,
- i.e. to find space in the lowest levels to accomodate the information needed by the process execution.

Performance/cost optimization

Memory hierarchies and address translation

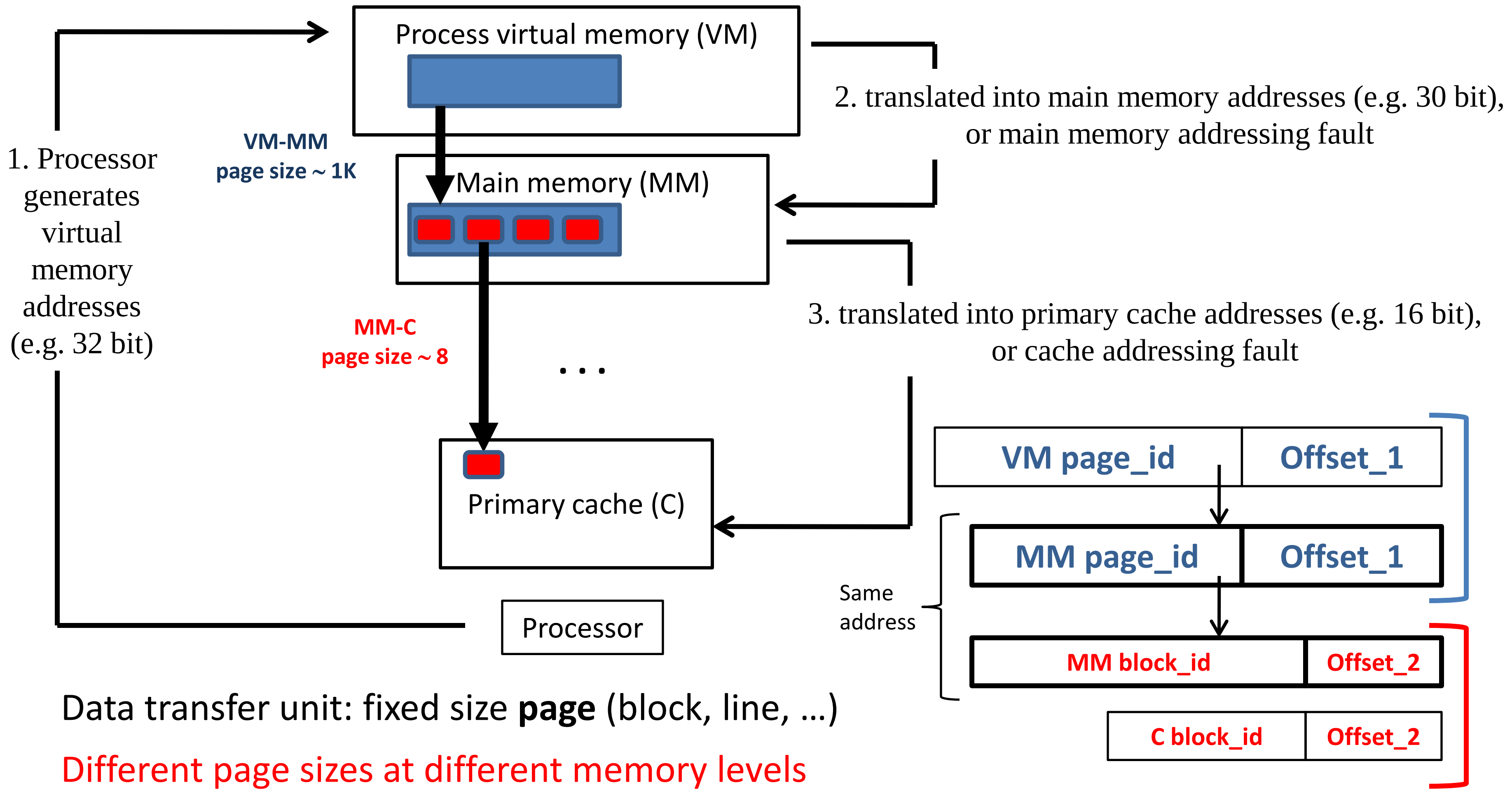

Process virtual memory
Main memory
. . .
Primary cache
Processor
1. Processor generates virtual memory addresses
2. translated into main memory addresses, or main memory addressing fault
3. translated into primary cache addresses, or cache addressing fault

Translation sequence must have very low latency (1 – 3 clock cycles) in case of success (no fault):

- MMU (with associative memory)
- Cache Unit

Fault handling (visible or invisible to processor): allocation of the needed information.

Memory hierarchies with Paging



Process virtual memory (VM)
1. Processor generates virtual memory addresses (e.g. 32 bit)
VM-MM page size ~ 1K
Main memory (MM)
2. translated into main memory addresses (e.g. 30 bit), or main memory addressing fault
MM-C page size ~ 8
. . .
3. translated into primary cache addresses (e.g. 16 bit), or cache addressing fault
Primary cache (C)
Processor
VM page_id
Offset_1
MM page_id
Offset_1
Same address
MM block_id
Offset_2
C block_id
Offset_2

Data transfer unit: fixed size **page** (block, line, …)

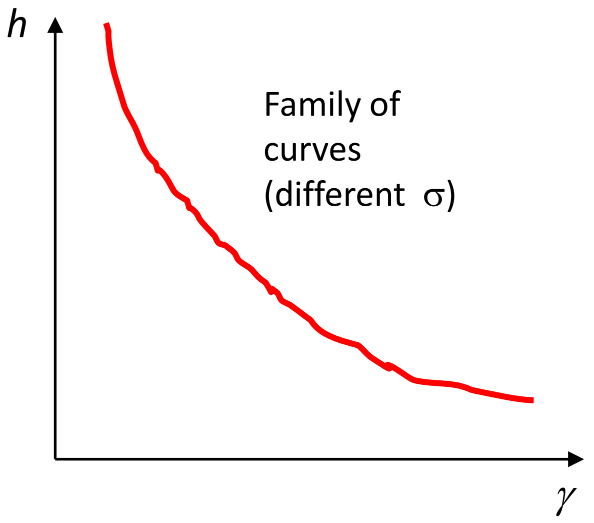
Different page sizes at different memory levels

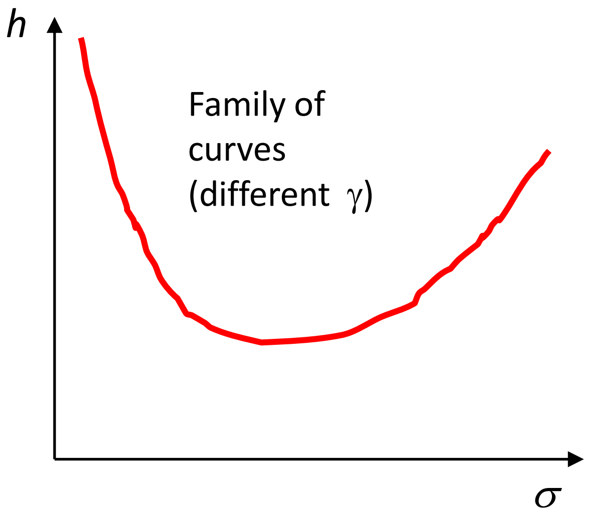
Address translation applied to page identifiers (+ offset concatenation)

Fault probability

- For a given sub-hierarchy M_i - M_{i-1} (e.g. MV-MM, MM-C):
  - Fault (miss) probability, h
  - Page (block, line, …) size, σ
  - Capacity of lower memory level, γ
  - $h = h\,(\sigma, \gamma, \textit{replacement strategy})$

page replacement: e.g. LRU algorithm





Experimental values for large benchmarking program sets in given application areas

Each program has its own fault probability.
Importance of optimization techniques to reduce fault probability.

Paging: motivations

Properties of address sequences in sequential programs:

- **LOCALITY** (spatial locality): references to program information belong to groups of addresses which are close together.
- **REUSE** (temporal locality): some information are referred several times during the program execution.

WORKING SET of a sub-hierarchy:

Pages that, if and when allocated in the lower memory level of the sub-hierarchy, minimize the fault probability (how many and which pages)

In some cases, the compiler is able to recognize the working set and can cause some run-time actions to mantain the working set in the lower memory level of the sub-hierarchy during the program execution.

Cache cost model

Completion time *<u>of a given program</u>*:

$$T_c = T_{c\text{-ideal}} + N_{fault} * T_{block}$$

$T_{c\text{-ideal}}$ = Completion time with no faults; memory access time = cache access time

N_{fault} = Average number of faults for that program

T_{block} = Block transfer time

Relative **efficiency**:

$$\varepsilon_{cache} = T_{c\text{-ideal}} / T_c$$

Examples (e.g. caching)

```
int A[N], B[N];
    for (i = 0; i < N; i++)
        A[i] =  A[i] + B[i];


LOOP:       LOAD   RA, Ri, Ra
            LOAD   RB, Ri, Rb
            ADD   Ra, Rb, Rc
            STORE   RA, Ri, Rc
            INCR   Ri
            IF <   Ri, RN, LOOP
            END
```

Reuse of instructions (loop)

- In general, the cache unit provides reuse (and prefetching) of instructions automatically

Data: locality only

Working set =

- Instructions (e.g., one block)
- **One block of A**
- **One block of B**
- + other information of the *process run-time support* (Process Control Block, Translation Table, run-time code, etc)

Examples (e.g. caching)

```
int A[N], B[N];
    for (i = 0; i < N; i++)
        A[i] =  F (A[i], B);


LOOP:      LOAD   RA, Ri, Ra
           …
           …
           LOAD   RB, Rj, Rb, reuse
           …
           …
           STORE   RA, Ri, Rc
           INCR   Ri
           IF <   Ri, RN, LOOP
           END
```

Compiler annotation:

if provided by the **assembler machine**

if supported by the **firmware machine (Cache Unit)**

Instructions: reuse (loop)

Data:

- A: locality only
- **B:** potential **reuse**

Working set:

- blocks of instructions
- one block of A
- **all blocks of B**, *once referred for the first time*

  if *B size is such that it can be entirely allocated in cache;*

  otherwise, if B is too large, reuse cannot be exploited, or it can be exploited only partially:

  some blocks of B are allocated in cache, the other are allocated one at the time

Reuse optimizations:

when decidable by a static analysis (in this case: yes);

typical non-decidable cases: when reuse opportunities depend on data values.

Write operations

- Write Back
  - A block is re-written in the upper memory level when it is de-allocated
- Write Through
  - Every write operations is carried out into the cache *and* into the upper memory level *in parallel*
  - Effective if the memory bandwidth is greater/equal to the inverse of the average time interval between two consecutive STORE instructions

On-demand *vs* prefetching strategies

- On-demand: page allocation only when a fault occurs
- Prefetching: try to anticipate the page allocation (to be ahead of fault occurrence)
  - Applied to the next page, or to a page to be determined
  - *If applied by default to data*, prefetching could lead to serious inefficiencies (examples of some Intel processors)
- *Compiler annotation,* e.g.

  LOAD RC, Rj, Rc, **prefetching**

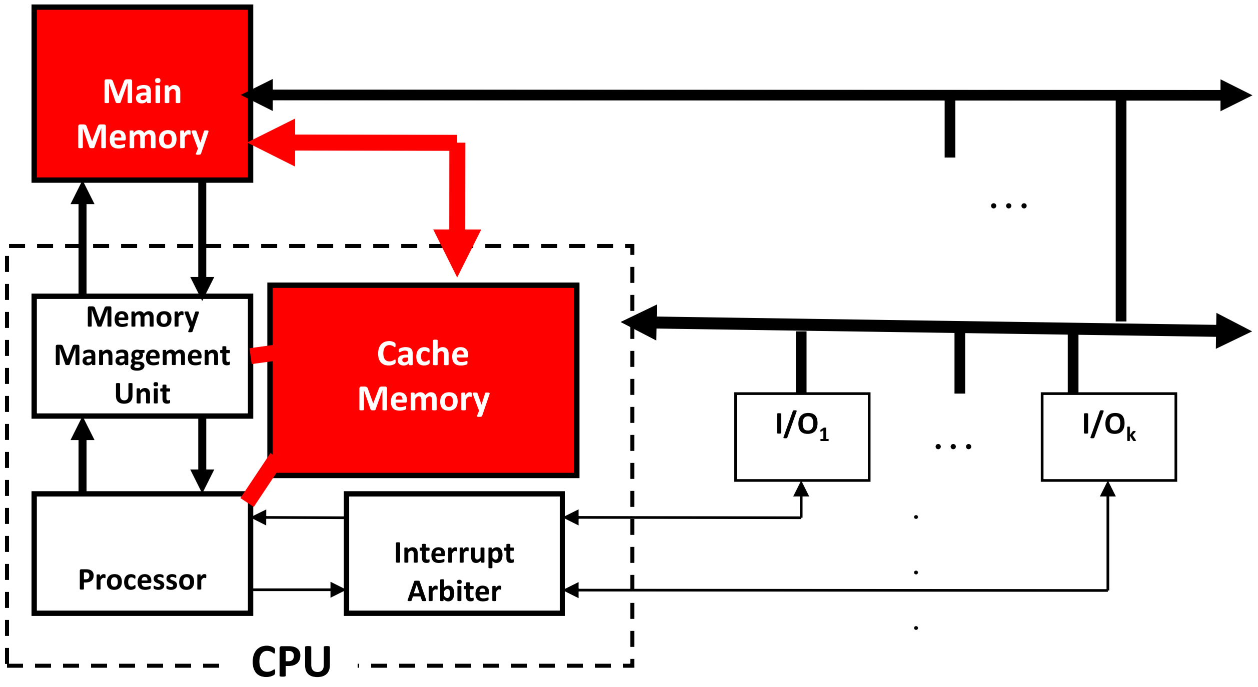
Other caching annotations

- Useful for multiprocessors:
  - explicit de-allocation of blocks
  - explicit re-writing of blocks

- Cache coherence in multiprocessors
  - See Part 2 of the course.

Cache fault handling

- Entirely invisible to the Processor: no exception is generated
    - *On the contrary: an exception is generated in the MV-MM sub-hierarchy, and handled at the process level.*
  - The Cache Unit is responsible of implementing the replacement strategy and the block transfer at the firmware level.



More levels of caching

- In a program characterized by ***locality only*** (or few reuse opportunities), the block transfer from Main Memory to Cache could be inefficient: too large latency
  - *Block transfer latency:*

$$T_{block} = \sigma \; T_{MMaccess}$$

  - The completion time is about the same of the architecture without cache !
  - Even in this architecture, *reuse* can increase efficiency significantly (importance of reuse, again)

- **Interleaved memory:**
  - **m** independent MM modules: $M_0, \ldots, M_k, \ldots, M_{m-1}$
  - Address **j** refers module **k**: **k = j *mod* m**
  - *Parallel access to* **m** *consecutive MM locations* (**MM bandwidth** is increased by a factor **m**)
  - *Block transfer latency:*

$$T_{block} = 2\,T_{tr} + \frac{\sigma}{m}\,\tau_M + m\,\tau$$

Interleaving could be not sufficient to solve the problem, if MM clock cycle and link latency are too large.

More levels of caching

SECONDARY CACHE (C2):

- Capacity and block size: one order of magnitude larger than Primary Cache (C1)
- Information of more than one process in C2
- Block transfer latency:
  - If C2 is **out of** CPU chip: **m-interleaved** static memory

    same formula for T_{block}, where now τ_M of C2 << MM clock cycle.
  - If C2 is **on-chip**: just one memory module is sufficient to achieve a good efficiency

    $$T_{block} \sim \sigma\, \tau$$

    applying *overlapping* of C2 read operations and C1 write operations.

Cache cost model: an example

```
int A[N], B[N];
    for (i = 0; i < N; i++)
        A[i] =  F (A[i], B);
```

```
LOOP:      LOAD   RA, Ri, Ra
           ...
           ...
F          LOAD   RB, Rj, Rb, reuse
           ...
           ...
           STORE   RA, Ri, Rc
           INCR   Ri
           IF <   Ri, RN, LOOP
           END
```

Completion time:

$$T_c = T_{c\text{-ideal}} + N_{fault} * T_{block}$$

$T_{c\text{-ideal}}$ **= Completion time with no faults**

N_{fault} **= Average number of faults for the program**

T_{block} **= Block transfer time**

Let: $T_F = k\,\tau$

Then: $T_{c\text{-ideal}} \sim N\,T_F = k\,N\,\tau$

If **B <u>reuse</u>** can be applied:

$$N_{fault} = N_{fault\text{-instructions}} + N_{fault\text{-A}} + N_{fault\text{-B}}$$
$$\sim N_{fault\text{-A}} + N_{fault\text{-B}} = N/\sigma + N/\sigma = 2\,N/\sigma$$

With <u>C2 on-chip</u>: $T_{block} \sim \sigma\;\tau$

Then: $T_c \sim k\,N\,\tau + 2\,N\,\tau \sim T_{c\text{-ideal}}$ for $k >> 2$

Efficiency: $\varepsilon_{cache} = T_{c\text{-ideal}} \,/\, T_c \sim 1$

Effect of faults is not significant, with the applied assumptions and optimizations.

Caching techniques

Mapping function of MM block identifier (MB) into C block identifier (CB):

CB = *mapping_function* (MB)

1) DIRECT CACHE: CB = MB *mod* NC

where NC = number of blocks in C

+ logic for verifying the possible fault

2) FULLY ASSOCIATIVE CACHE: CB = Block_Table (MB)

Random Association - Table Lookup implementation

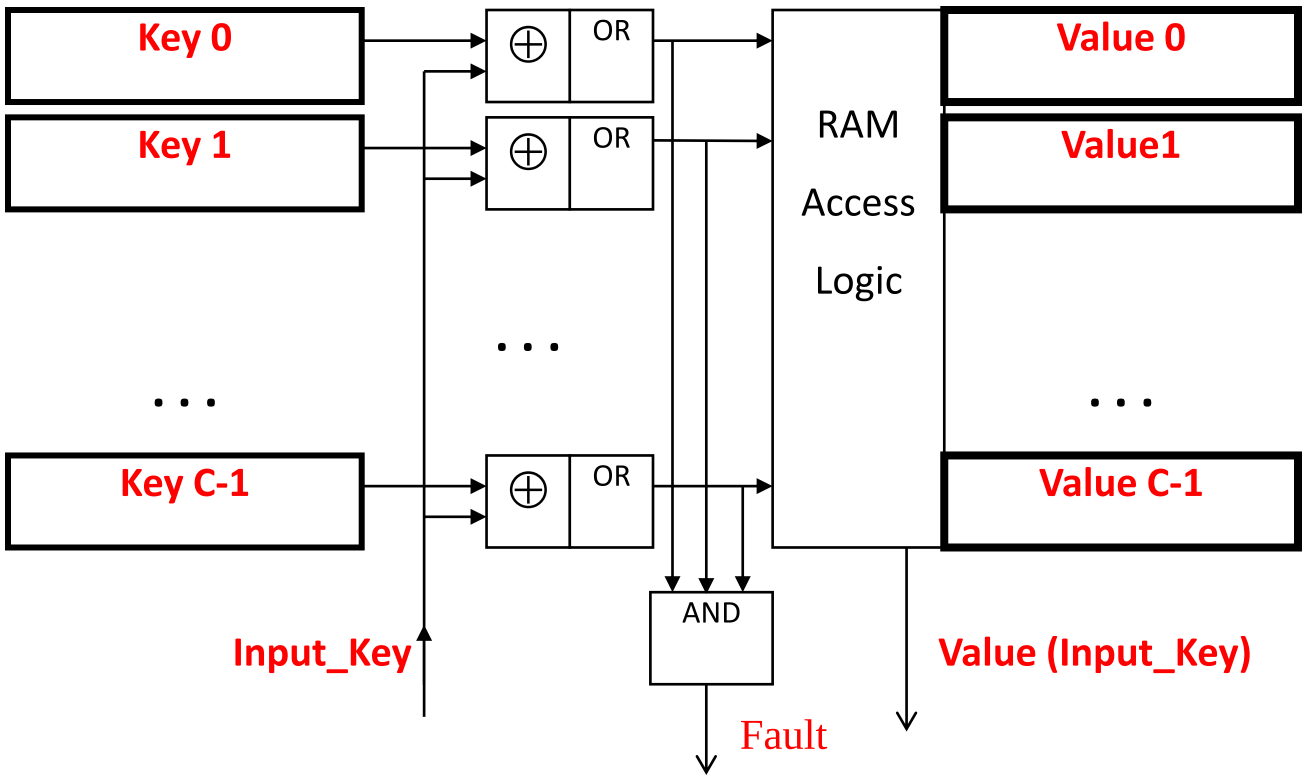
Associative Memory

Direct Cache: rigid association, leading to increased fault probability in some programs; cheap realization; lowest latency in case of success

Fully Associative: most general and most flexible, LRU applied for block replacement, at least one more clock cycle of latency

Associative Memory (also for MMU)

Hardware implementation of a Content-Addressable Table



Caching techniques

3) Trade-off: SET ASSOCIATIVE CACHE

Cache blocks are partitioned into **SETS**, e.g. 4 blocks per set.

Let **NS** = number of cache sets.

Identifier of Cache Set where the Memory Block *may* reside:

$$\text{SET} = \text{MB} \; \textit{mod} \; \text{NS}$$

Inside this Set, the Memory Block can be allocated anywhere**: Associative Technique is applied to the blocks in each Set.**

- Simple implementation (no Associative Memory).
- Performances (fault probability) similar to Fully Associative Cache, LRU applied to Set blocks, same latency in case of success.

TYPICAL ARCHITECTURE:

- **Instruction Cache**: Direct
- **Data Cache**: Set (Fully) Associative

Test 1:
to be submitted and discussed at Question Time

PRIMARY and SECONDARY CACHE

- Assume that Secondary Cache resides on CPU chip
- Why distinguishing between Primary Cache (L1) and Secondary Cache (L2) ?
- *i.e.*, why not just one level of cache only, with larger capacity (sum of capacity of L1 and of L2) ?

As usually: give a convincing answer from a methodological viewpoint (cost model, opportunities, utilization policies, and so on), … not because existing processors have both L1 and L2!

Test 2: to be submitted and discussed at Question Time

CACHING and I/O

- Memory Mapped I/O: can CPU-caching be applied efficiently to information allocated in the I/O Memories?

- Evaluate the efficiency (or other parameters) with an I/O Bus structure
  - *hint:* for the sake of the cost model, assume that the I/O Bus adopts a RDY-ACK communication protocol with level-transition interfaces as for dedicated links (See: Firmware Prerequisites) – although this assumption is not formally correct;
  - does the evaluation depend on the *reuse* opportunities of I/O-mapped information ? (give the answer wrt this specific question, not in general).

- Find possible I/O architectures (suitable interconnection structure and memory organization) able to exploit CPU-caching at best when Memory Mapped I/O is adopted.