Advance Computer Architecture and x86 ISA

University of Science and Technology of Hanoi

MS. LE Nhu Chu Hiep

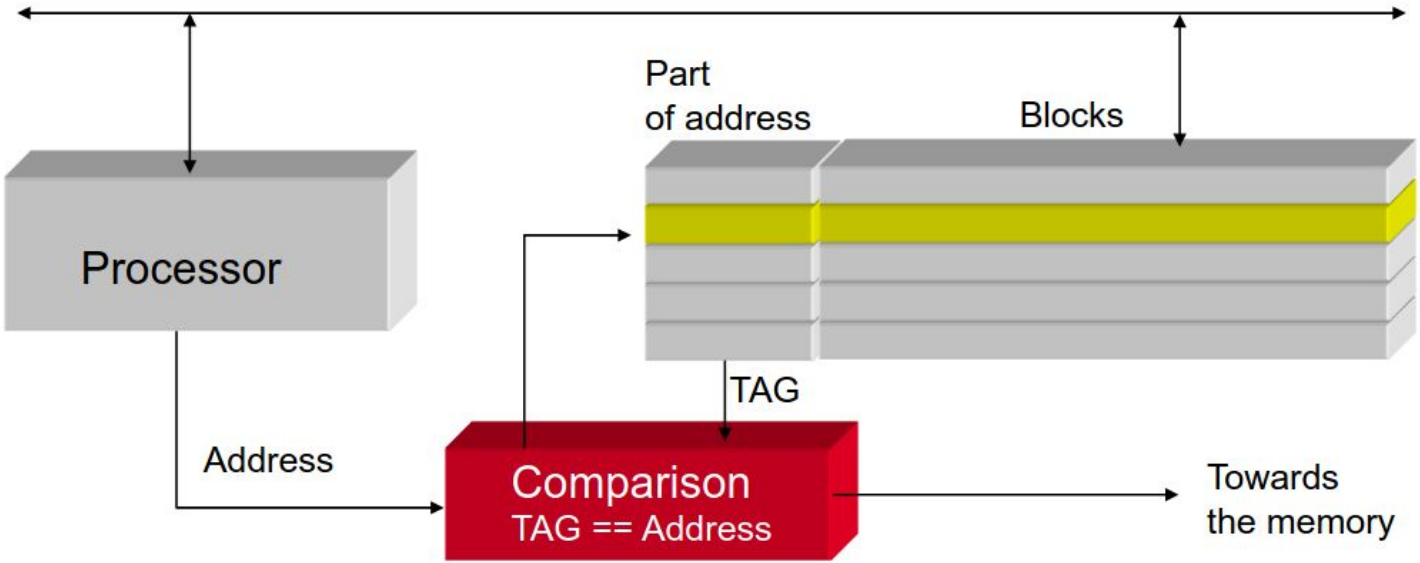
Cache

Cache Memory

- Fast memory
  - Access time and cycle time of processor are very closed
- Cache taxonomy
  - **Cache miss**: A cache default appears when the data required by the processor is not in the cache memory
  - **Cache hit**: A cache appears when the data required by the processor is in the cache memory
- The memory cache copies some small blocks of main memory

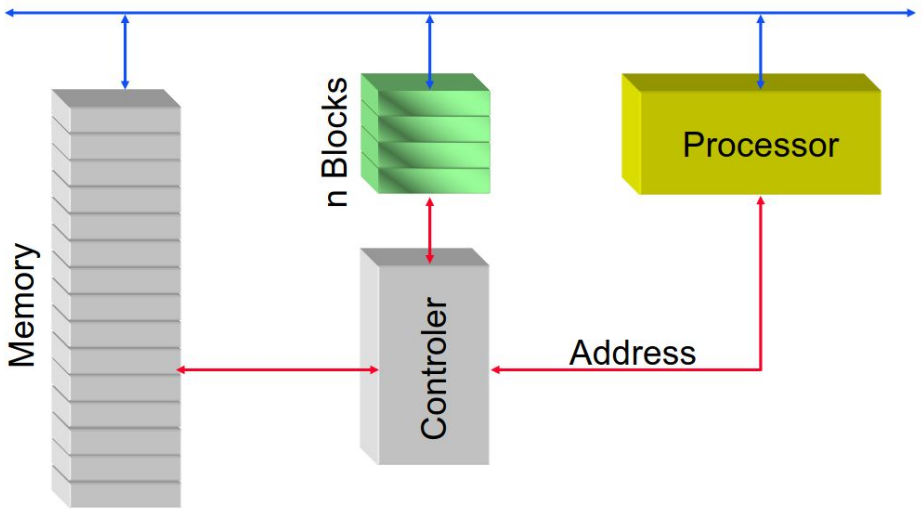
Cache blocks and memory blocks

- Data (or blocks of data)
- addresse (part of address)



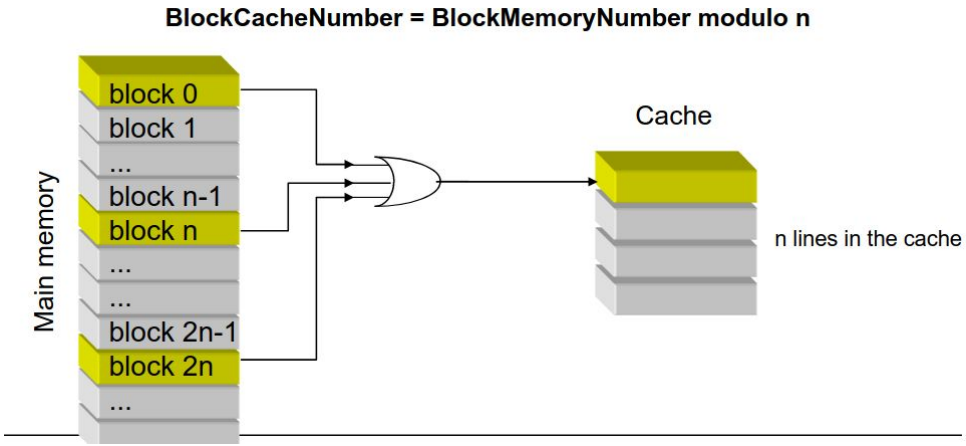
Cache blocks and memory blocks (cont.)

- The cache memory is divided into n blocks
- The main memory is divided into N blocks
- N >> n



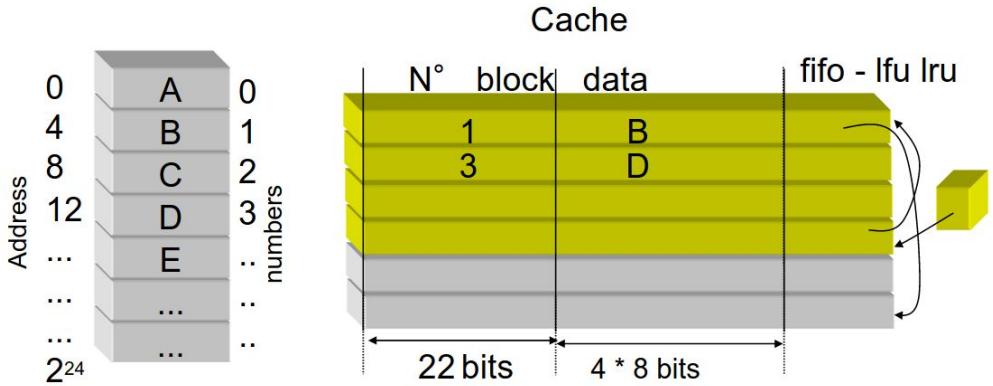
Cache block organization

- Cache memory with direct associativity
  - A bock have a fix location in the cache
  - Simple technique, management of storage needs small logic
  - No optimal technique to replace a block
  - The block i from memory is stored in the cache in the block



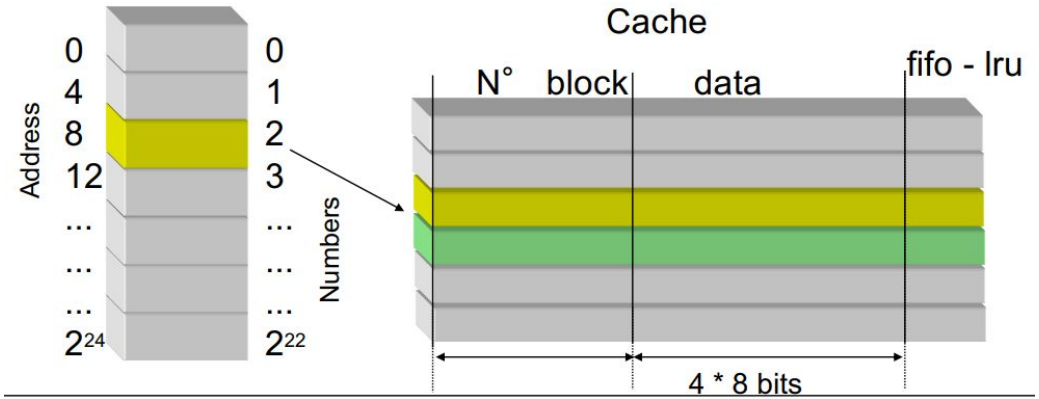
Cache block organization (Associative memory)

- A block can have different location in the cache, it depends on the state of cache when a new block must be stored
- Enable to implement a more adapted replacement policy
  - Random : Random replacement
  - Fifo : First in first out
  - Lru : The last recently used
  - Lfu : The least frequently used



Cache block organization (N-way Associative memory)

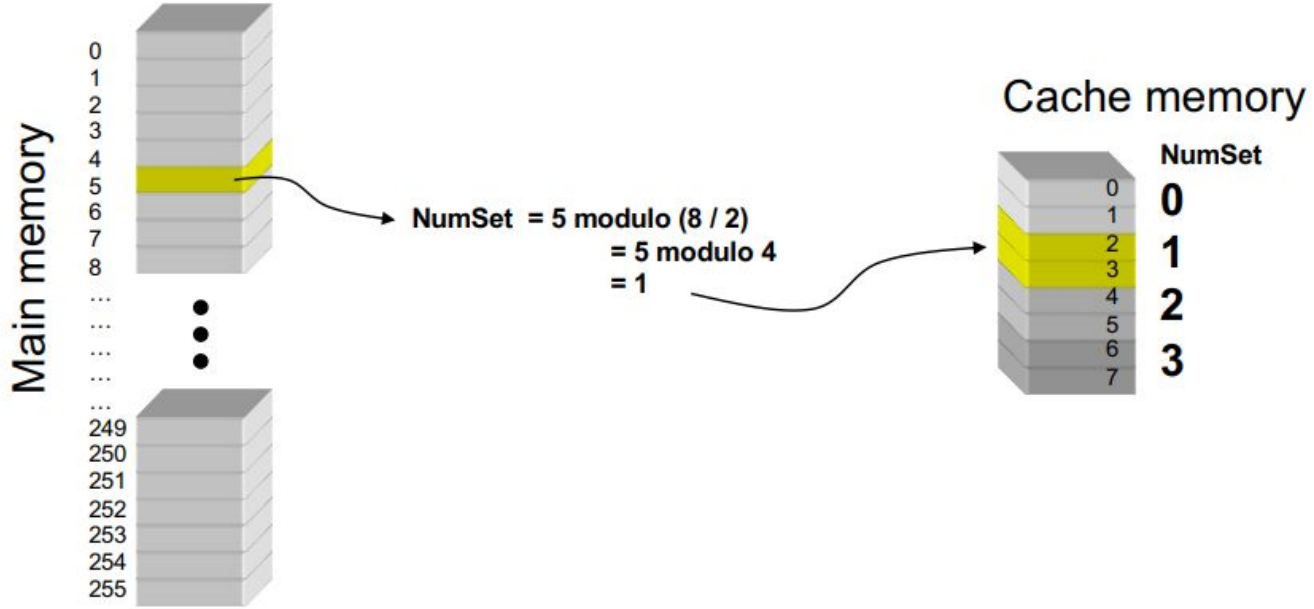
- A block can have N locations in the cache memories
- Simplification of control
- Generally sufficient to ensure good performance
- Policies to replace in set of clocks
  - Random : Random replacement
  - Fifo : First in first out
  - Lru : The last recently used
  - Lfu : The least frequently used



Example: find a new location in the cache

- If N if the number of the main memory block
- If L is the number of cache lines
- If V is the number of way for the associativity
- Then
- NumSet = N mod (L/V)
- The numbers of possible blocks are:
    - NumSet * V <= NumBlocksCache <= (NumSet + 1) * V -1

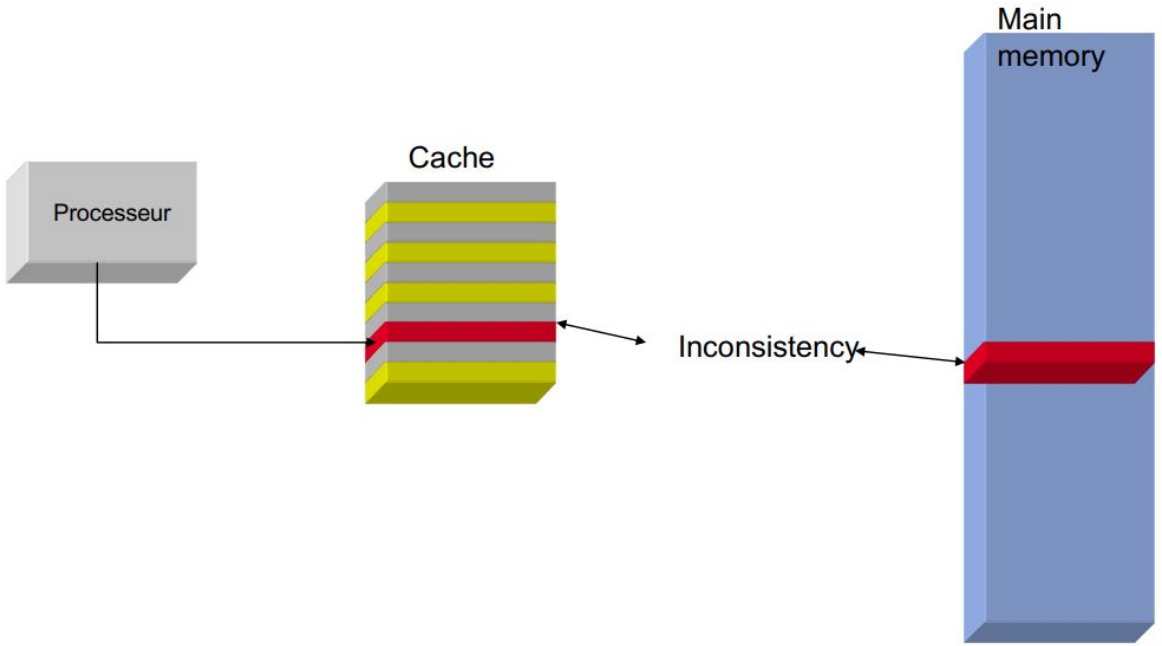
Example: find a new location in the cache (cont.)



Cache Synchronize

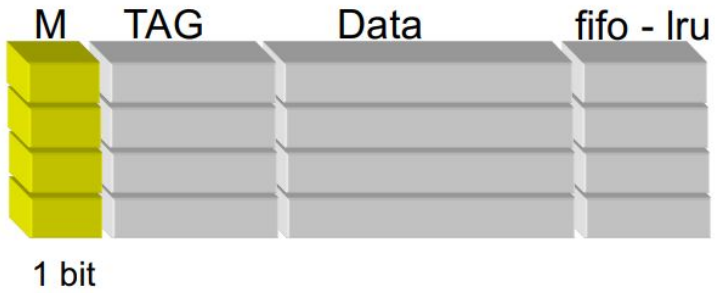
Policy for writing in the main memory

- When the processor modifies a value stored in a cache block



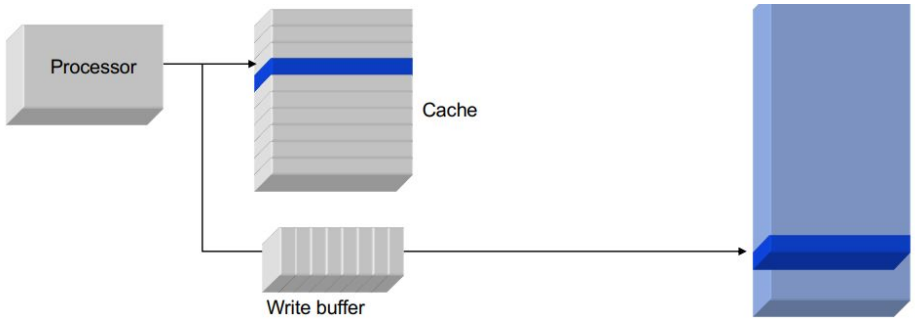
Policies to update the main memory (Write back)

- Write back
  - Update is done when block replacement
  - The memory and cache can stored inconsistency blocks
  - A cache miss on modified block is costly
    - Need a block copy to main memory, then a copy from memory of the new block
  - The cache stores the information about block modification: bit M



Policies to update the main memory (Write through)

- Write through
  - Update of main memory is done at each write operation in the cache
  - The main memory and the cache are always consistent
  - No need a bit to store inconsistency
  - The memory access bus is overload by all the write operation
    - These write operations are slow
    - Solution: a write buffer places between processor and main memory



Policies for writing a data which is not in cache

- Write through, write allocate
  - The block is placed in cache, the write operation is done in the cache
    - The penalty is the same as classical miss
    - The block must be copy from memory to cache, then vice versa
- Write through, no write allocate
  - The write operation is done in the main memory, not in cache
    - The next write operation in the same blocks provokes a new write operation in the main memory
- Write back
  - The write operation is done when the block is ejected from the cache memory

		Write in cache memory	
		Yes	No
Write in main memory	Yes	Write through Write allocate	Write through No write allocate
	No	Write Back	

Cache memory simulations

- Example: matrix product (2 matrices 32 * 32)
  - Unified cache memory
  - Total number of access: 269476
  - Cache memory size 1024 octets
  - Simulations for cache having
    - The following block sizes:
      - 4, 8, 16, 32, 64, 128
    - The following replacement policies:
      - Direct, LRU, FIFO, Random
    - The following associativities (between 1 to Nb Blocks, always power of 2)

```
for (i = 0 ; i < N ; i++) {
    for (j = 0 j < N ; j++) {
            c[i][j] = 0;
            for (k = 0 ; k < N ; k++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j] ;
            }
    }
}
```

Example: the program of matrix product

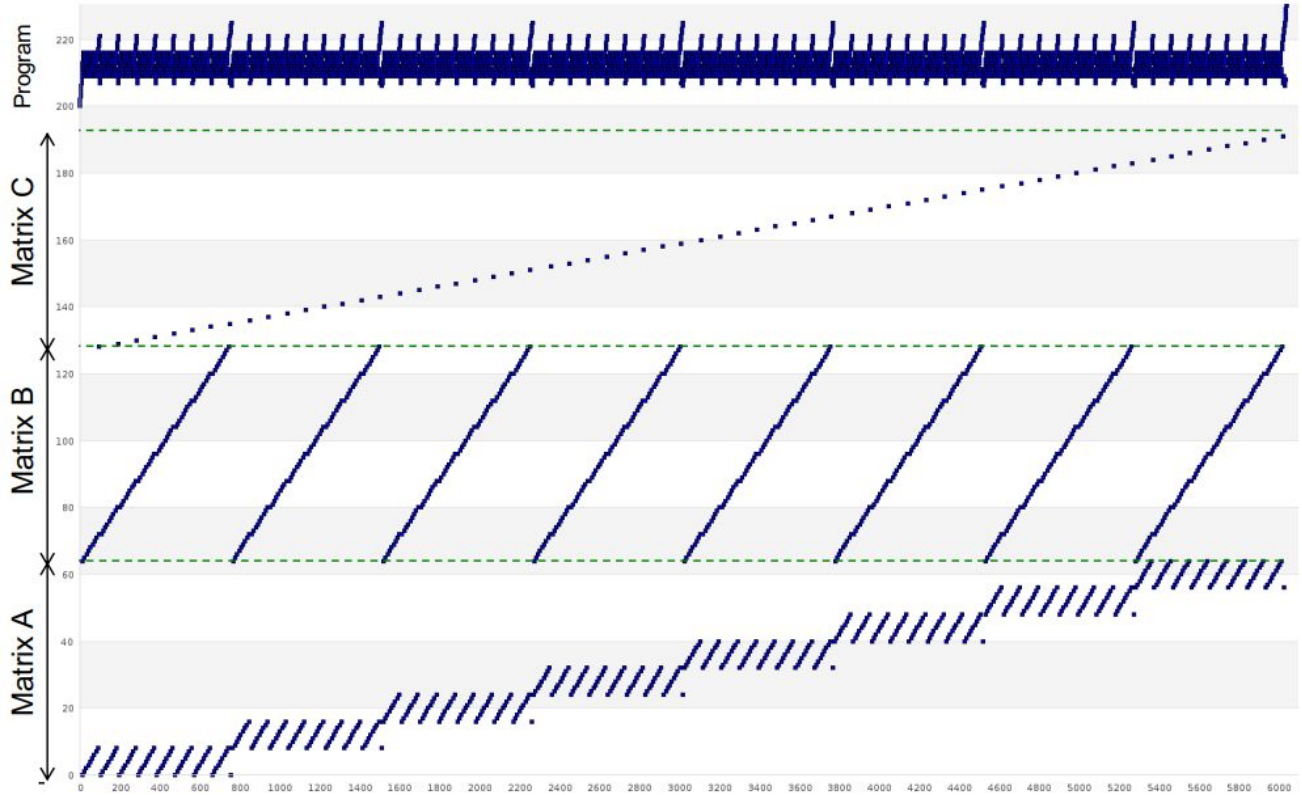
- The program

```
1600        MOVE R1, @a00
            MOVE R2, @b00
            MOVE R3, @c00
            MOVE R4, N
boucle1
            MOVE R5, N
boucle2
            MOVE R6, N
            MOVE R7, 0
boucle3
            MOVE R8, (R1)++
            MOVE R9, (R2)++
            MULT R8, R9
            ADD  R7, R8
            SUB  R6, 1
            BRnz boucle3
            MOVE (R3)++, R7
            SUB  R1, N
            SUB  R5, 1
            BRnz boucle2
            MOVE R2, @b00
            ADD  R1, N
            SUB  R4, 1
            BRnz boucle1
```

- Access list

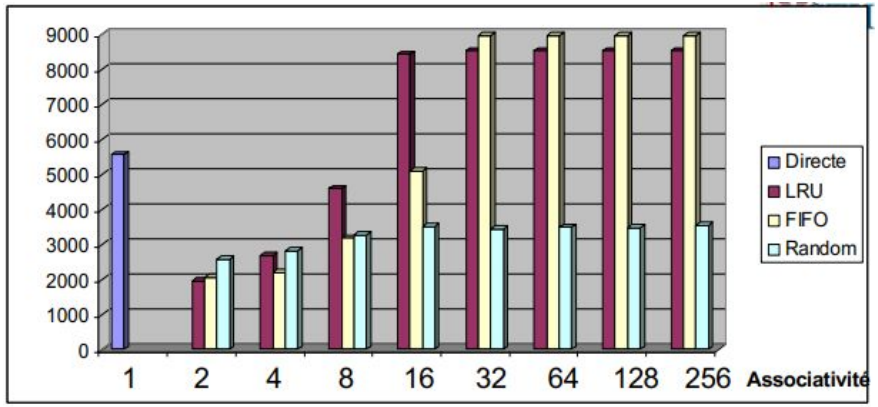
```
2 640 -- Acces a l'instruction MOVE R1,@a00
2 641 -- Acces a l'instruction MOVE R2,@b00
2 642 -- Acces a l'instruction MOVE R3,@c00
2 643 -- Acces a l'instruction MOVE R4,N
2 644 -- Acces a l'instruction MOVE R5,N
2 645 -- Acces a l'instruction MOVE R6,N
2 646 -- Acces a l'instruction MOVE R7,0
2 647 -- Acces a l'instruction MOVE R8,(R1)++
0 0 --- Acces A[0][0]
2 648 -- Acces a l'instruction MOVE R9,(R2)++
0 40 --- Acces B[0][0]
2 649 -- Acces a l'instruction MULT R8,R9
2 64a -- Acces a l'instruction ADD R7,R8
2 64b -- Acces a l'instruction SUB R6,1
2 64c -- Acces a l'instruction BRnz boucle3
2 647 -- Acces a l'instruction MOVE R8,(R1)++
0 1 --- Acces A[0][1]
2 648 -- Acces a l'instruction MOVE R9,(R2)++
0 41 --- Acces B[1][0]
2 649 -- Acces a l'instruction MULT R8,R9
2 64a -- Acces a l'instruction ADD R7,R8
...
...
2 649 -- Acces a l'instruction MULT R8,R9
2 648 -- Acces a l'instruction MOVE R9,(R2)++
0 47 --- Acces B[7][0]
2 649 -- Acces a l'instruction MULT R8,R9
2 64a -- Acces a l'instruction ADD R7,R8
2 64b -- Acces a l'instruction SUB R6,1
2 64c -- Acces a l'instruction BRnz boucle3
2 64d -- Acces a l'instruction MOVE (R3)++,R7
1 80 --- Acces C[0][0]
2 64e -- Acces a l'instruction SUB R1,N
...
...
2 646 -- Acces a l'instruction MOVE R7,0
2 647 -- Acces a l'instruction MOVE R8,(R1)++
0 0 --- Acces A[0][0]
2 648 -- Acces a l'instruction MOVE R9,(R2)++
0 48 --- Acces B[0][1]
2 649 -- Acces a l'instruction MULT R8,R9
```

Localities for 8*8 matrix product

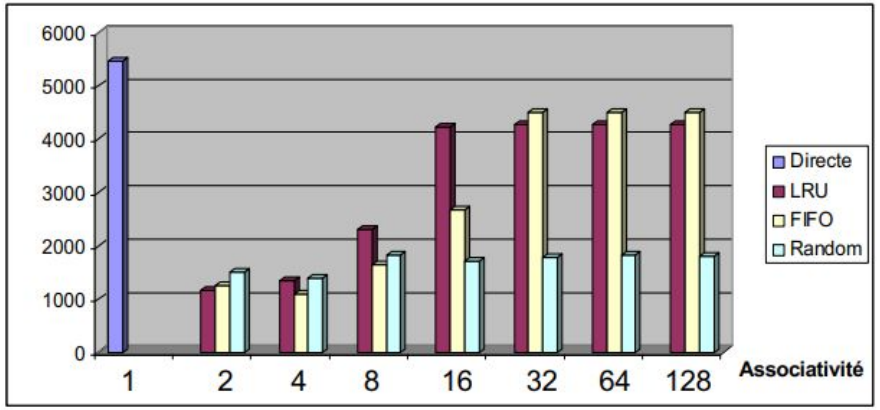


Simulation Evaluation

- Block size = 4

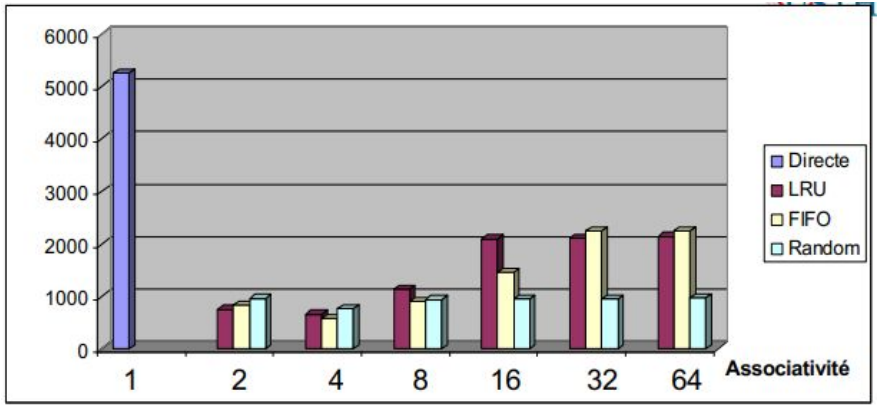


- Block size = 8

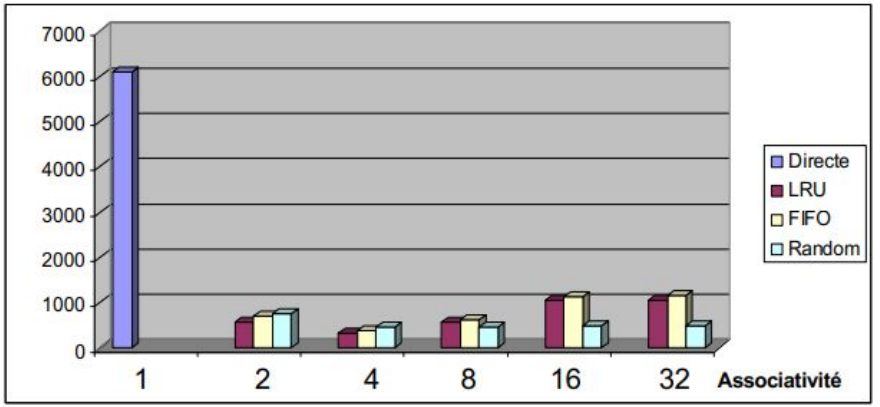


Simulation Evaluation (cont.)

- Block size = 16

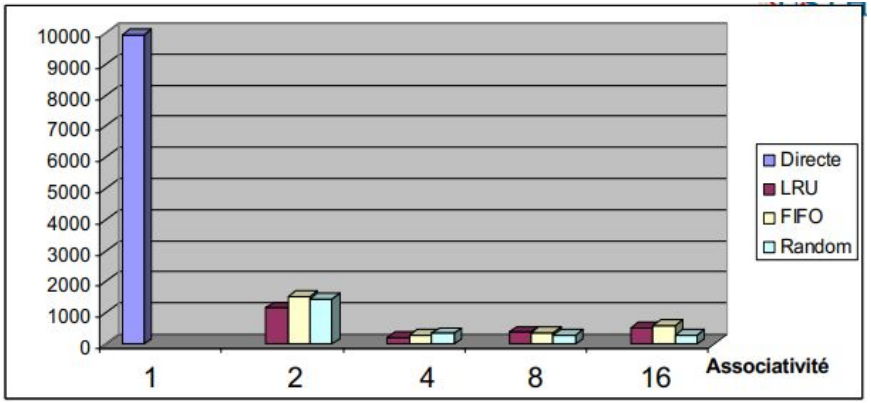


- Block size = 32

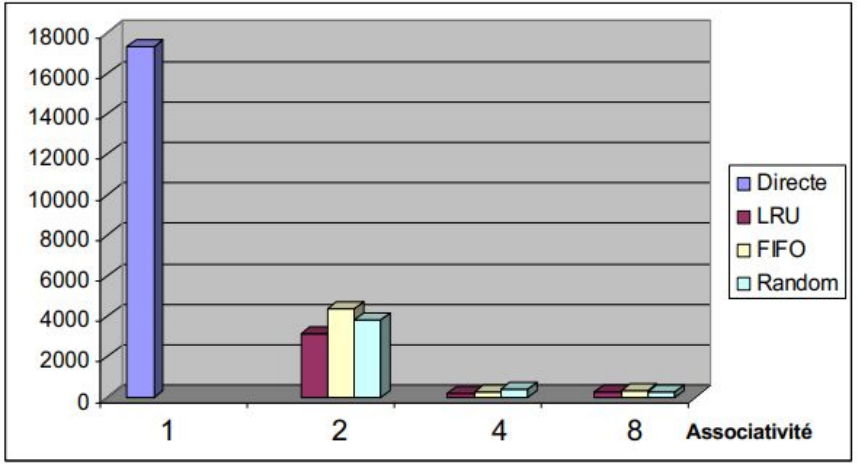


Simulation Evaluation (cont.)

- Block size = 64



- Block size = 128



Conclusions of these simulation

- A small size leads to large number of misses
  - Small size leads to more flexible cache, but large number of misses
- A large associativity does not improve the result
  - In general, the cache memory are associated with a way equal to 2, 4, and 8

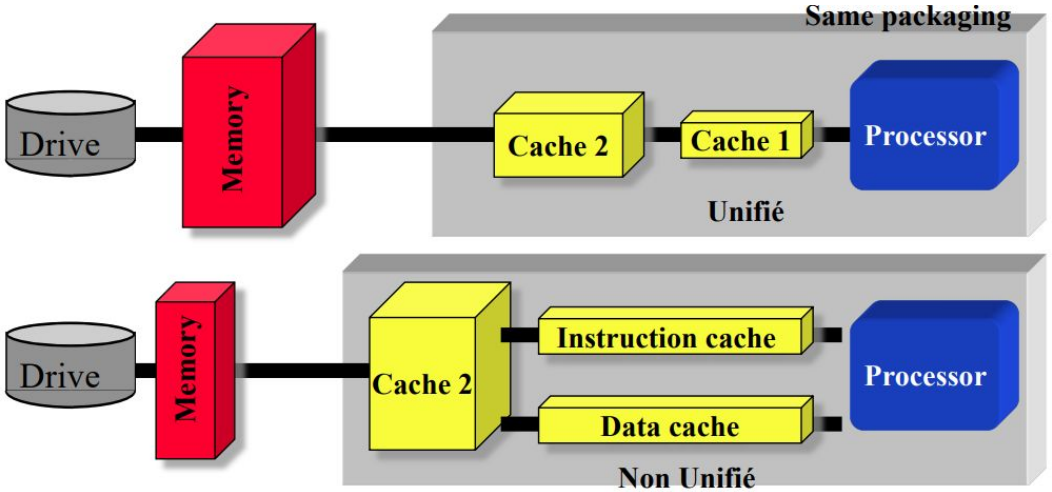
Cache Evolution

Evolutions

- The cache sizes increase
- The access times are reduced
- Caches are more and more often include in the chip
  - Enables to use access time equal to processor cycle
  - Enables to increase the bus between processor and cache
  - More instructions can be loaded at each cycle
- Number of levels increases
- The first level of cache is generally non unified
  - Instruction cache and data cache
- The second level of cache is generally unified

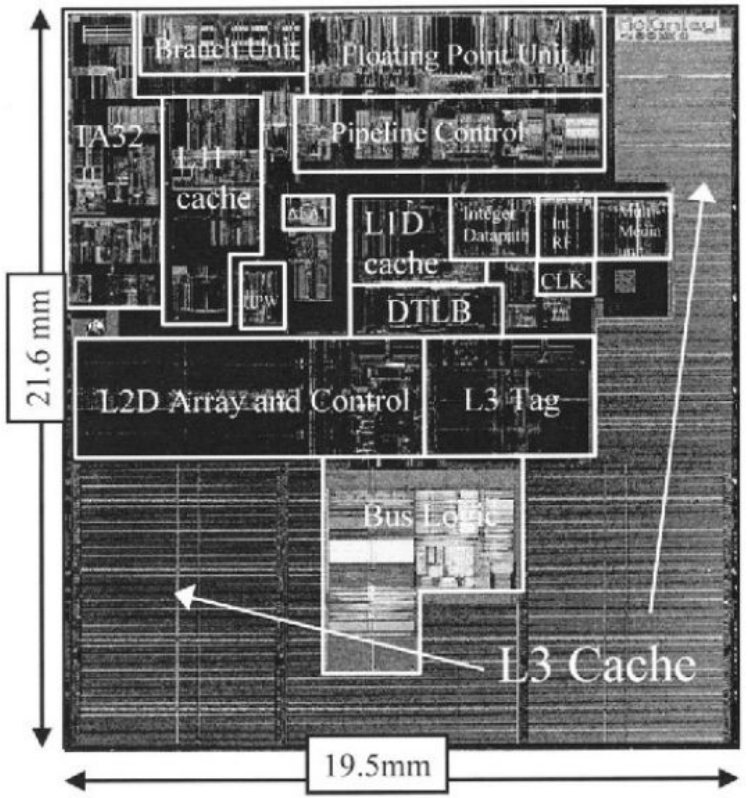
Evolutions (cont.)

- Enables to place different policies for the two different cache
- Supports no blocking cache: during a miss on data cache access, it is possible to continue with the other cache



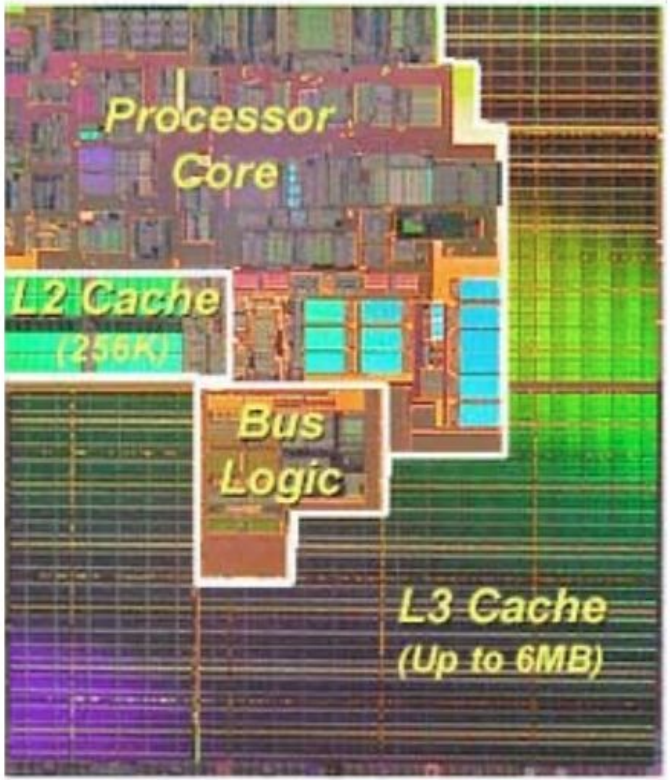
Third level of cache

- Solution developed for Itanium 2
  - 3MB for level 3



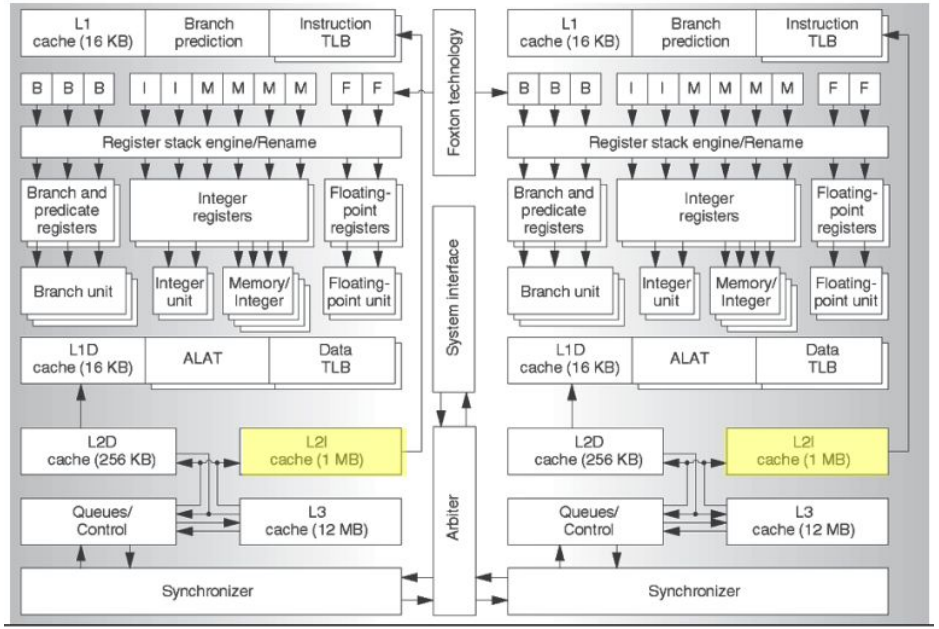
Third level of cache (cont.)

- Itanium 2, Madison core
  - 410 millions of transistors



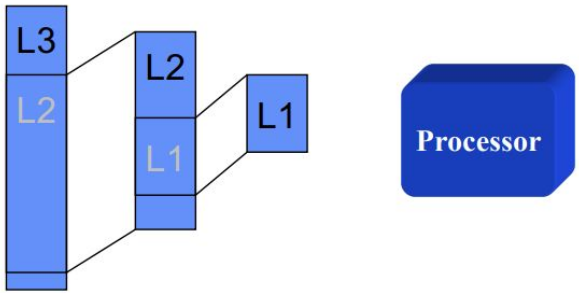
Third level of cache (cont.)

- Montecito: Dual Core Itanium
  - Non unified level 2 cache memory
  - Total cache: 27 Moctets



Inclusive and Exclusive caches

- Inclusive: L1 -> L2 -> … -> Memory
  - A block placed in L1 cache is also copied in the L2, L3, etc
- Exclusive
  - A block placed in L1 cache is not automatically stored in L2, L3, etc
  - When a block is ejected from cache Li, it is copied in cache Li + 1



Thank you for you listening