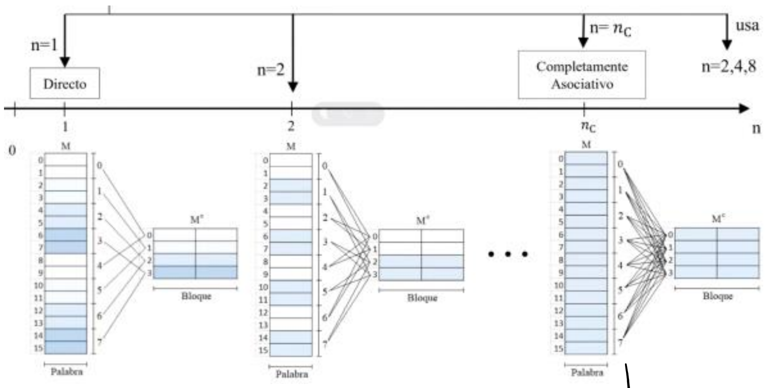
Cache Memory

martes, 29 de marzo de 2022 06:57 a. m.



this very costly (usually not all mem spaces can go anywhere in the cache)

Commonly, real caches use n = 4 or n = 8

Once the cache is full, in order to put another block in it, we have Replacement Politics:

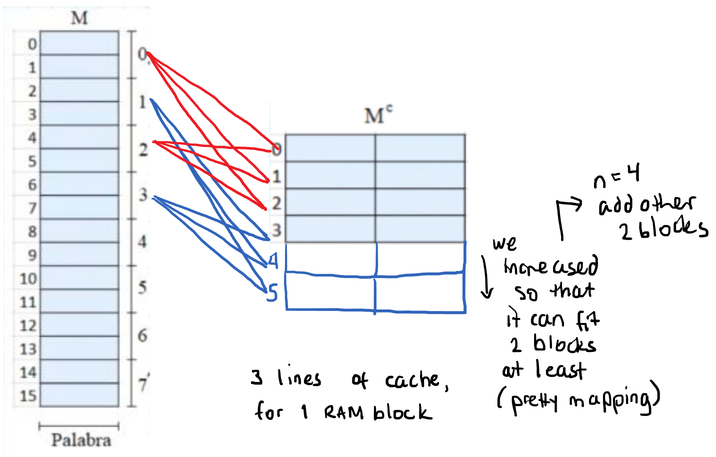
→ It defines which block is free-d to fit another
→ Depends directly on the mapping
→ These politics apply to
↳ in direct mapping, there are only two possible ways for each block.

maintain

Temporal Locality: if x is needed, likely x will be needed in the near future

→ RAM blocks are grouped in a set of cache lines
In n = 2, we have a set of two cache lines for blocks 0, 2, 4 and 6 (half the RAM)

→ In n = 3 (n are the connections)

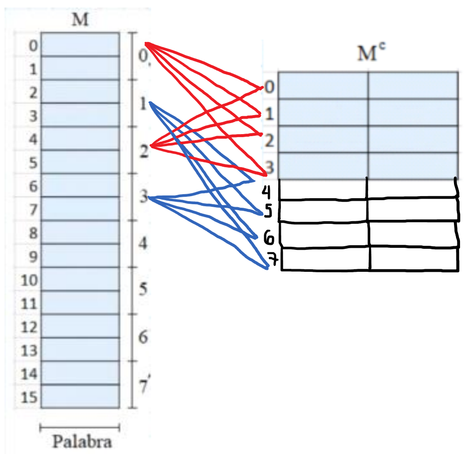


M
M^c
Palabra
n=4
add other 2 blocks
we increased so that it can fit 2 blocks at least (pretty mapping)

3 lines of cache, for 1 RAM block

→ for n = 4

If it's not completely associative, performance goes down.



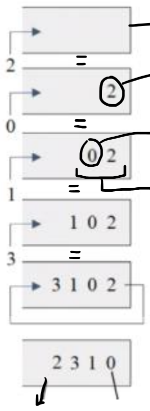
On those sets of cache, we apply the replacement politics

which cache address should be free when cache is full

simple ←

4 types:

1) FIFO: like a queue
i.e.: we have 4 lines in cache: 0, 1, 2, 3. Let's suppose completely Associative mapping.



queue state through time

→ empty cache

cache address that is ocupied

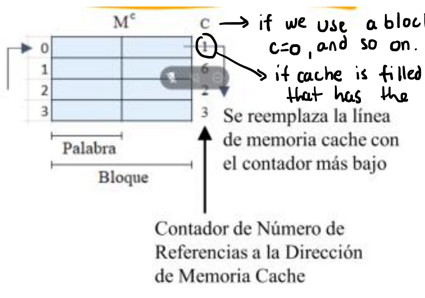
cache address is used now

the 2 would be the one out

} If we need to put another block, we free 2, and use it (goes to front)

queue of cache addresses

2) LFU: Least Frequently Used. This is the one PC's use most.


M^c
c
0
1
2
3
1
6
2
3
Palabra
Bloque
Se reemplaza la línea de memoria cache con el contador más bajo
Contador de Número de Referencias a la Dirección de Memoria Cache

→ if we use a block, the c starts in c=0, and so on. (c=6 is used 7 times)

→ if cache is filled, we free the one that has the smallest counter

↓

the biggest counter is using locality principle.

3) LRU: Least Recently Used

→ Uses a list of cache memory

Two rules for replacement

1) If the next line that the processor needs, it puts it in front. I.e., we need address 1,

3-0-1-2

2) Replacement: we replace the last on the list (2), and use it, so goes to front

1-3-0-2 ⇨ 2-1-3-0

↳ new element in address 2

4) Random: Chooses the cache line to replace randomly

↑ example

These politics are applied by set: in $n=2$, we need a politic (FIFO) for each two cache lines ($n=2$)

There are concepts to define cache performance:

→ C_H = Cache coincidence: If the processor looks for a cache address and finds it, the cache address is filled

→ C_M = Cache Miss: If the processor looks for a cache address and cannot find it, thus has to go to a RAM address and move it to cache

→ ↑ R_{CH} = Ratio of Cache coincidences: Of all times the processor looked for an adress (in cache), how many did it find it in cache (C_H)

$C_H + C_M$

$$R_{CH} = \left[\frac{C_H}{C_H + C_M}\right]$$ the closest to 1 the better $R_{CH} \in [0, 1]$

i.e. $C_H = 20$, $C_M = 30$ Therefore $R_{CH} = \frac{20}{20+30} = \frac{20}{50} = 0.4 \longrightarrow$ this cache is failing half the time suggesting problems with replacement politic.

i.e. Suppose $R_{CH} = 0.7$ and misses are $C_M = 25$, find C_H.

$$R_{CH} = \frac{C_H}{C_H + C_M}$$

$$0.7 = \left[\frac{C_H}{C_H + 25}\right]$$

$$(C_H + 25)(0.7) = C_H$$

$$0.7C_H + 17.5 = C_H$$

$$0.7C_H - C_H = -17.5$$

$$-0.3C_H = -17.5$$

$$C_H = \frac{17.5}{0.3} = \boxed{58.33}$$ This means that 58 times is found in cache, while 25 missed.

How the cache REALLY works?

1 processor: we have a concept/problem called Cache Coherence, and it cause is that it has redundancy and we should minimize redundancy, because it there is
↳ have many copies of the same data

a change in RAM, we must change it in two places or more.

↳ The cache is a redundant set of copies from RAM data.

↳ Therefore, we need to maintain coherence between RAM and cache

If the processor modifies an address in cache (M_C), we need to perform a process for consistency: cancel the previous and replace it for the new.

If we dont do this, we have a problem. There are two solutions:

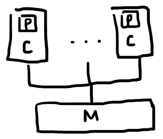
1) Write-through: the same moment the processor changes it, the processor changes the RAM as well: problem, RAM access.

2) Write-back: the processor waits: when an address in cache is modified, we mark it as dirty, and when that address is going be free and reused, on that moment we change the RAM value.

Shared Memory: P_1 changes its cache address, thus if another P had it, we need to change P and M. But we have a problem: at the moment the P_1 changed one address, it needs to send the message before change to

processors that have that address to MARK the address in all
p's as invalid.

→ If another processor wants to operate a dirty address, it asks for the correct in RAM, and thus in that moment the processor that modified it, is required to modify that address in RAM. It uses the 2) technique



Therefore we need to update data in : 1) M (RAM) : with previous two techn.
2) Caches

It uses two techniques:

1) Snooping: the message is sent to all processors using BUS (one at a time) → waiting processors

↳ this is only good for small amount of processors

=

the solution is to use shared memory, since the network can be hypercube, Ring, etc

∴ the use of parallel programming is often with distributed systems

2) Directory based