Learning Cache Replacment Policies using Register Automata

Guillem Rueda Cebollero

Uppsala Universitet

Motivations

The motivations are:

- Which block replacement policy does the system use?
- If the policy is unknown, how does it work?

Goals

The goals of the project are:

- Creating a tool for inferring the algorithm behind the block replacement policy.
- With this infomation will be possible to develop less-pessimistic WCET.

Hits / Misses

A block replacement tries to have minimum misses possible:

Hit:
When a block is inside of the cache memory.

Miss:
When a block is not in the cache memory. It means more processing time, because this block must be requested at the next memory level.

Policies

Exists different replacement block policies:

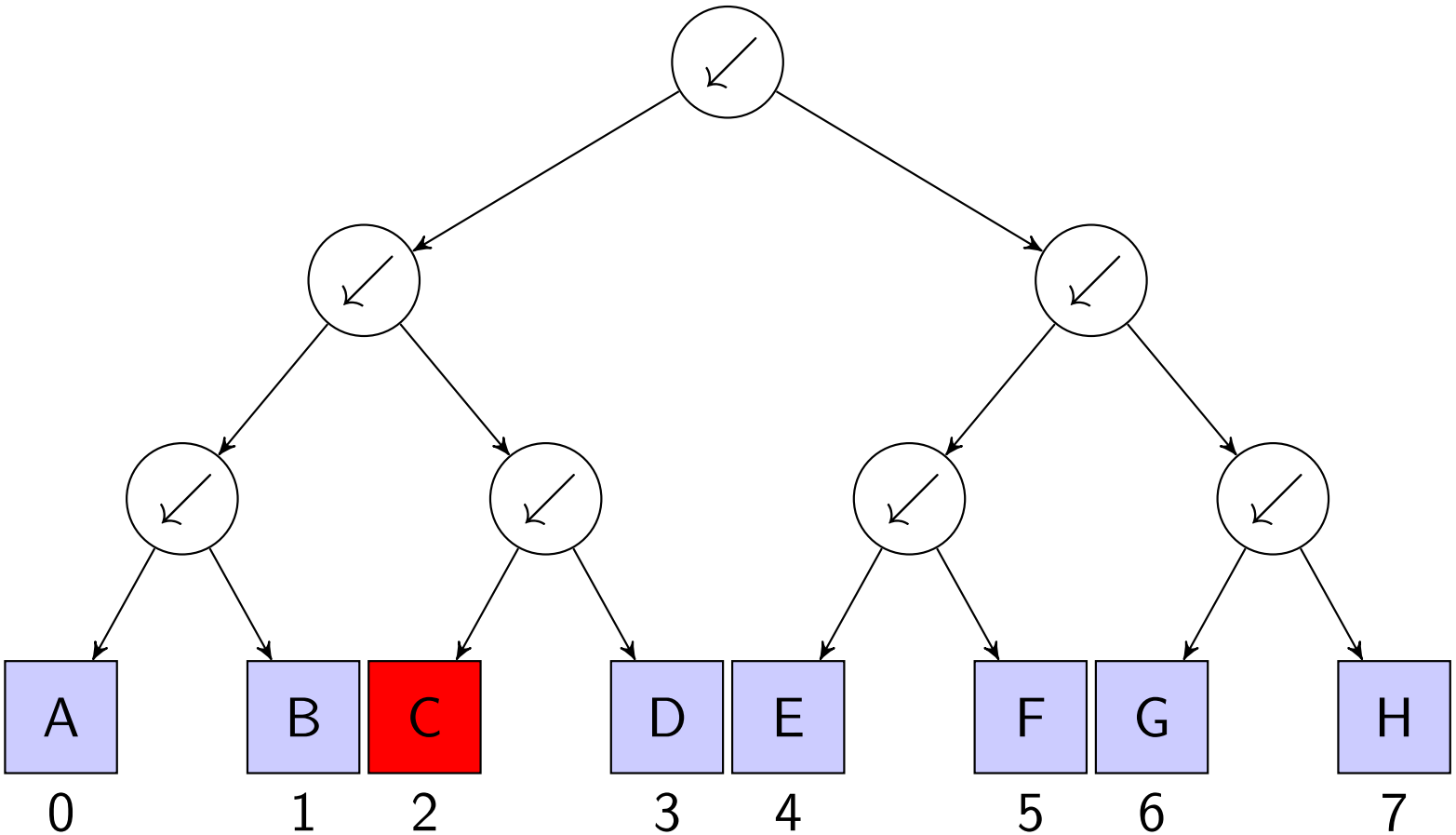
- First-in, First-out (FIFO)
- Least Recently Used (LRU)
- **Pseudo-LRU (PLRU)**
- Most Recently Used (MRU)

PLRU

- It's the most common policy in last processor generations
- Used by Intel and AMD processors
- Uses a Binary-position to taget the position to be replaced
- Cheaper than LRU

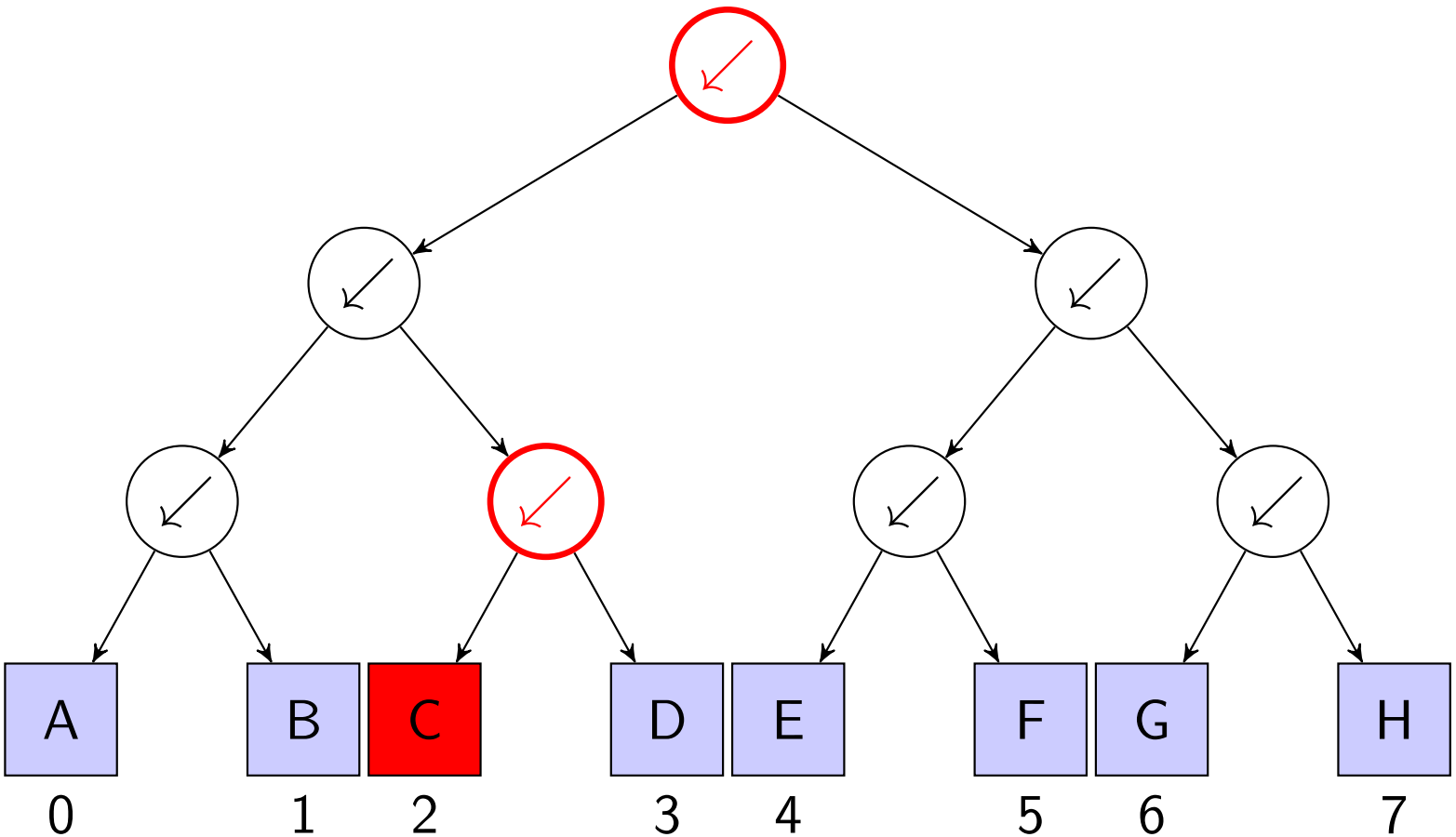
PLRU:Hit

In case of hit in the element in Position 2 (element C):



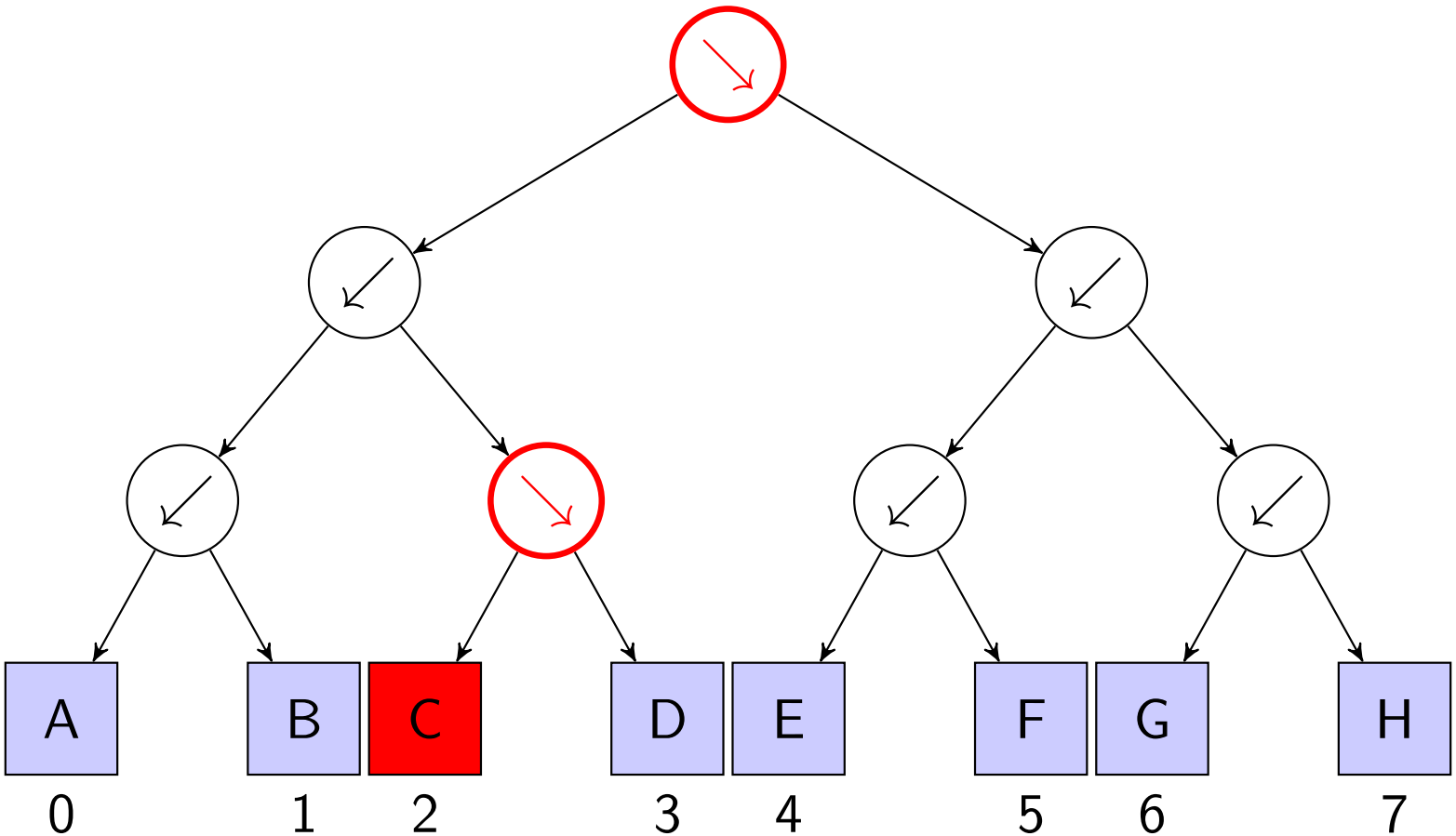
PLRU:Hit

In case of hit in the element in Position 2 (element C):



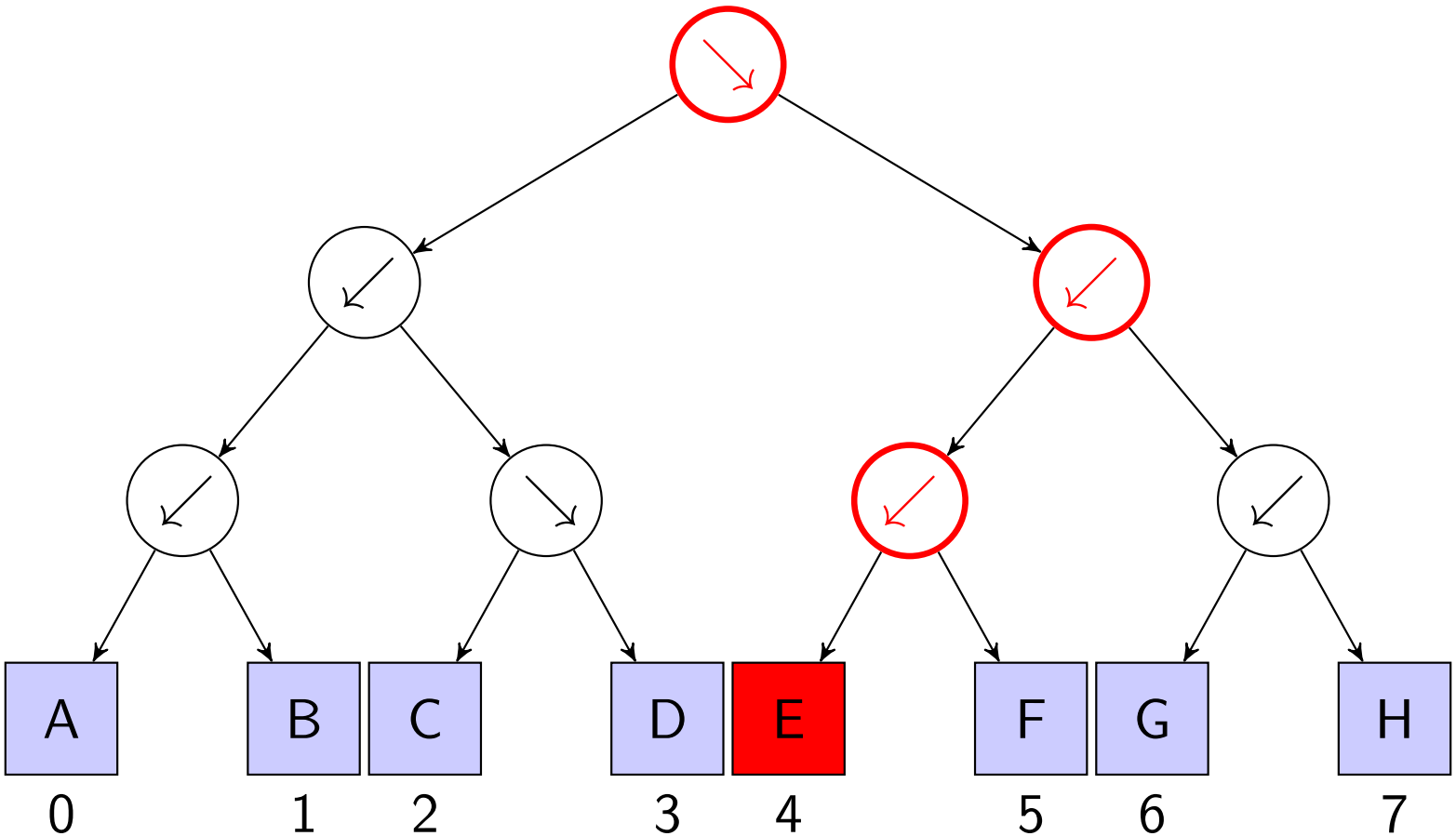
PLRU:Hit

In case of hit in the element in Position 2 (element C):



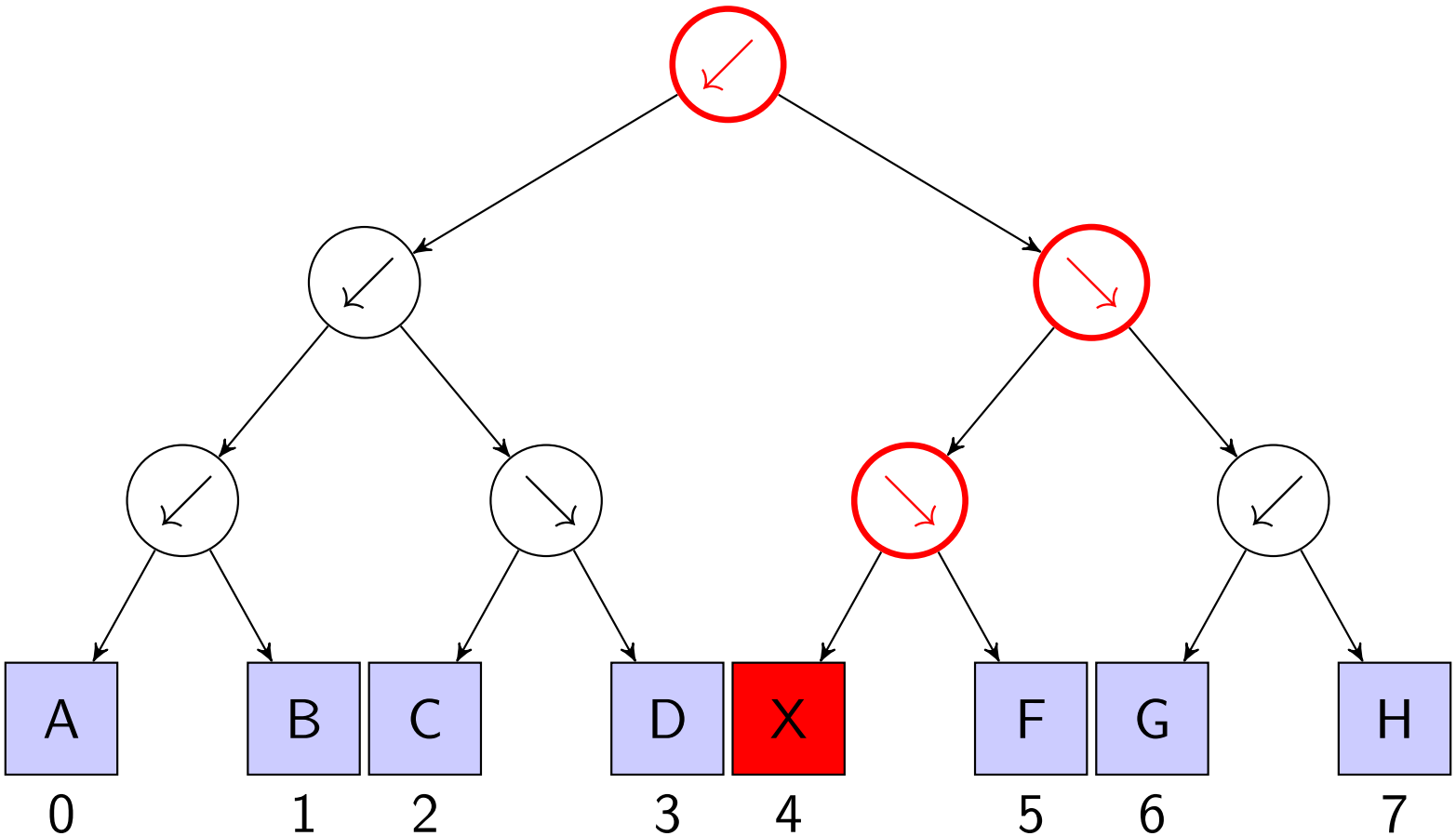
PLRU:Miss

In case of miss:



PLRU:Miss

In case of miss:



Permutation Vectors

Where will be the element on position 7 after a hit?

A	B	C	D	E	F	G	H
0	1	2	3	4	5	6	7

H	A	B	C	D	E	F	G
7	0	1	2	3	4	5	6

Permutation Vectors

It's the Fingerprint of the permutation policies. For example, LRU:

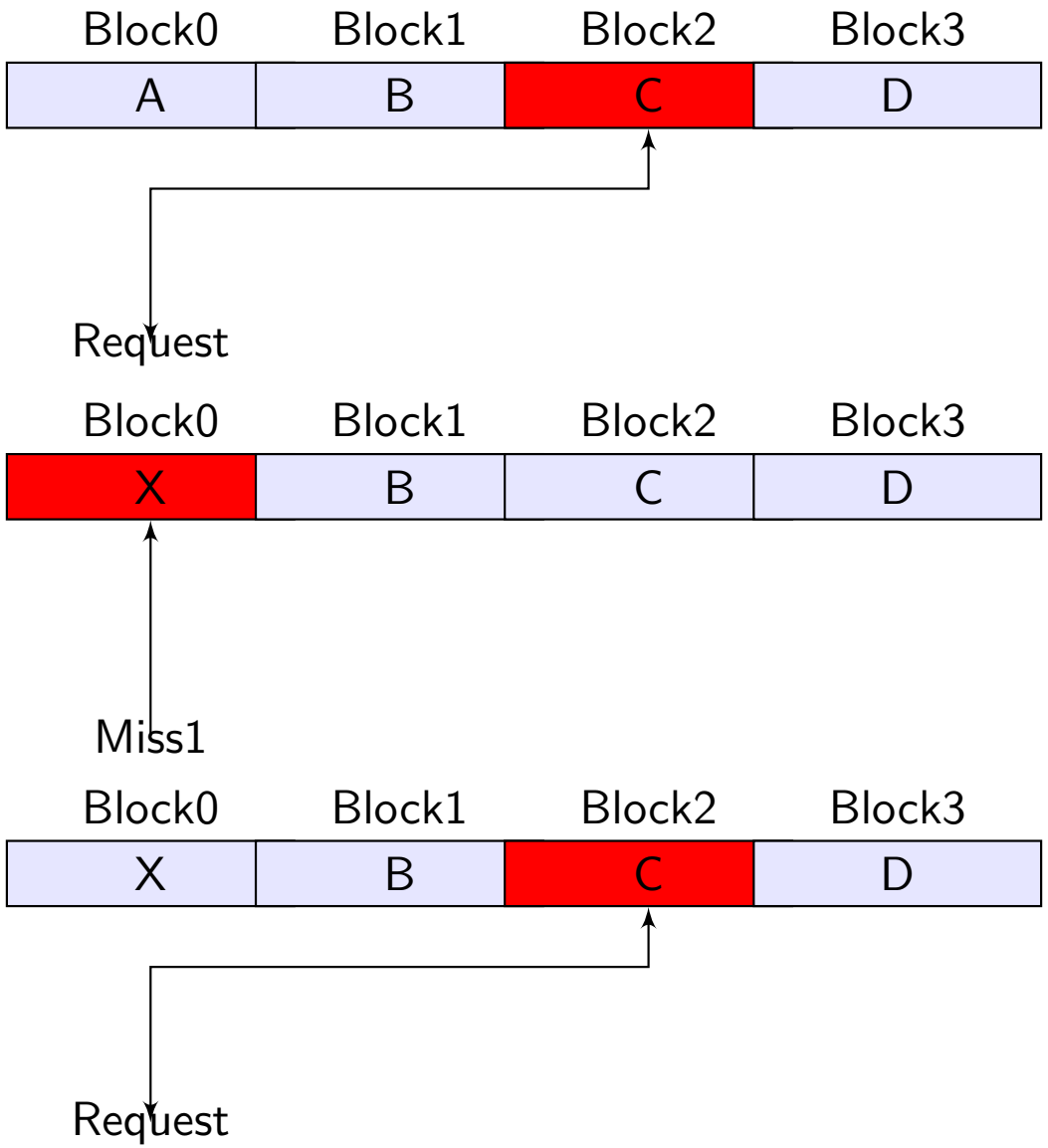
$$\prod_0^{LRU} = (0; 1; 2; 3; 4; 5; 6; 7)$$
$$\prod_1^{LRU} = (1; 0; 2; 3; 4; 5; 6; 7)$$
$$\prod_2^{LRU} = (2; 0; 1; 3; 4; 5; 6; 7)$$
$$\prod_3^{LRU} = (3; 0; 1; 2; 4; 5; 6; 7)$$
$$\prod_4^{LRU} = (4; 0; 1; 2; 3; 5; 6; 7)$$
$$\prod_5^{LRU} = (5; 0; 1; 2; 3; 4; 6; 7)$$
$$\prod_6^{LRU} = (6; 0; 1; 2; 3; 4; 5; 7)$$
$$\prod_7^{LRU} = (7; 0; 1; 2; 3; 4; 5; 6)$$

$$\prod_{miss}^{LRU} = (0; 1; 2; 3; 4; 5; 6; 7)$$

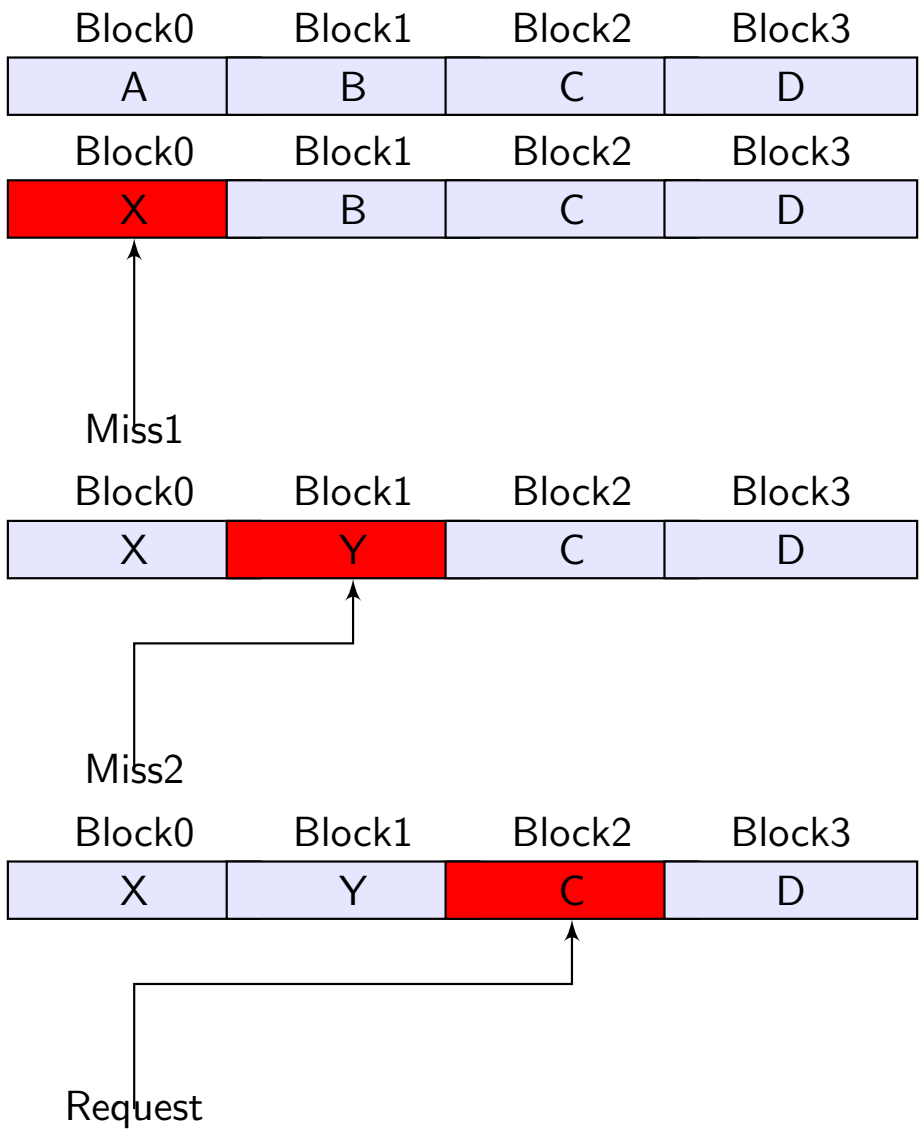
ChiPC

- Program for inferring block replacment policy
- Gets all characteristics of cache: size, number of blocks, replacment policy
- Uses Permutation Vectors (PV) to infer the replacment policy
- Developed in C/C++
- Uses the Performance Counters (PC) furnished in the CPU

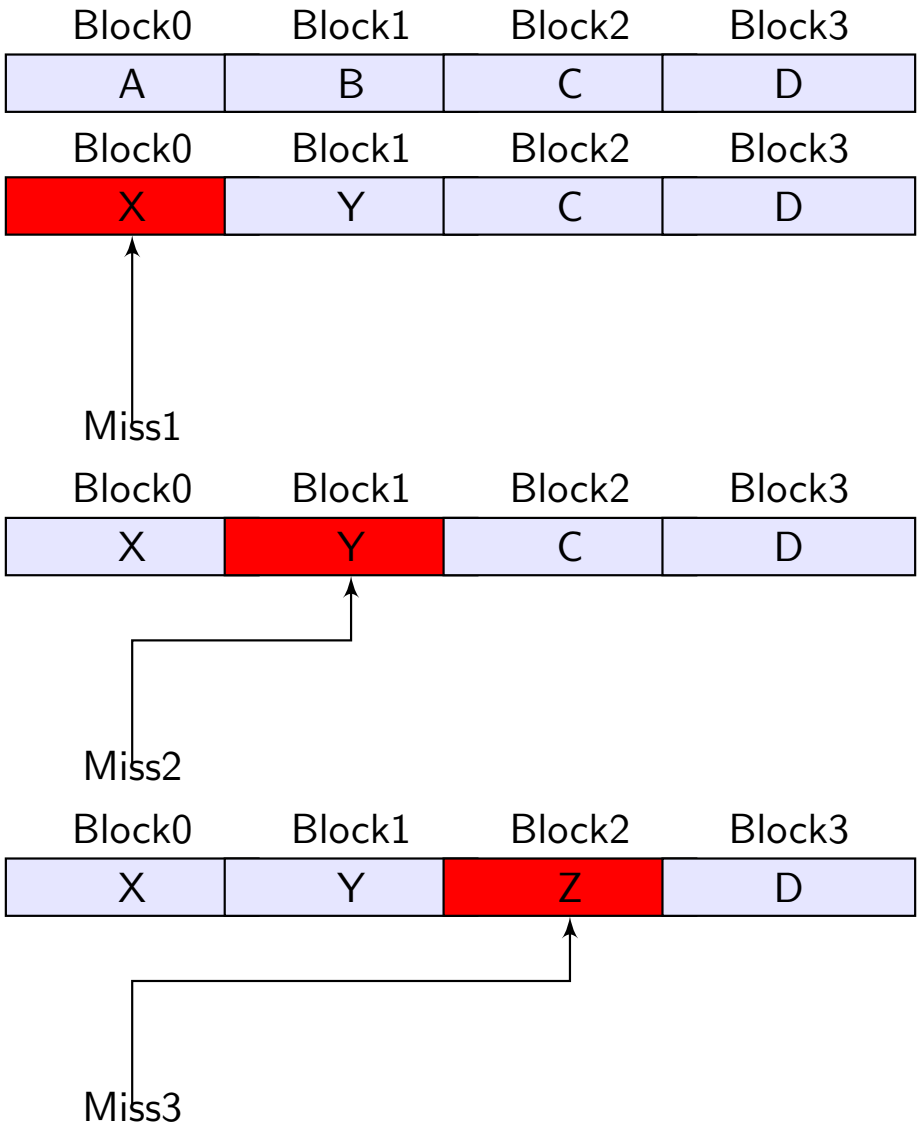
Learning PV


Block0
Block1
Block2
Block3
A
B
C
D
Request
Block0
Block1
Block2
Block3
X
B
C
D
Miss1
Block0
Block1
Block2
Block3
X
B
C
D
Request

Learning PV


Block0
Block1
Block2
Block3
A
B
C
D
Block0
Block1
Block2
Block3
X
B
C
D
Miss1
Block0
Block1
Block2
Block3
X
Y
C
D
Miss2
Block0
Block1
Block2
Block3
X
Y
C
D
Request

Learning PV


Block0
Block1
Block2
Block3
A
B
C
D
Block0
Block1
Block2
Block3
X
Y
C
D
Miss1
Block0
Block1
Block2
Block3
X
Y
C
D
Miss2
Block0
Block1
Block2
Block3
X
Y
Z
D
Miss3

Learning PV (Enhancement of the algorithm)

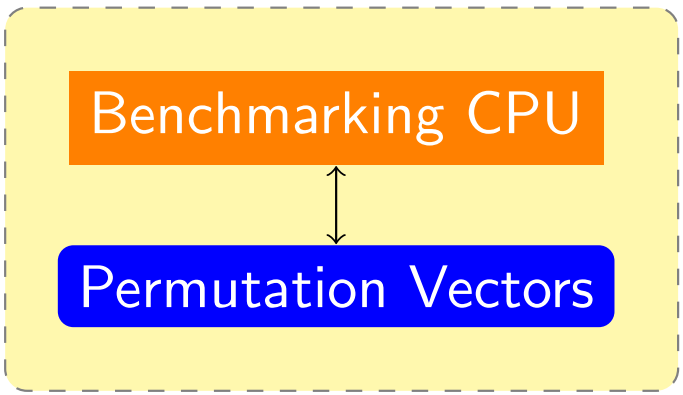
- The process is repeated N times
- The number of misses and requested blocks are alternated
- The objective is reduce the noise or effect of hardware prefetching mechanisms

Learning PV: Problems

- PV offers a limited modelling of the block replacement policies
- For example, MRU can't be modelled with PV
- This Learning process must be changed for another more general

LearnLib

ChiPC:



Project with LearnLib: