Multi-Processor Systems

Pirouz Bazargan Sabet

Sorbonne Université - LIP6

Pirouz.Bazargan-Sabet@lip6.fr

Introduction

- A Risc can execute on instruction at a cycle
- This rhythm imposes a high exigence on the memory sub-system

Outline

- Memory subsystem
- Caches
- Coherency

Introduction

What are these requirements ?

- Work at the processor's frequency
- Answer to an instruction fetch request and potentially to a data read or write request in one cycle
- Contain the processor's memory space : 4GB per process
- Reasonable cost

Introduction

Is there any memory technology with such characteristics ?

- CMOS Static Memories
  - Work at processor's frequency
  - One cycle per access
  - KBs to few MBs

Introduction

Is there any memory technology with such characteristics ?

- Hard Disks
  - Require several million cycles per access
  - GBs to few TBs

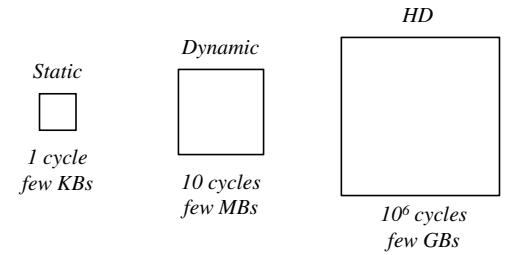
Introduction

Is there any memory technology with such characteristics ?

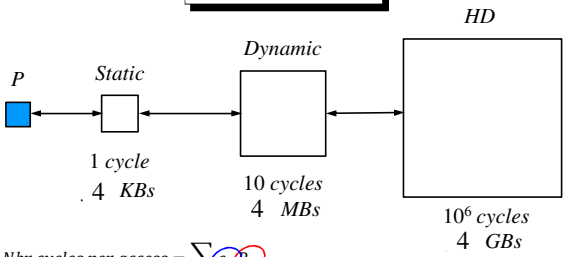
- CMOS Dynamic Memories
  - Work at processor's frequency
  - Require several cycles per access
  - MBs to few GBs

Introduction

Summary



Introduction



$$Mean\ Nbr\ cycles\ per\ access = \sum_i c_i P_i$$

$$= 1 \times \frac{4K}{4G} + 10 \times \frac{4M - 4K}{4G} + 10^6 \times \frac{4G - 4M}{4G}$$

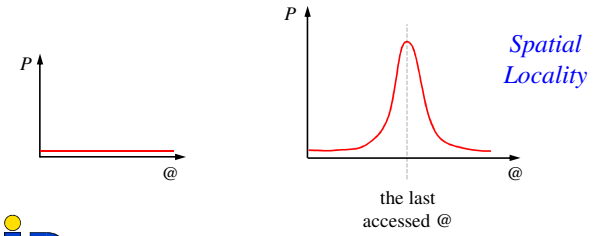
$$= 1 \times 10^{-6} + 10 \times (10^{-3} - 10^{-6}) + 10^6 \times (1 - 10^{-3})$$

$$= 999\,000\ cycles$$

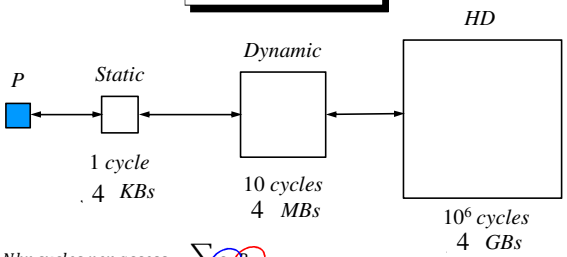
Introduction

Predict the @ that the processor will access ?

- The previous calculation was based on the assumption that all the @ have the same probability to be accessed



Introduction



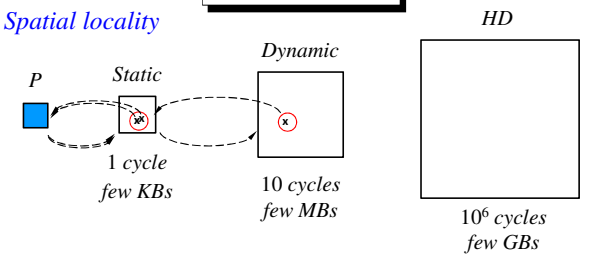
$$Mean\ Nbr\ cycles\ per\ access = \sum_i c_i P_i$$

To improve the probability of the data requested by the processor to be present in fast memories, it is necessary to ...

Anticipate
=
Predict

Introduction

Spatial locality



The probability for a requested data to be present in a memory level : ***Hit Rate***

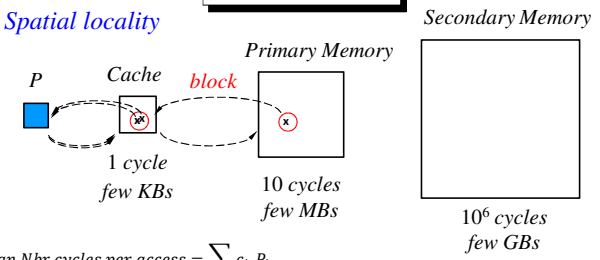
$HitRate_{instruction}$ = 95% à 98%

$HitRate_{data}$ = 90% à 95%

$$\boldsymbol{MissRate = 1 - HitRate}$$

Cache

Spatial locality

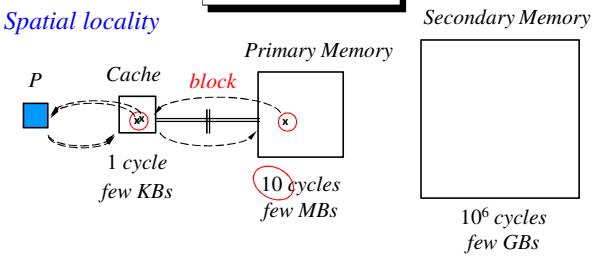


$$\text{Mean Nbr cycles per access} = \sum_i c_i \, P_i$$
$$= 1 \times 0.9 \quad + 10 \times (1 - 0.9)$$
$$= 1.9 \text{ cycles}$$

block = few dozen bytes
32, 64, 128

Cache

Spatial locality



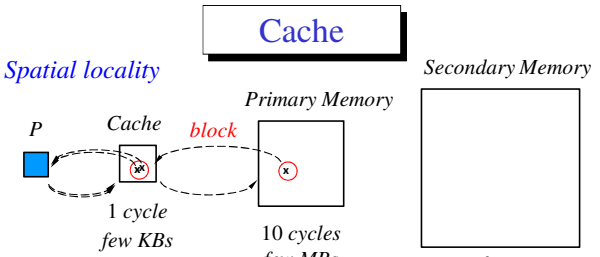
$$\#\text{cycles to access the PM} = \#\text{cycles to initiate the access} + \#\text{cycles to find the block} + \#\text{cycles to transfer the block}$$

do not depend on the block's size

depends on the block's size

Cache

Spatial locality

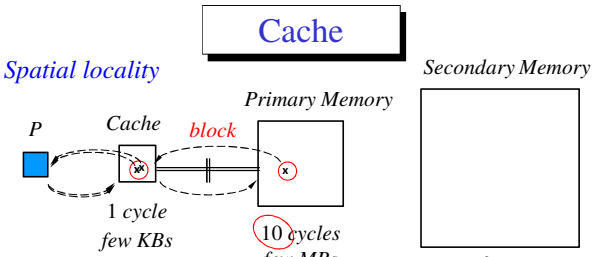


$$\text{Mean Nbr cycles per access} = \sum_i c_i \, P_i$$
$$= 1 \times 0.9 \quad + 10 \times (1 - 0.9)$$
$$= 1.9 \text{ cycles}$$
$$= 1 \times 0.9 \quad + 10 \times (1 - 0.9) \times 8$$
$$= 8.9 \text{ cycles}$$

block = 32 Bytes

Cache

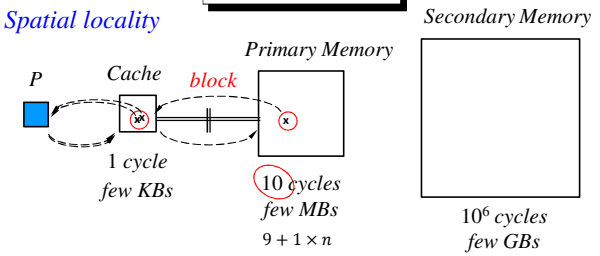
Spatial locality



$$\#\text{cycles to access the PM} = \#\text{cycles to initiate the access} + \#\text{cycles to find the block} + \#\text{cycles to transfer the block}$$
$$= A + B \times n$$
$$= A + B \times \frac{\#of\ Bytes}{Bus\ size}$$

Cache

Spatial locality



$$\text{Mean Nbr cycles per access} = \sum_i c_i \, P_i$$

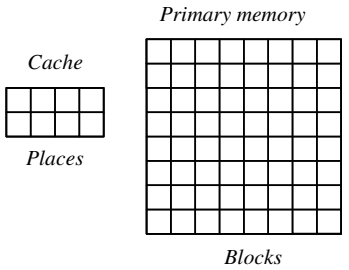
$$= 1 \times 0.9 \quad + (9 + 1 \times 8) \times (1 - 0.9)$$

$$= 2.6 \text{ cycles}$$

block = 32 *bytes*

Cache

Cache organization



How the cache place is selected to save a PM's block ?

- *Direct Mapping* cache
- *Set Associative* cache
- *Full Associative* cache

Cache

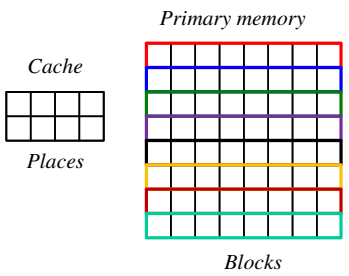
Summary

- A hierarchical organization of the memory is used to cover the addressing space
- The *Spatial Locality* is exploited to increase the *Hit Rate*
- The *Primary Memory* is organized in order to reduce the access delay in case of *Miss* : organized as a set of *Blocks*
- The *System Bus* is organized to optimize the *transfer* of blocks (*burst*)

⇨ *Reduction Mean nbr of cycles per access*

Cache

Cache organization :
Direct Mapping



Partition of blocks in *Sets* (#Set = #Place)

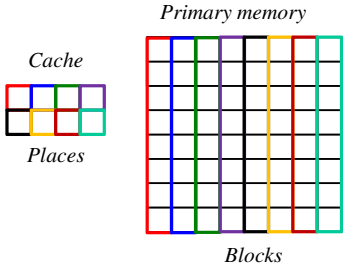
A great number of partitions are possible

The best partition takes the best advantage of *Spatial locality*

meaning :
Maximizes the minimal distance between 2 blocks

Cache organization :
Direct Mapping

Cache



Partition of blocks in *Sets*
(#Set = #Place)

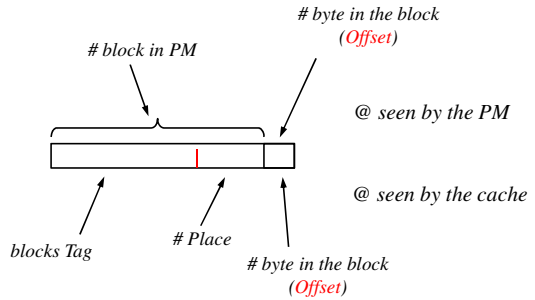
The best partition takes the best advantage of *Spatial locality*

meaning :
Maximizes the minimal distance between 2 blocks

The LSbs of the block number gives the place number

Cache organization :
Direct Mapping

Cache



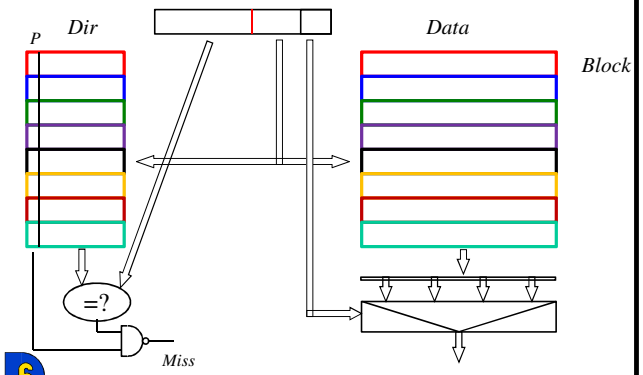
Cache organization :
Direct Mapping

Cache

- The least significant bits of the block number define the set thus the cache place number
- At each time a single element of a set (a single block) may be present in the cache
- The identifier of the set element present in the cache should be saved
- The most significant bits of the block number give the blocks identifier in the set
- Thus, the cache is composed of 2 pieces of memory :
  - The *Data* memory saves the data contained in a block
  - The *Directory* memory saves the block identifier in the set

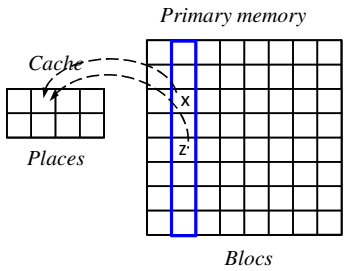
Cache organization :
Direct Mapping

Cache



Cache organization

Cache

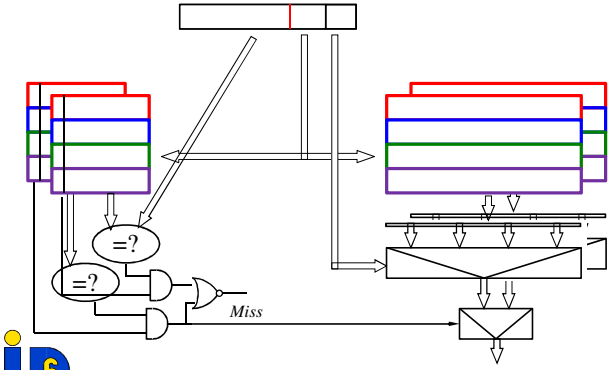


How the cache place is selected to save a PM's block ?

- *Direct Mapping* cache
- *Set Associative* cache

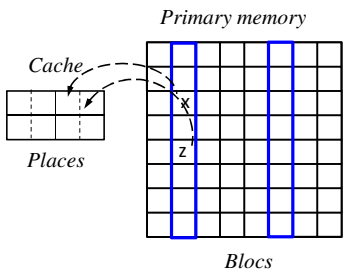
Cache organization : *Set Associatif*

Cache



Cache organization

Cache

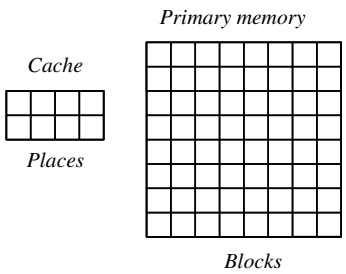


How the cache place is selected to save a PM's block ?

- *Direct Mapping* cache
- *Set Associative* cache

Cache organization

Cache

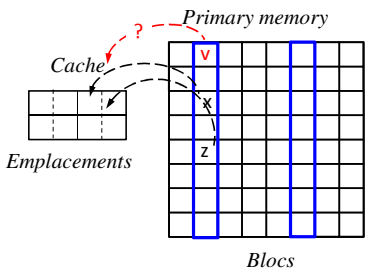


How the cache place is selected to save a PM's block ?

- *Direct Mapping* cache
- *Set Associative* cache
- *Full Associative* cache

Cache

Cache organization :
Set Associatif



How the cache place is selected to save a PM's block ?

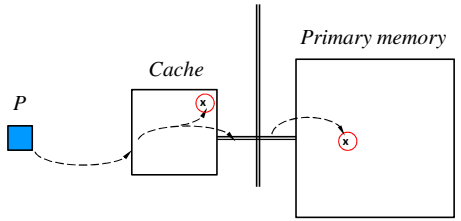
- *Set Associative* cache

Replacement strategy : *Pseudo LRU*

Cache

Write Strategy

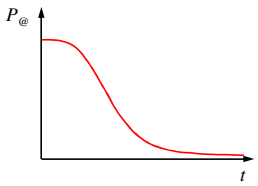
- *Write Through*



Cache

Predict the @ that the processor will access ?

- Replacement strategy : *LRU*



Temporal Locality

Cache

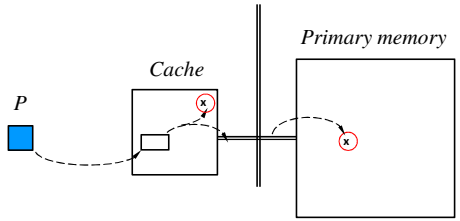
Write Strategy

- *Write Through*
  - Data are written in the primary memory *through* the cache
  - The coherency in the memory hierarchy is fulfilled by construction
  - Each single write requires a memory access
  - Processor's stall cycles may be avoided by introducing a *Write Buffer*
  - The *Write Buffer* contains a small number of places. Stall cycles may be necessary if several writes are received close in time and the *Buffer* goes to be full

Cache

Write Strategy

- *Write Through*



Cache

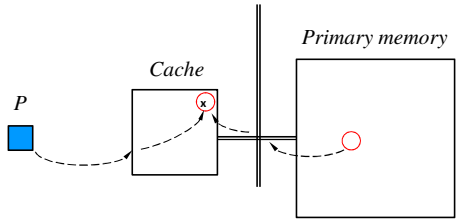
Write Strategy

- *Write Back*
  - The data are only saved in the cache
  - The PM is updated when a modified block is evinced from the cache
  - Difficulties with coherency in the memory hierarchy
  - Do not require a memory access
  - An additional mechanism is required to insure the coherency
  - Updating the PM requires 2 block transfers on the bus
  - Unnecessary memory updates may be avoided by identifying modified blocks (*Dirty*)

Cache

Write Strategy

- *Write Back*



Cache

Memory coherency in a multi-processor system

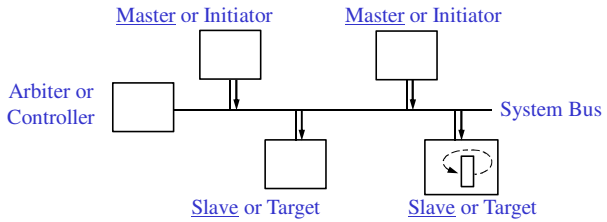
- If a processor P_i writes a value V_0 at an address A_0 **then**, reads the address A_0 , it should obtain V_0
- If a processor P_i writes a value V_0 at an address A_0 and if another processor P_j $(j \neq i)$ reads continuously the address A_0 , it should eventually obtain V_0 after an undefined number of attempts
- If a processor P_i writes a value V_0 at an address A_0 then, then value V_1 at the same address and if another processor P_j $(j \neq i)$ reads continuously the address A_0, it should never see V_1 then, V_0

Cache

How to insure the memory coherency ?

- No cache !!

  Some parts of the memory space may be defined as *uncachable.*



Pi – Bus

Name	*Emitter*	*Receiver*	*N bits*	*Signification*
Ck	*Ext*	*CMS*	`1`	*Clock* : Synchronization signal
ResetN	*Ext*	*CMS*	`1`	*Reset* : Reset signal – negative logic
Req	*M*	*C*	`1*M`	*Request* : Request to use the bus
Gnt	*C*	*M*	`1*M`	*Grant* : Permission to use the bus
A	*M*	*CS*	`30`	*Address*
Opc	*M*	*S*	`4`	*Opcode* : Byte selection
Sel	*C*	*S*	`1*S`	*Select* : Selection of the slave
D	*MS*	*MS*	`32`	*Data*
Read	*M*	*S*	`1`	*Read* : Operation type – read or write
Ack	*S*	*MC*	`3`	*Acknowledge* : Slave's acknowledgment
Lock	*M*	*C*	`1`	*Lock* : Still need the bus
Tout	*C*	*MS*	`1`	*TimeOut* : Too long ! – Transaction aborted

Cache

How to insure the memory coherency ?

- No cache !!

  Some parts of the memory space may be defined as *uncachable.*

  - *Uncachable* addresses are known by the cache

or

  - An *uncachable* address is notified by the slave during a read

Pi – Bus

Ack	*S*	*MC*	`3`	*Acknowledge* : Slave's acknowledgement

RDY	`011`	*Ready*
WAT	`000`	*Wait*
RTR	`100`	*Retract*
UNC		*Uncachable*

Cache

How to insure the memory coherency ?

- No cache !!

  Some parts of the memory space may be defined as *uncachable.*

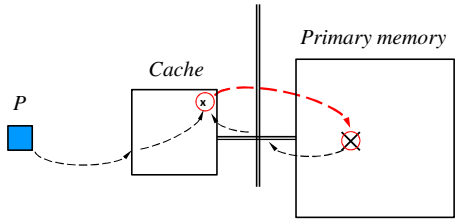
- *Write Through* caches guarantee by construction the memory coherency ... in a mono-processor system

  and ease the implementation of the coherency in a multi-processor system

Cache

Write Strategy

- *Write Back*



A *Directory* is required for the PM

Cache

How to insure the memory coherency ?

- No cache !!

  Some parts of the memory space may be defined as *uncachable.*

- *Write Through* caches ease the implementation of memory coherency

- In a *Write Back* cache, when a block is modified the primary memory should be notified that his block is no more up-to-date ... during the first modification
  From now on, the block is tagged as *Invalid* in the PM

Pi – Bus

Opc	*M*	*S*	*4*	*Opcode* : Byte selection

WDU	0010	*Word Undefined*
HW0	1000	*Half Word 0*
HW1	1010	*Half Word 1*
BY0	1100	*Byte 0*
BY1	1101	*Byte 1*
BY2	1110	*Byte 2*
BY3	1111	*Byte 3*
NOP	0000	*No Operation*
INV		*Invalidate*

Cache

How to insure the memory coherency ?

- No cache !!

  Some parts of the memory space may be defined as *uncachable.*
- *Write Through* caches ease the implementation of memory coherency
- In a *Write Back* cache, when a block is modified the primary memory should be notified that his block is no more up-to-date ... during the first modification
  From now on, the block is tagged as *Invalid* in the PM

An additional mechanism should be implemented in the cache to *monitor* the transfers on the bus. *Snoopy*

Cache

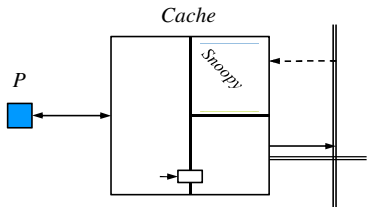
Snoopy – WT Cache

- The *Snoopy* monitors the *transfers* on the bus
  - *Write* transfers
    - Check if the address of the transfer matches a block present in the cache (check the *Directory* – processor's request should wait – be blocked)
    - Extract the block (read the *Data* – processor's request should wait – be blocked)
    - Modify the block with the data received from the bus
    - Save the modified block in the cache (write into *Data* – processor's request should wait – be blocked)

requires 3 or 4 cycles – *Write* transfer 1 cycle

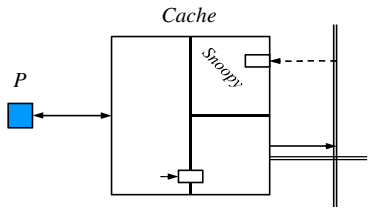
Cache

Cache organization



Cache

Cache organization



A *Fifo* can be inserted to save a series (small number) of write transfers close in the time

Cache

Snoopy – WT Cache

- The *Snoopy* monitors the *transfers* on the bus
  - *Write* transfers

    What if the *Fifo* is near to overflow ?

    - Check if the address of the transfer matches a block present in the cache (check the *Directory* – processor's request should wait – be blocked)
    - Invalidate the block if present (update the *Directory* – processor's requests should wait – be blocked)

requires 2 cycles – *Write* transfer 1 cycle

Cache

Snoopy – WT Cache

- The *Snoopy* monitors the *transfers* on the bus
  - *Invalidate* transfers

    - Check if the address of the transfer matches a block present in the cache (check the *Directory* – processor's request should wait – be blocked)
    - Invalidate the block if present (update the *Directory* – processor's requests should wait – be blocked)

requires 2 cycles – *Invalidate* transfer 1 cycle

Cache

Snoopy – WT Cache

- The *Snoopy* monitors the *transfers* on the bus
  - *Write* transfers

    What if the *Fifo* is still near to overflow ?

    - Check if the address of the transfer matches a block present in the cache (check the *Directory* – processor's request should wait – be blocked)
    - Invalidate the cache (update the *Directory* – processor's requests should wait – be blocked)

Cache

Snoopy – WB Cache

- The *Snoopy* monitors the *transfers* on the bus
  - *Write* transfers
    - Check if the address of the transfer matches a block present in the cache (check the *Directory* – processor's request should wait – be blocked)
    - Extract the block (read the *Data* – processor's request should wait – be blocked)
    - Modify the block with the data received from the bus
    - Save the modified block in the cache (write into *Data* – processor's request should wait – be blocked)

requires 3 or 4 cycles – *Write* transfer 1 cycle

Cache

Snoopy – WB Cache

- The *Snoopy* monitors the *transfers* on the bus
  - *Write* transfers
    - Check if the address of the transfer matches a block present in the cache (check the *Directory* – processor's request should wait – be blocked)
    - Invalidate the block if present (update the *Directory* – processor's requests should wait – be blocked)

requires 2 cycles – *Write* transfer 1 cycle

Cache

Snoopy – WB Cache

- The *Snoopy* monitors the *transfers* on the bus
  - *Invalidate* transfers
    - Check if the address of the transfer matches a block present in the cache (check the *Directory* – processor's request should wait – be blocked)
    - Invalidate the block if present (update the *Directory* – processor's requests should wait – be blocked)

requires 2 cycles – *Invalidate* transfer 1 cycle

Cache

Snoopy – WB Cache

- The *Snoopy* monitors the *transfers* on the bus
  - *Write* transfers

    What if the *Fifo* comes to is near to overflow ?
    - Check if the address of the transfer matches a block present in the cache (check the *Directory* – processor's request should wait – be blocked)
    - Invalidate the cache (update the *Directory* – processor's requests should wait – be blocked)

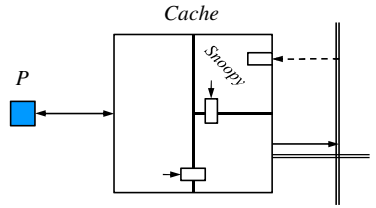
Cache

Snoopy – WB Cache

- The *Snoopy* monitors the *transfers* on the bus
  - Aborted transactions (*Retract*)
    - Check if the address of the transfer matches a block present in the cache (check the *Directory* – processor's request should wait – be blocked)
    - Request send to *Bus Interface Controller* to update the memory (*Update Transaction*)

Cache

Cache organization : *WB*



Pi – Bus : ***Retract***