CE4302 – Arquitectura de Computadores II

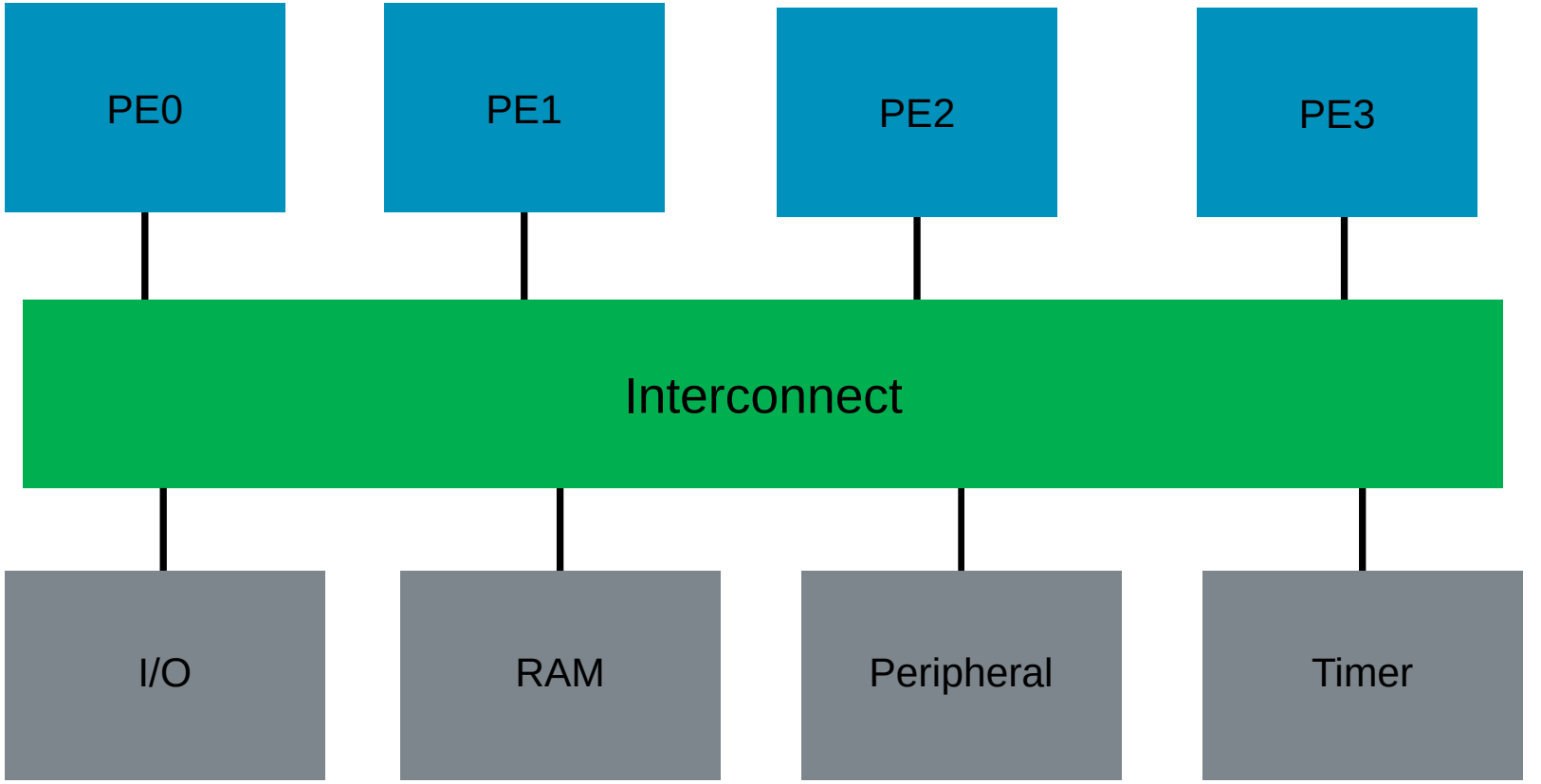
Coherencia Caché

PROFESOR: ING. LUIS BARBOZA ARTAVIA

Agenda

- Comunicación multiprocesadores
- Conceptos básicos caché
- Coherencia caché
- Consistencia de memoria
- Referencias

Comunicación multiprocesadores



Ejemplo de una arquitectura multiprocesador

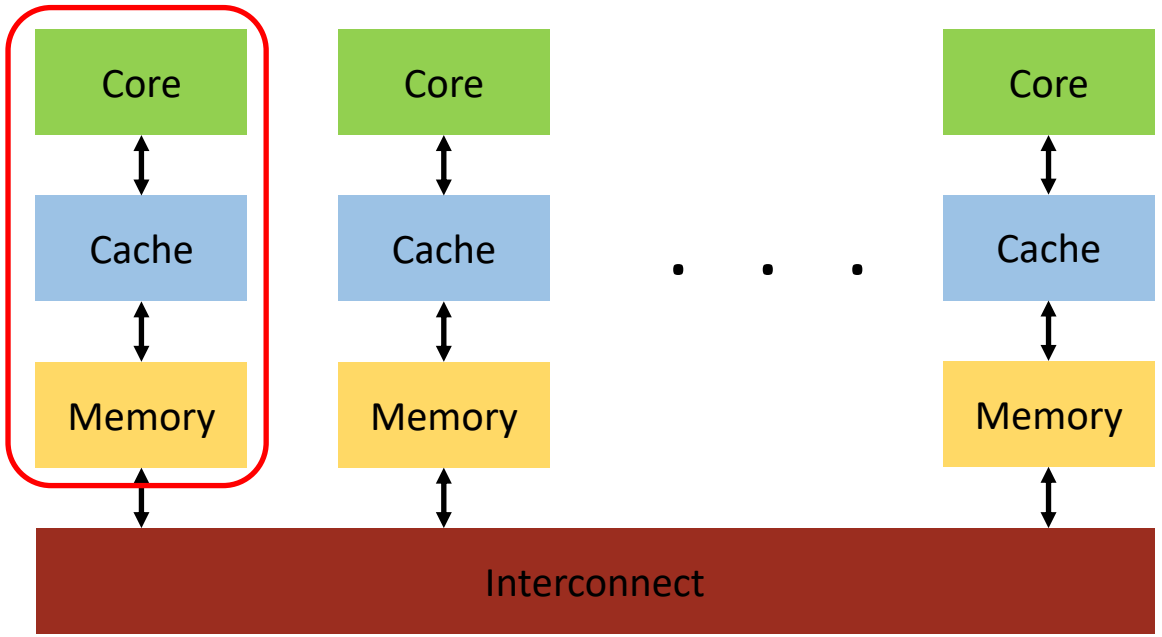
Comunicación multiprocesadores

- **Paso de Mensajes**
- Las aplicaciones “envuelven” los datos que necesitan compartir en mensajes.
- Comunicación explícita con mensajes send() & receive()
- Sincronización es implícita mediante bloques de mensajes

- **Memoria compartida**
- Todos los PEs pueden leer y escribir datos en un espacio de memoria compartido
- Comunicación es implícita mediante acceso a memoria
- Sincronización se realiza mediante uso de operaciones atómicas de memoria (atomic memory operations)

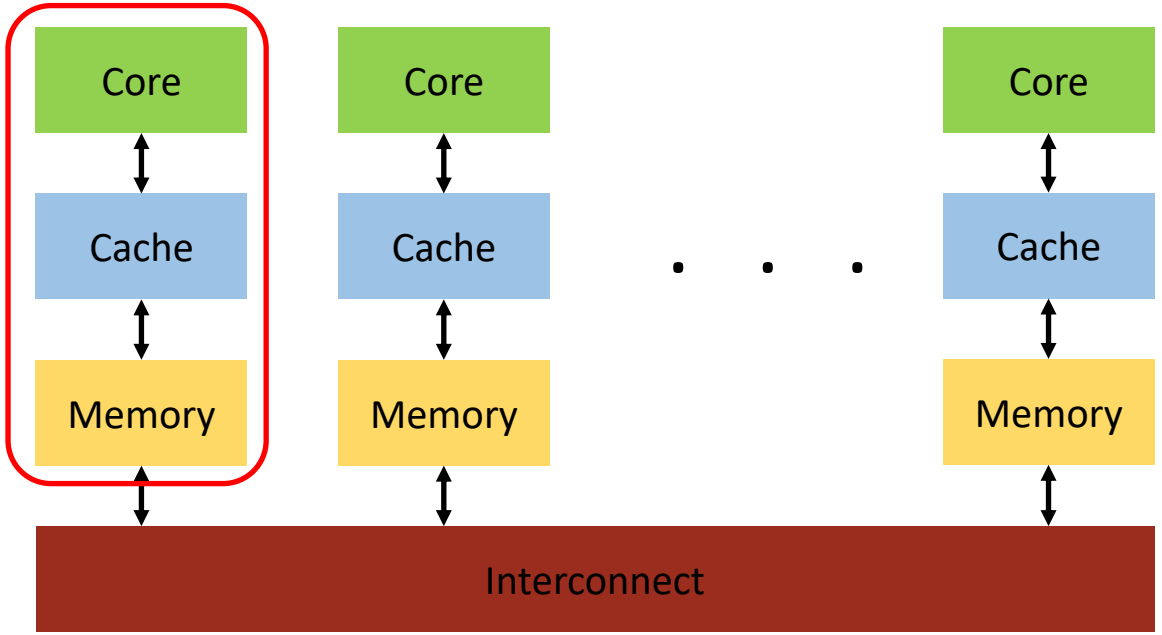
Paso de mensajes

- Cores/PEs no dependen de espacio de memoria compartida.
- Cada PE tiene sus datos, I/O, Memoria
- Operaciones explícitas de I/O para comunicación
- Sincronización con mensajes



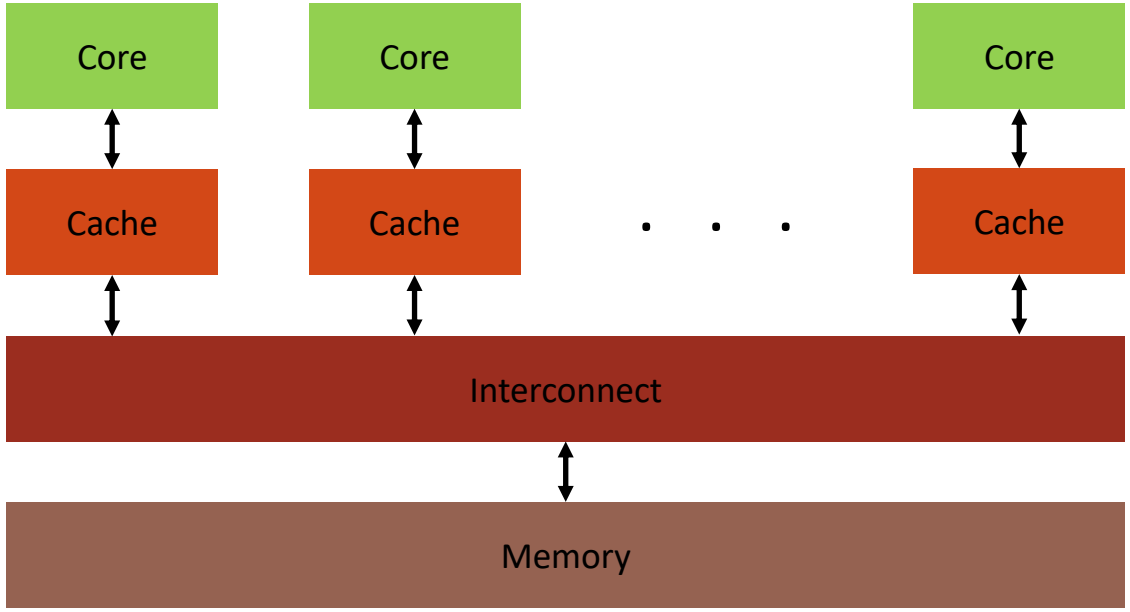
Paso de mensajes

- Ventajas
  - Menos HW -> “Facilidad” de diseño
  - Se enfoca en operaciones costosas no locales
- Desventajas
  - ¿Implementación de SW? ¿Por qué?



Memoria compartida

- PEs observan un único espacio de direccionamiento de memoria
- PEs pueden tener caché que almacena datos.
  - Comunicación se da mediante MEM R/W
  - Operaciones atómicas -> Sincronización
    - ISA contiene instrucciones específicas
    - HW garantiza operación correcta



Memoria Caché

- Memoria de alta velocidad.
- Encontrar los datos más recientes utilizados por el CPU.
- Puede existir independiente para instrucciones y para datos.
- Puede estar organizada en niveles: L1, L2.

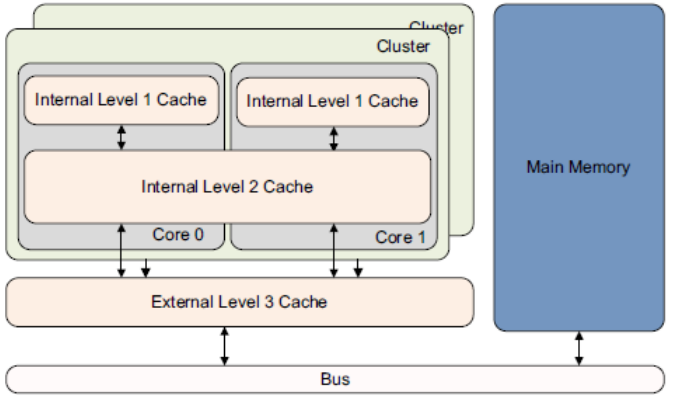


Figure 11-1 A basic cache arrangement

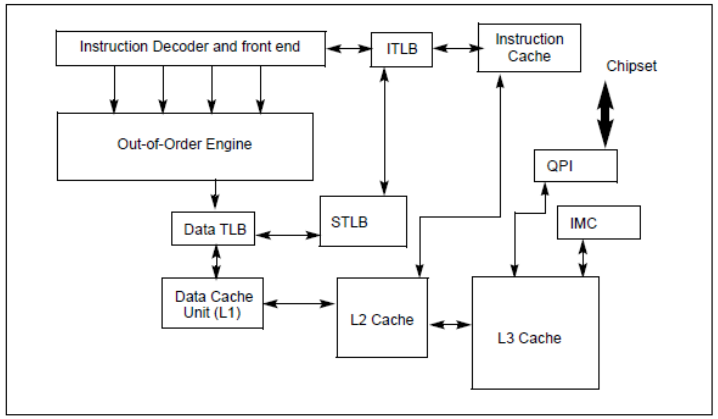


Figure 12-2. Cache Structure of the Intel Core i7 Processors

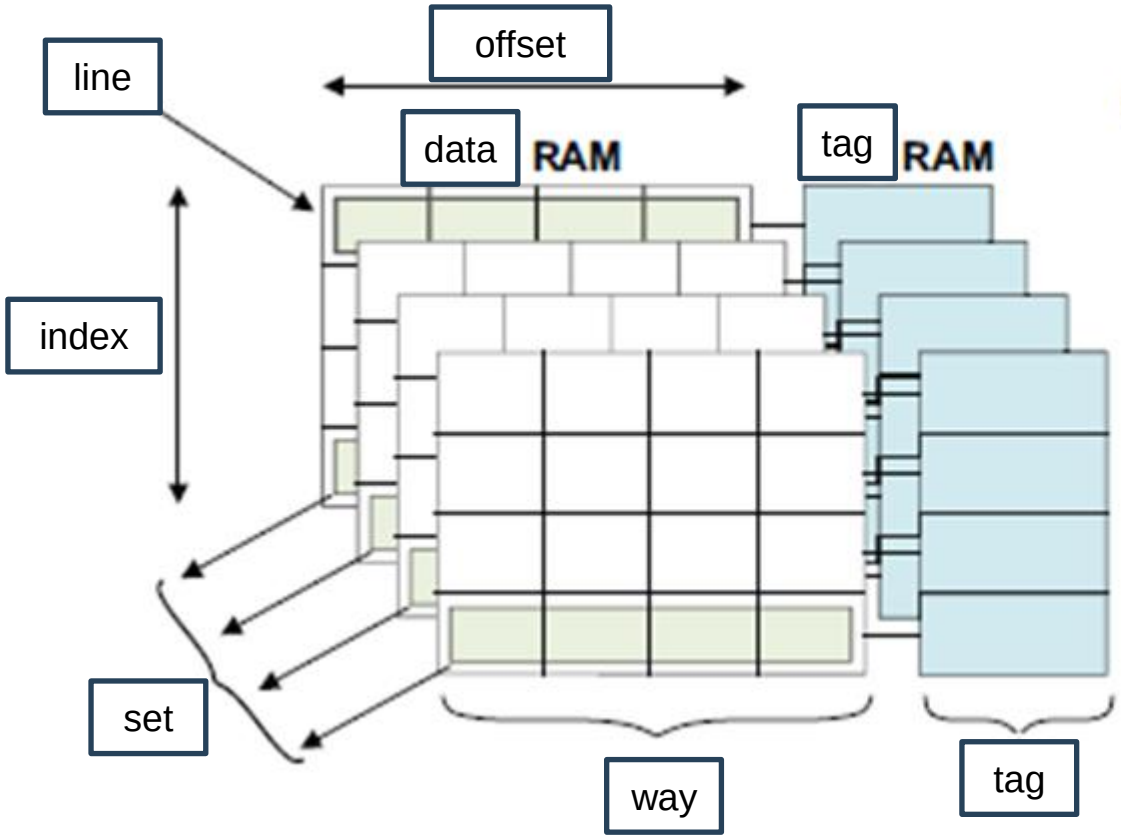
Terminología de Caché

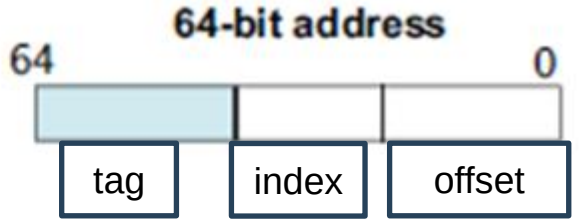
- **Caché Hit:** CPU busca un dato en la caché y se encuentra.
- **Caché Miss:** CPU busca un dato en la caché y no se encuentra o se encuentra invalidado.
- **Miss Penalty:** Penalización (tiempo para acceder a un nivel superior) producto de un miss de caché.
- **Políticas de reemplazo:** formas en que las caché reemplazan bloques de memoria.
- **Correspondencia:** método en que se mapean bloques de memoria principal a la memoria caché.

Caché miss

- Son inevitables y dependiendo del contexto se pueden categorizar:
- Compulsoy/Cold misses
- Capacity misses
- Conflict misses
- **Coherency misses**

Estructura básica caché



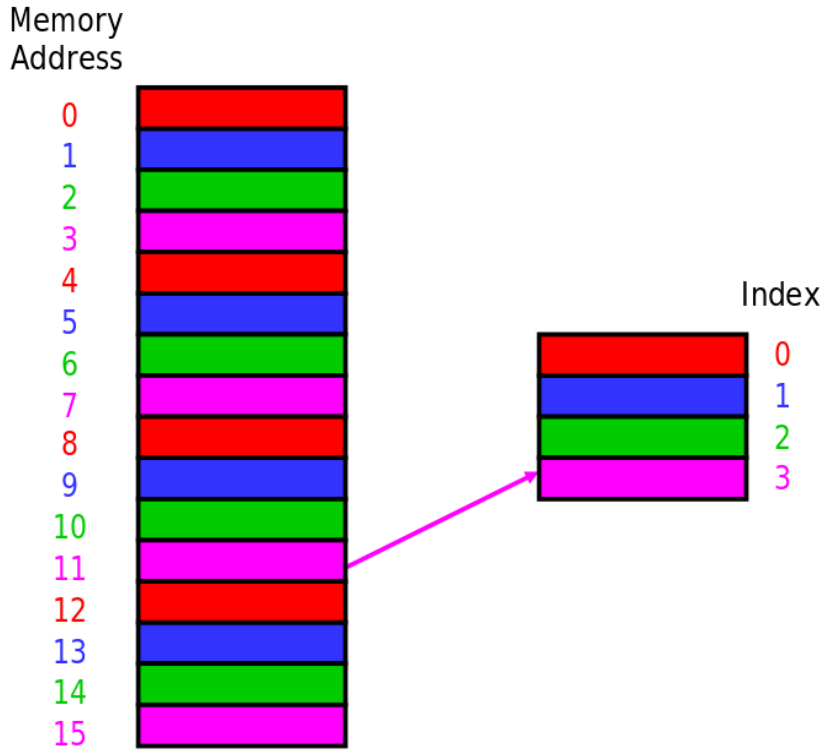


1. tag
2. index
3. offset
4. line
5. set
6. way
7. data

Caché – Correspondencia Directa

- Cada bloque de memoria principal está mapeado a un único bloque en la caché.

$$B_{cache} = B_{mem} mod\ NB_{cache}$$

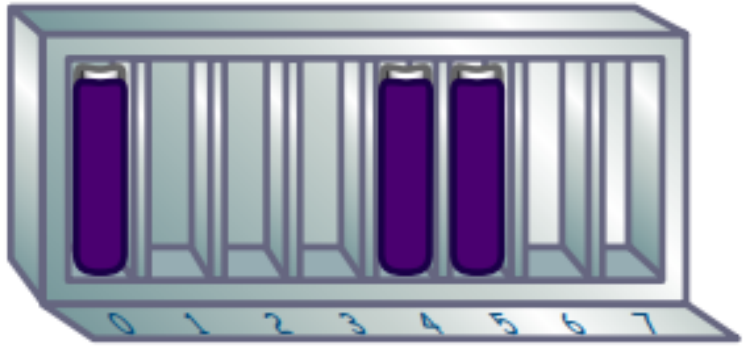


B_{mem}: bloque de memoria principal.

B_{cache}: bloque en caché.

NB_{cache}: número de bloque de la caché.

Caché – Correspondencia Directa



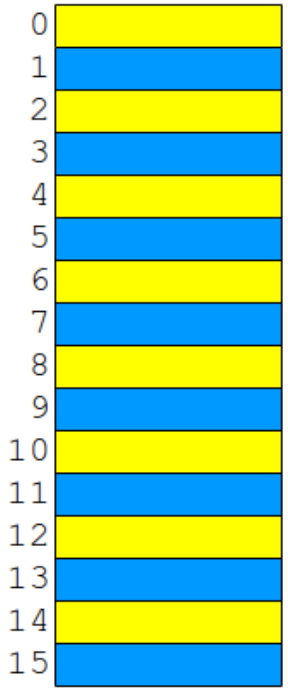
Direct Mapped

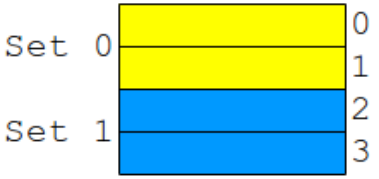
A cache block can only go in one spot in the cache. It makes a cache block very easy to find, but it's not very flexible about where to put the blocks.

Caché – Asociativa por Set

- Cada bloque de memoria principal puede mapearse de *n* formas a la caché.

$$S_{cache} = B_{mem} mod\ N$$



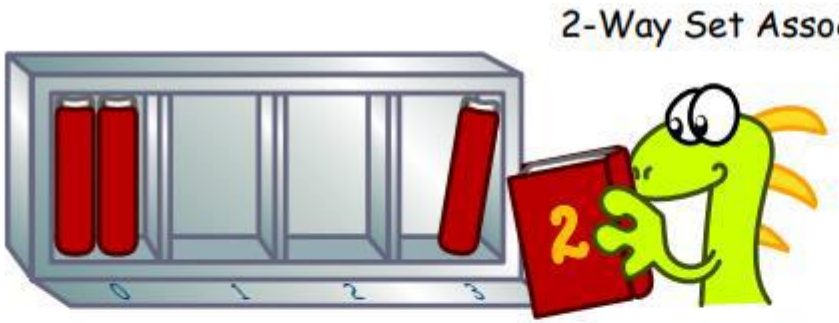


B_{mem}: bloque de memoria principal.

S_{cache}: bloque en caché.

N: número de sets en caché.

Caché – Asociativa por Set



Tag	Index	Offset

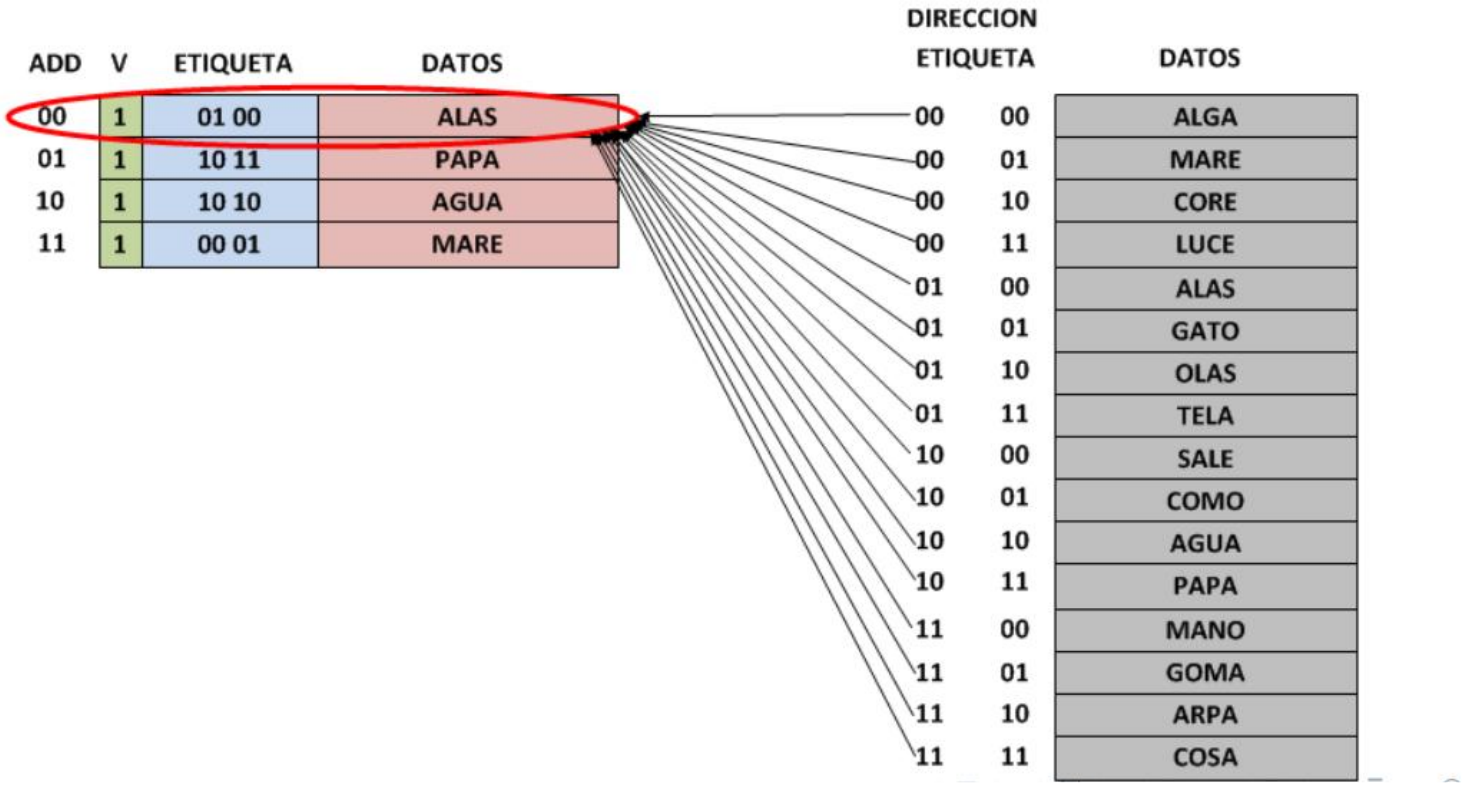
This cache is made up of sets that can fit two blocks each. The index is now used to find the set, and the tag helps find the block within the set.

Tag	Index	Offset

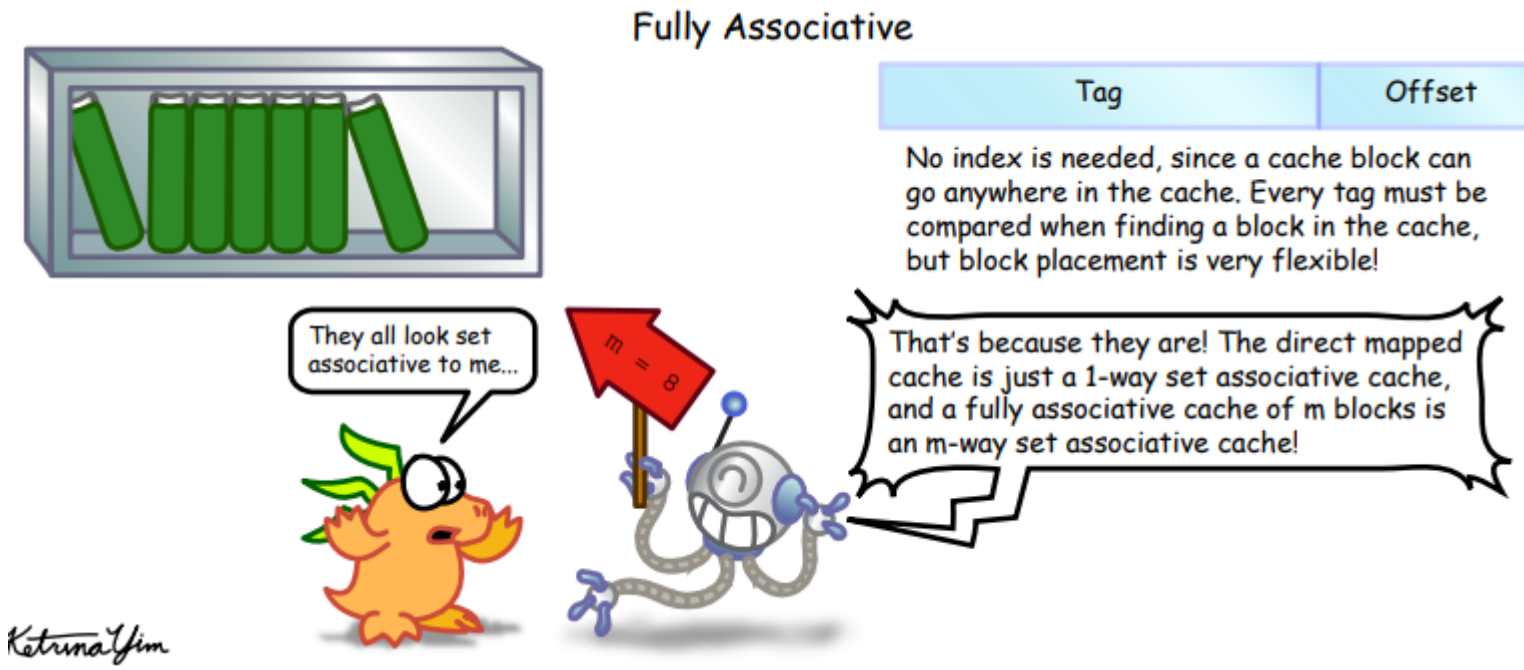
Each set here fits four blocks, so there are fewer sets. As such, fewer index bits are needed.

Caché – Asociativa (completa)

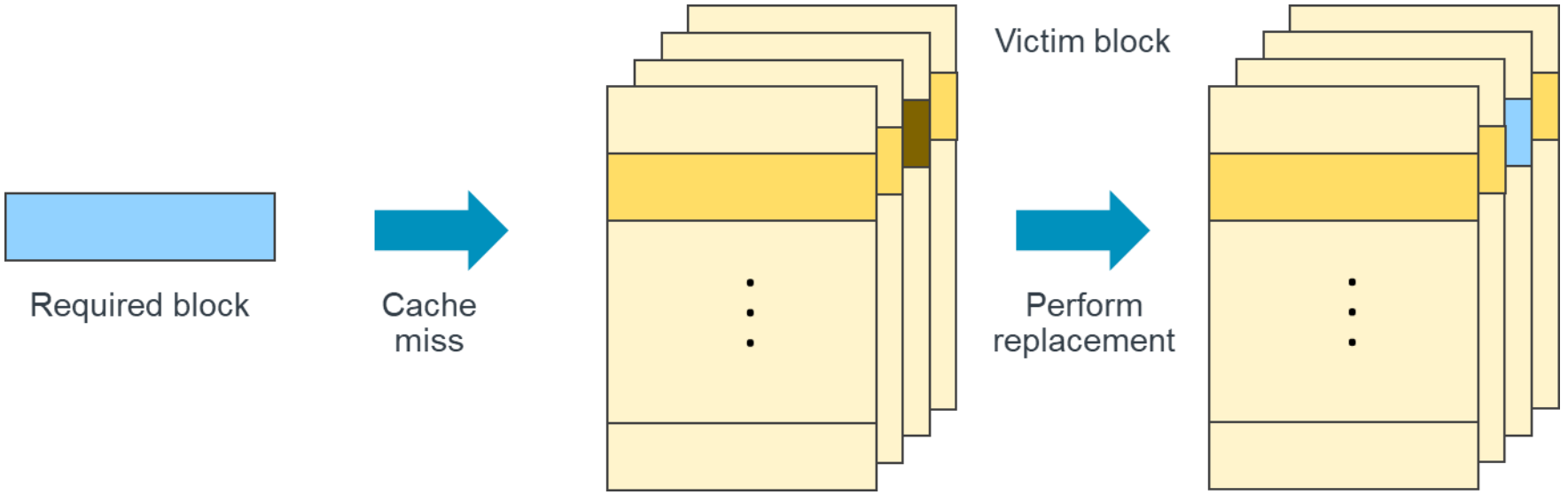
- Cada bloque de memoria principal puede mapearse a cualquier bloque de caché.



Caché – Asociativa (completa)



Políticas de remplazo



- **Mapeo directo:** un candidato.
- **Set n-way:** uno de los bloques.
- **Asociativo completo:** cualquier bloque.

Políticas de remplazo

- Random : entre los posibles bloques, se selecciona uno de manera aleatoria.
- Least Recently Used(LRU): se reemplaza el bloque haya sido utilizado menos recientemente.
- FIFO: Temporalmente, el primer bloque en ser escrito es el que será reemplazado.

Políticas de escritura

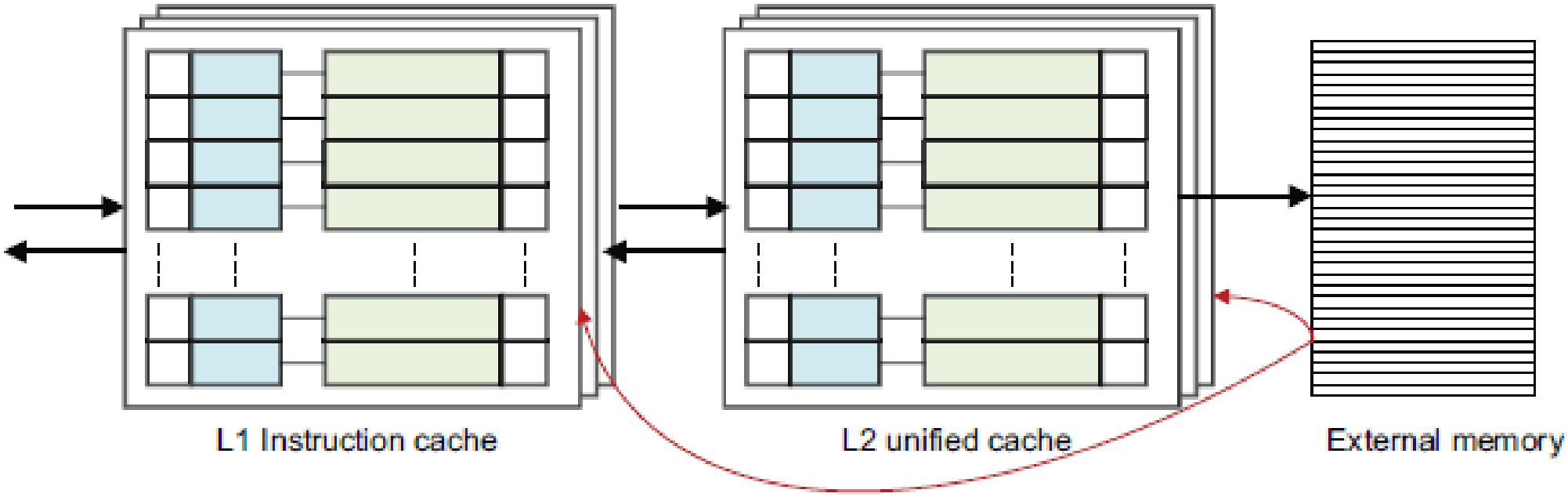


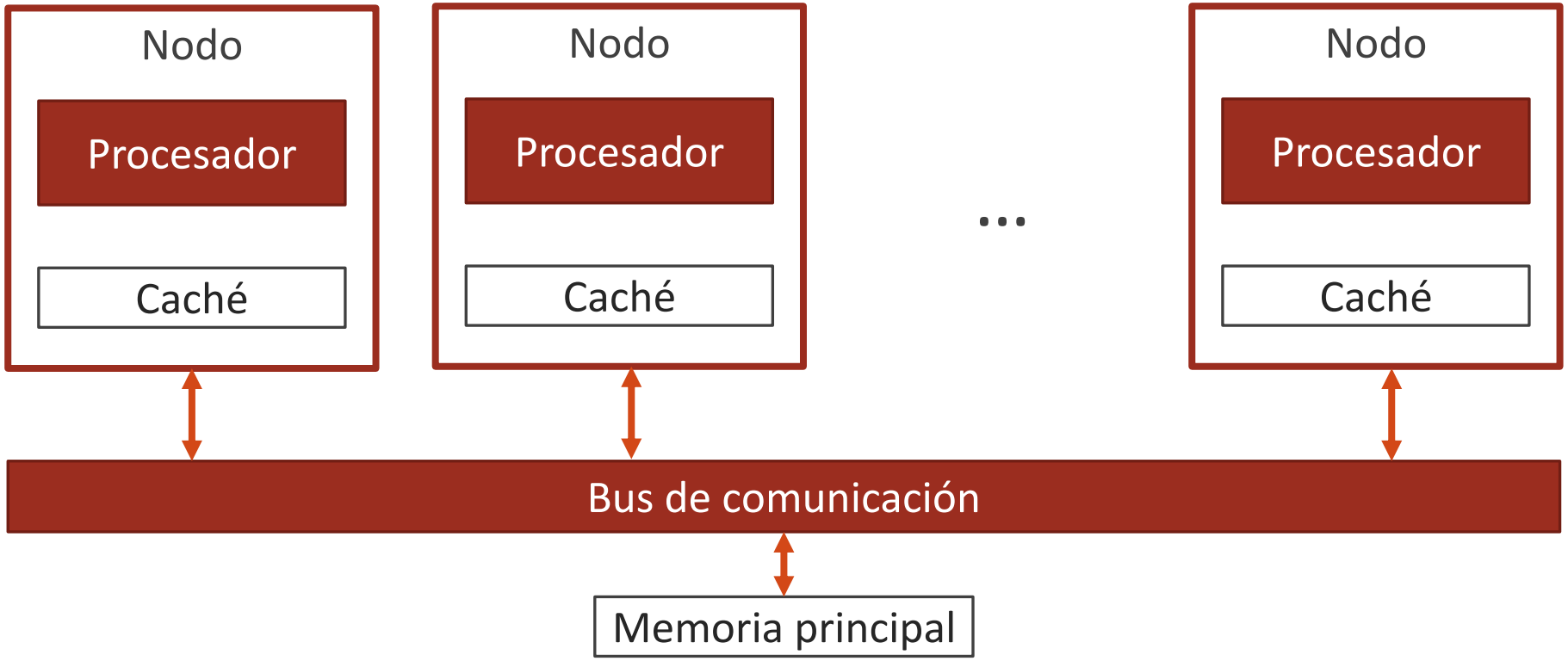
Figure 11-7 Found in external memory

Políticas de escritura

- Write-Back: La escritura se da únicamente en el cache de esta forma se genera incoherencia (MEM[x] != $[x])
- Write around: Se hace bypass al cache y se escribe únicamente en memoria
- Write-Through: La escritura se realiza en cache y Memoria

Investigue ¿Qué es Write-Allocate, Read-Allocate como es manejado en HW y SW?

Problema de coherencia de caché



Coherencia

- Cuando los datos están compartidos y están en caché, se replican en otras caché.
- **Problema de coherencia:** en un mismo instante de tiempo, las múltiples cachés pueden tener la misma variable con valores distintos.

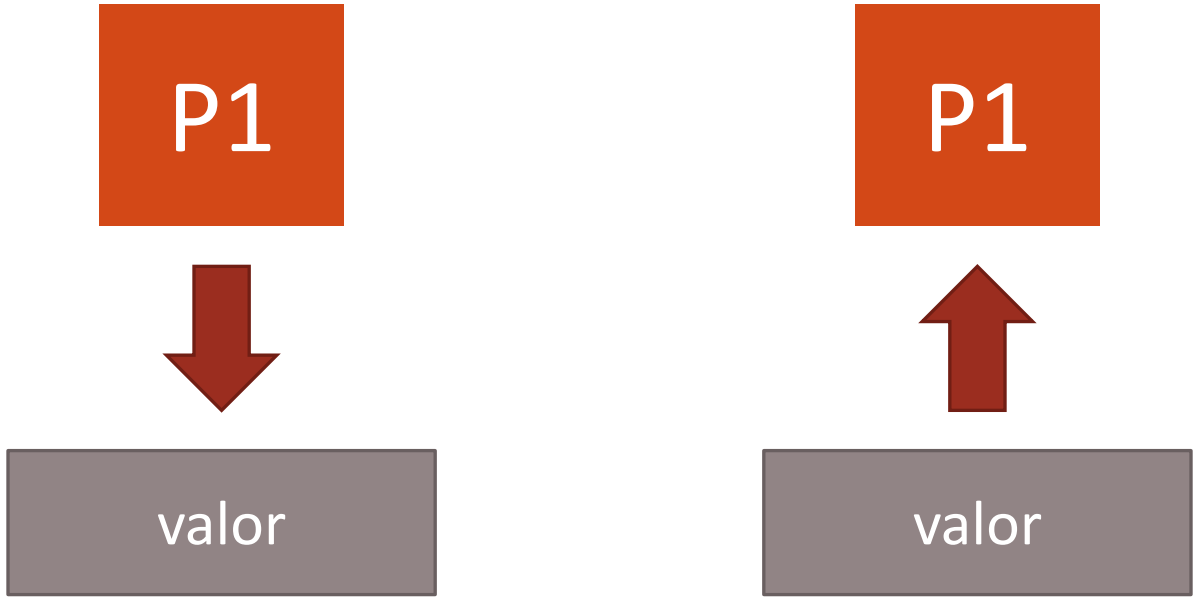
Incoherencia

Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0				1
1	Processor A reads X	1		1
2	Processor B reads X	1	1	1
3	Processor A stores 0 into X	0	1	0

Problema de la coherencia

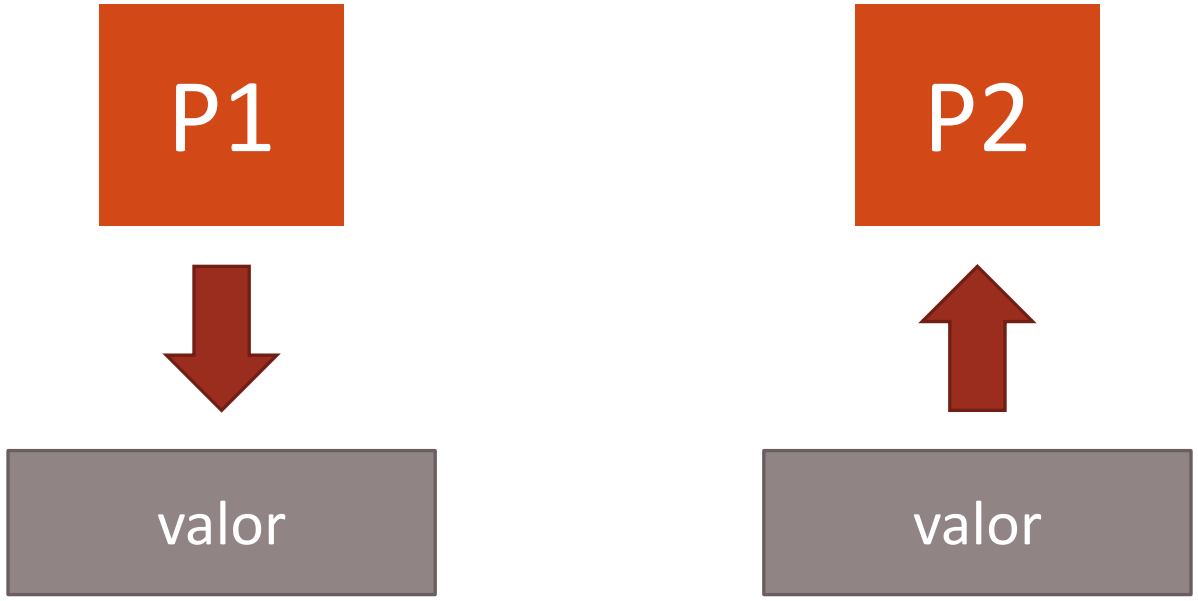
- La escritura va a generar incoherencia en las caché.
- Las copias son invalidadas o actualizadas.

Condiciones de coherencia – Escenario #1



valor escrito
=
valor leído

Condiciones de coherencia – Escenario #2



valor escrito
=
valor leído

Condiciones de coherencia

- Una **lectura** de un procesador P a una dirección **X**, luego de una **escritura** a X por P, sin que existan escrituras a X por otro procesador, siempre debe retornar el valor escrito por P.
- Una **lectura** por un procesador a un dirección X, **luego** de una **escritura** de otro procesador a X, retorna el **valor escrito**, si la escritura y la lectura están suficientemente separadas en tiempo, y no ocurre otra escritura entre ellas.
- Escrituras a la misma dirección son serializadas: Las escrituras son vistas en el mismo orden por todos los procesadores.

Caché en Sistemas Coherentes

En procesadores coherentes, las caché deben tener:

- **Migración:** mover un dato de una caché local a otra.
  - Reduce el costo de acceso a memoria y ancho de banda.
- **Replicación:** copias de los datos compartidos en cada caché local.
  - Reduce tiempo y ancho de banda.
- Las implementaciones de coherencia deben soportar ambas capacidades.

Protocolos de coherencia

El objetivo es vigilar los datos compartidos para:

- **Consistencia de datos:** todos los observadores vean el mismo dato en la misma posición de memoria.
- **Consistencia secuencial:** todos los observadores ven el mismo orden de las actualizaciones en diferentes posiciones de memoria.

Posibles soluciones

Se debe rastrear el estado de los bloques compartidos.

- Protocolo de monitoreo.
- Protocolo basado en directorios.

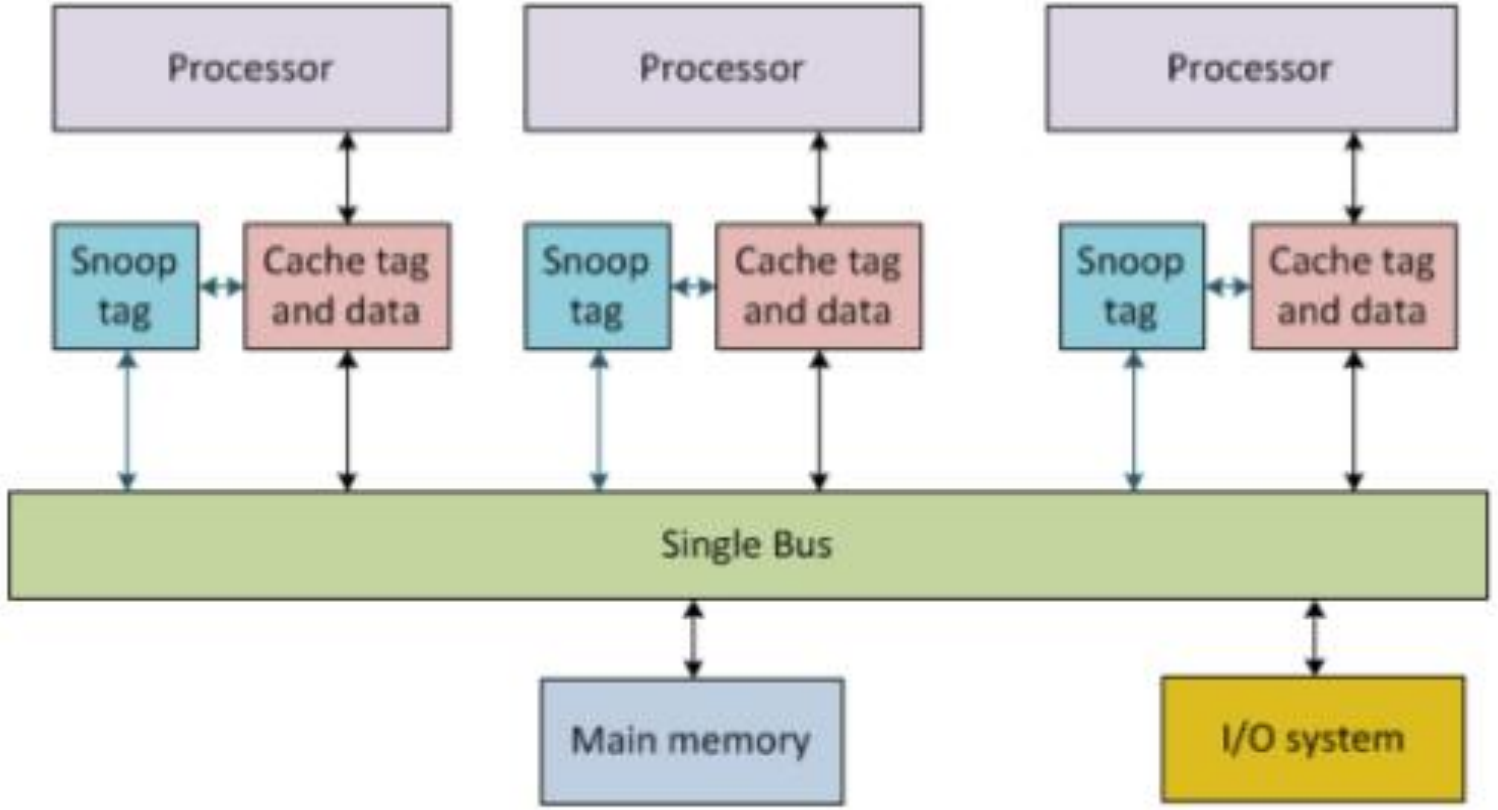
Protocolo de monitoreo

- **Monitoreo en el bus:** controladores de memoria monitorean para determinar si existe copia en caché del bloque solicitado.
- El bloque debe tener un estado (bit de validez).
- Conveniente para arquitecturas con memoria centralizada con baja cantidad de procesadores.

Protocolo de monitoreo

- Puede existir interferencia con las operaciones de los procesadores.
  - El monitoreo es en el bus.
- Para solucionar esto, se duplica la parte de la dirección de caché que corresponde a la etiqueta.
- En la práctica se agrega un puerto extra de lectura a la posición de caché que corresponde a la etiqueta.

Protocolo de monitoreo



Protocolo de monitoreo

- **Invalidación por escritura:** en cada escritura hacia memoria se invalidan las caché de todos los otros procesadores.
- **Actualización por escritura:** en cada escritura se genera una petición general para que se actualice cada una de las caché.

Invalidación por escritura

- Se monitorea el bus para verificar si existen copias.
- En caso positivo, se invalidan los bloques.
- Bus se utiliza únicamente durante la primera escritura para invalidar el resto.
- Un solo escritor, múltiples lectores.
- La información no se actualiza inmediatamente.

Actualización por escritura

- Todos los controladores de caché monitorean el bus para verificar si tienen copia en caché.
- Se debe actualizar la copia en la caché correspondiente.
- **Ventajas:**
  - Reduce desaciertos de caché.
- **Desventajas:**
  - Mayor uso del bus.

Monitoreo: implementación

- Método de broadcast para invalidación: bus compartido.
- Serializar las invalidaciones por escrituras simultáneas a bloques compartidos.
- **Bloque sucio:** bloque de caché local difiere con la memoria principal, pero es el más actual.
- **Bloque inválido:** un bloque en caché que difiere al más actual.
- El monitoreo es sobre la etiqueta de la caché (o una copia del bloque).

Monitoreo: implementación

- **Bloque compartido:**
  - Un bit para denotar compartido – exclusivo.
  - Cuando un procesador genera una invalidación a un bloque compartido, se cambia el estado a exclusivo.
  - El estado exclusivo es hasta que otro procesador requiera una lectura del bloque.
  - Escrituras sobre bloques exclusivos no se envían al bus.
  - No se generan invalidaciones sobre un bloque exclusivo.

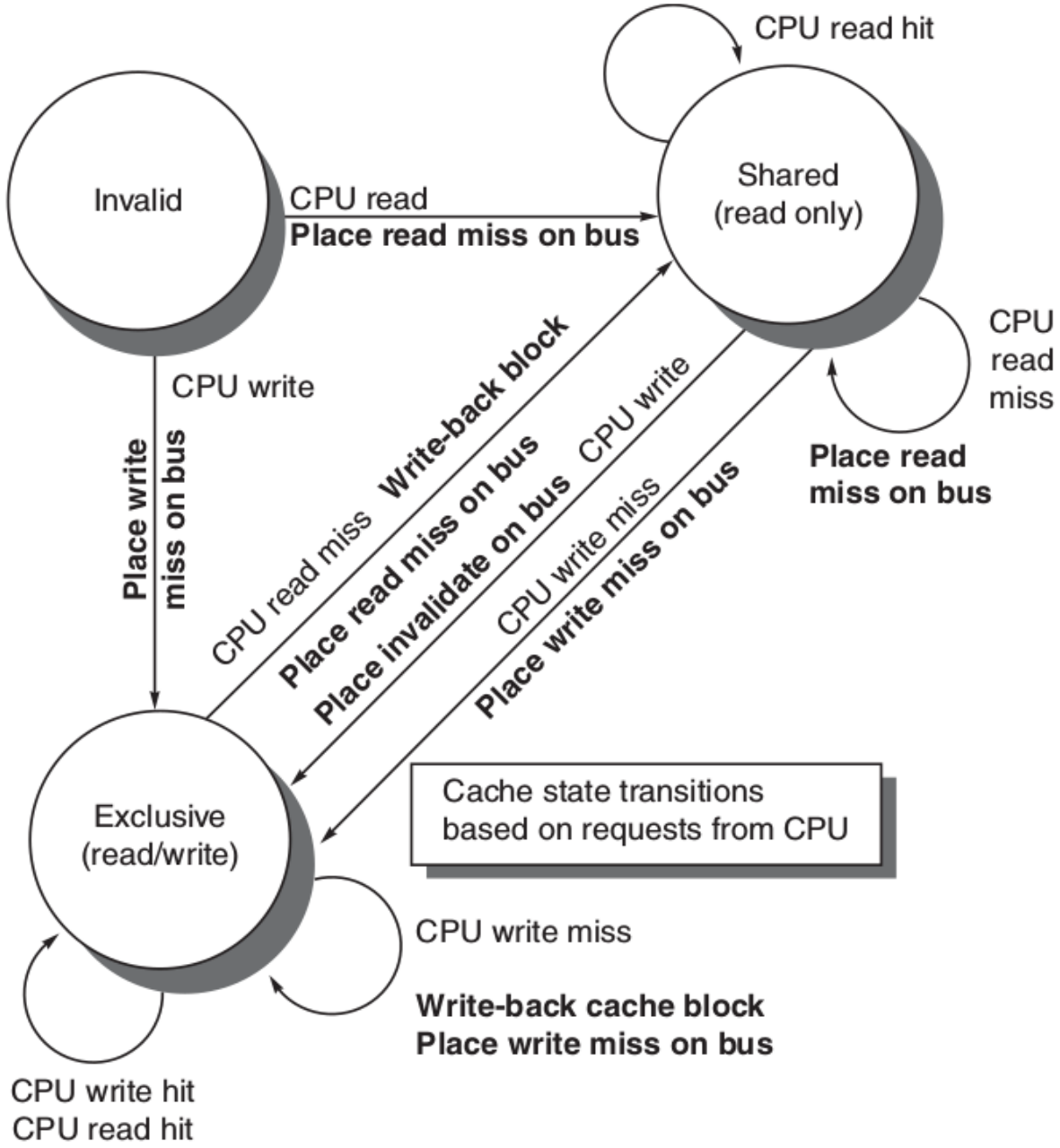
Protocolo MSI

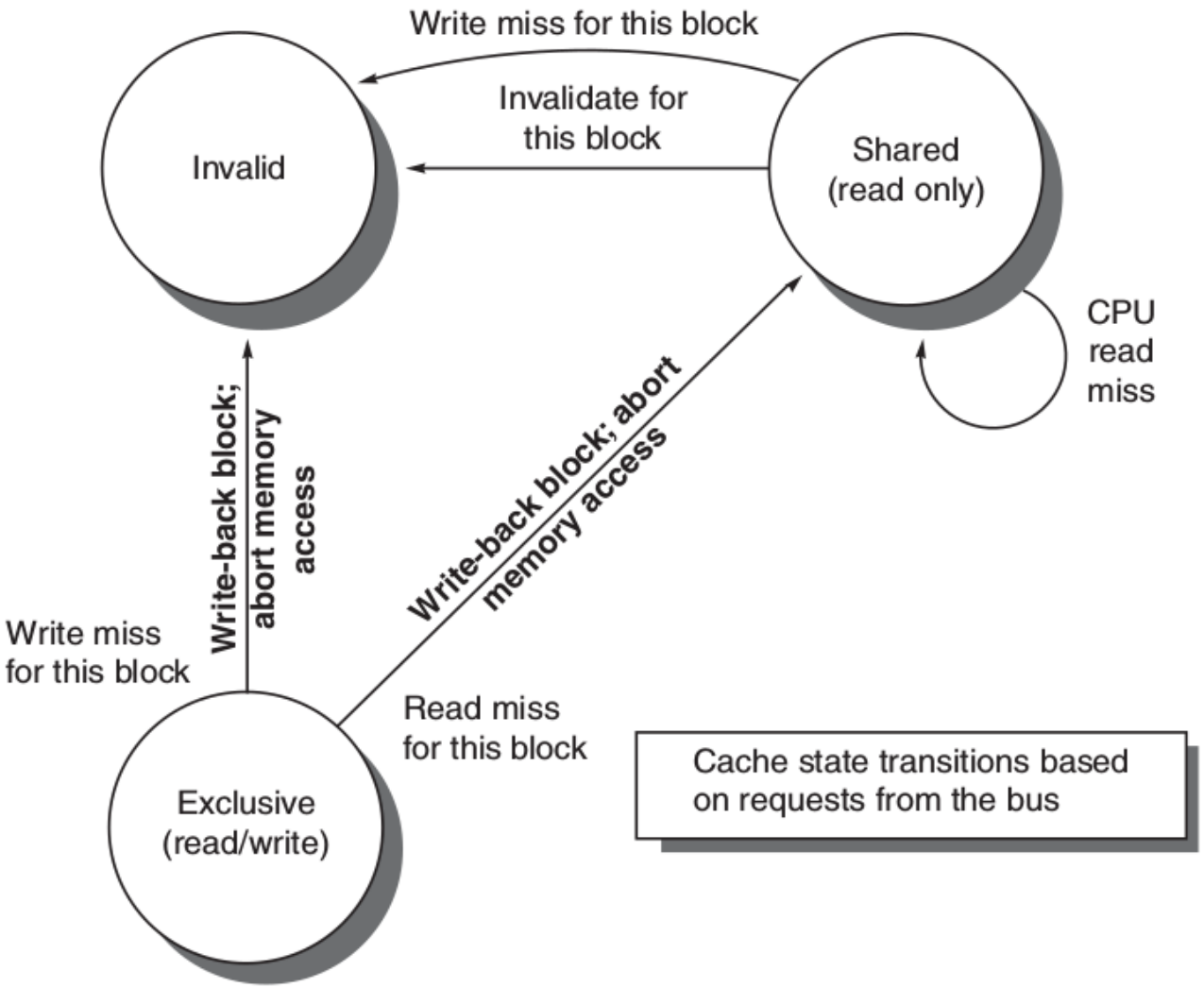
Hay tres estados posibles:

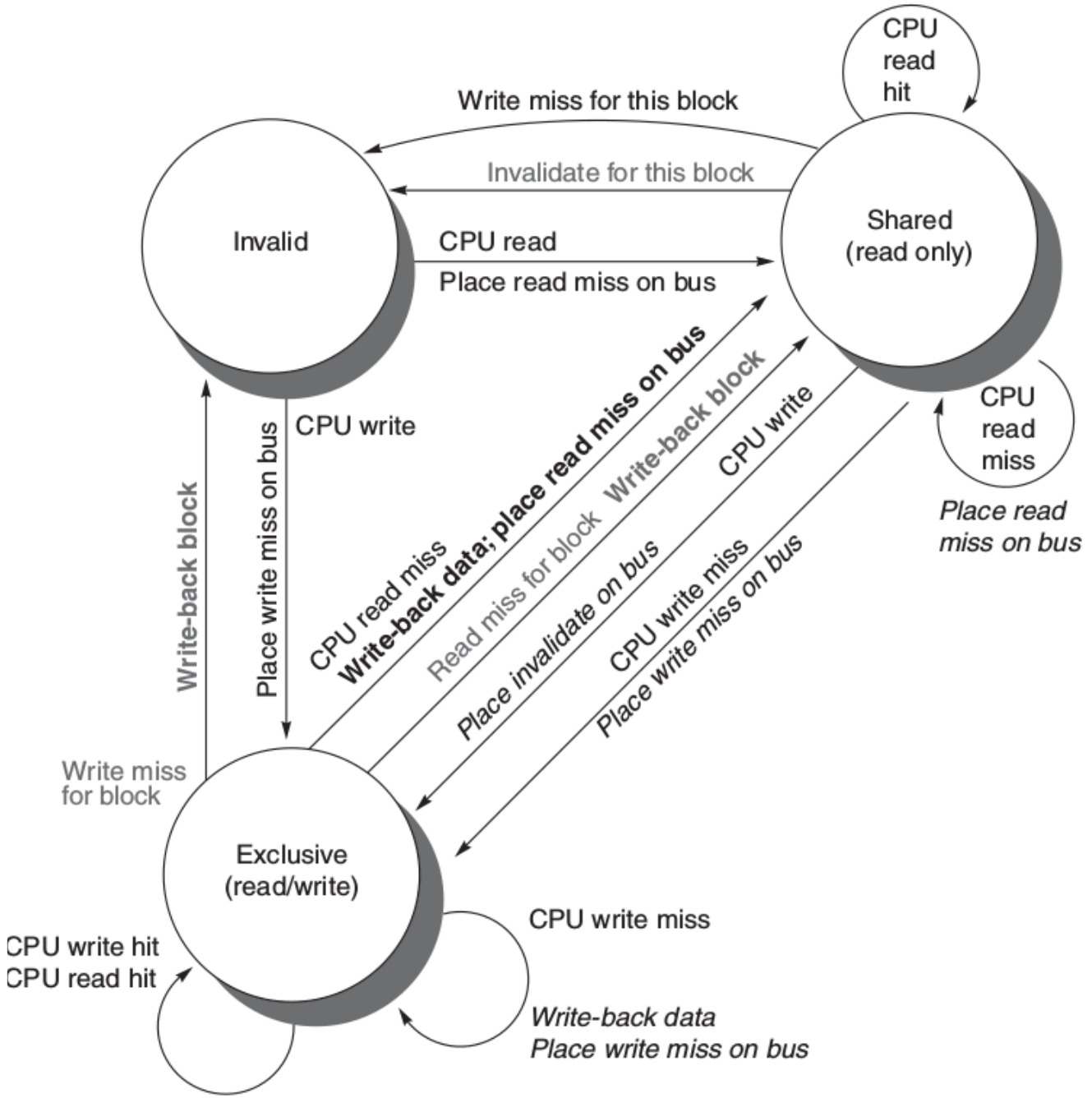
- Modificado en caché local (implica exclusivo).
- Compartido.
- Inválido.

- Implementación de controlador por máquina de estados finitos.
  - M, S, I

Request	Source	addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.


CPU read hit
Invalid
CPU read
Place read miss on bus
Shared
(read only)
CPU
read
miss
Place read
miss on bus
CPU write
Place write
miss on bus
CPU read miss
Write-back block
Place read miss on bus
Place invalidate on bus
CPU write
CPU write miss
Place write miss on bus
Exclusive
(read/write)
Cache state transitions
based on requests from CPU
CPU write miss
Write-back cache block
Place write miss on bus
CPU write hit
CPU read hit


Write miss for this block
Invalidate for
this block
Invalid
Shared
(read only)
CPU
read
miss
Write-back block;
abort memory
access
Write miss
for this block
Write-back block; abort
memory access
Read miss
for this block
Exclusive
(read/write)
Cache state transitions based
on requests from the bus


CPU
read
hit
Write miss for this block
Invalidate for this block
Invalid
Shared
(read only)
CPU read
Place read miss on bus
CPU write
Write-back block
Place write miss on bus
CPU read miss
Write-back data; place read miss on bus
Read miss for block
Write-back block
Place invalidate on bus
CPU write
CPU write miss
Place write miss on bus
CPU
read
miss
Place read
miss on bus
Write miss
for block
Exclusive
(read/write)
CPU write miss
CPU write hit
CPU read hit
Write-back data
Place write miss on bus

Mejora de MSI

- MESI.
- MOESI.

Protocolo MESI

- Estado E (exclusivo).
- Un bloque en E puede ser escrito sin generar invalidaciones.
- Ante un *miss* de otro procesador al bloque debe cambiar el estado a S.
- Escrituras consecutivas a un mismo bloque en estado E no necesita el bus o invalidar.
- En escritura se pasa de E a M.

Protocolo MOESI

- Estado O (Owned).
- Designa que un bloque le pertenece a esa caché y está desactualizado en memoria.
- En MSI y MESI cuando se comparte un bloque en M se cambia a S y se escribe en memoria.
- En MOESI se pasa de M a O sin necesidad de escribir a memoria.
- Sólo una caché tendrá el bloque en estado O, las demás lo mantendrán en S.

Protocolo MOESI

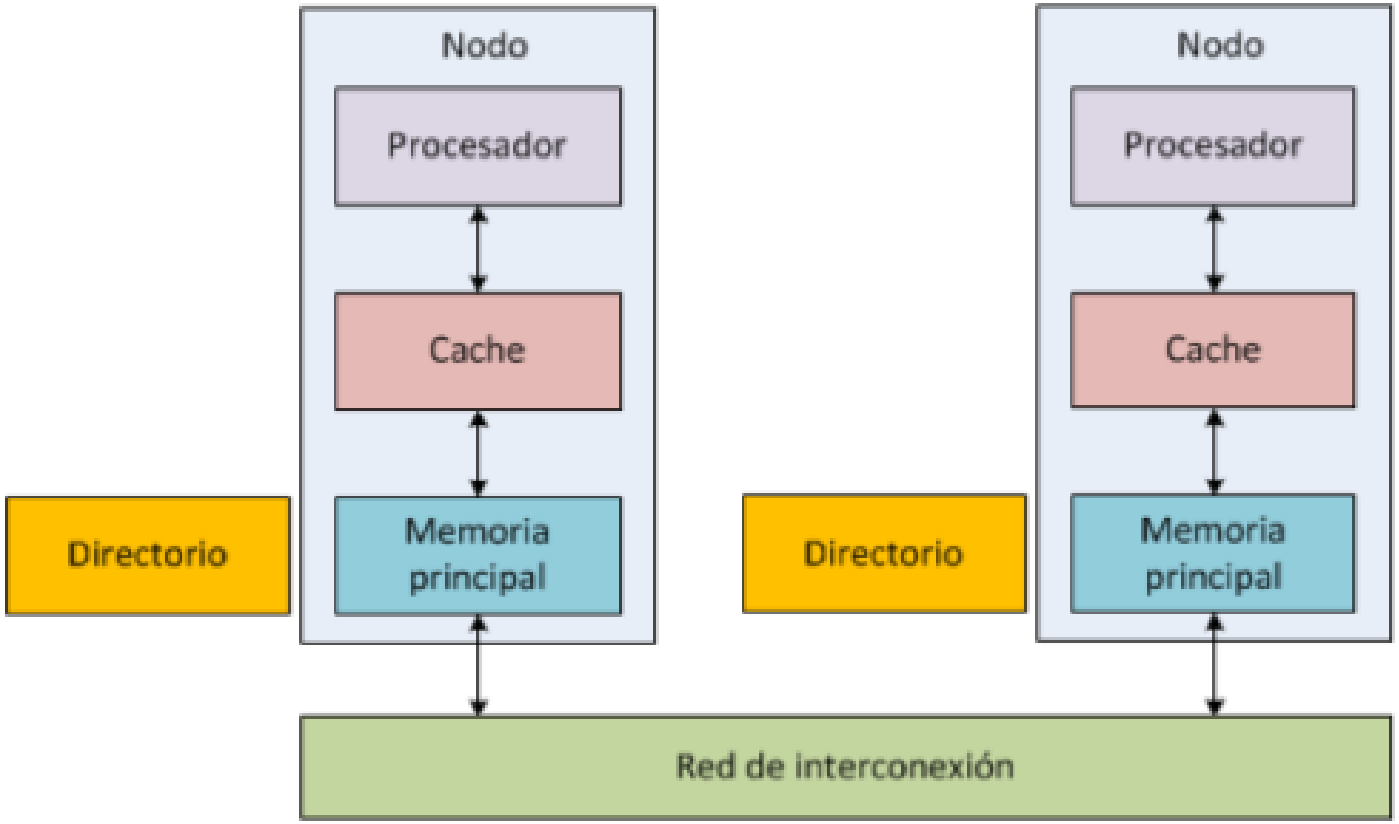
- En un *miss* el dueño del bloque debe darlo.
  - La memoria estará desactualizada.
  - Debe hacerlo a los otros procesadores o a memoria.

Basado en directorios

El estado de un bloque de memoria física en las caché se mantiene en un directorio.

- Cada entrada del directorio es un bloque de memoria.
- En memoria distribuida, el directorio también es distribuido.
- Las solicitudes se realizan punto a punto con los procesadores (procesador – directorio)
  - Evitar write-through.

Basado en directorios



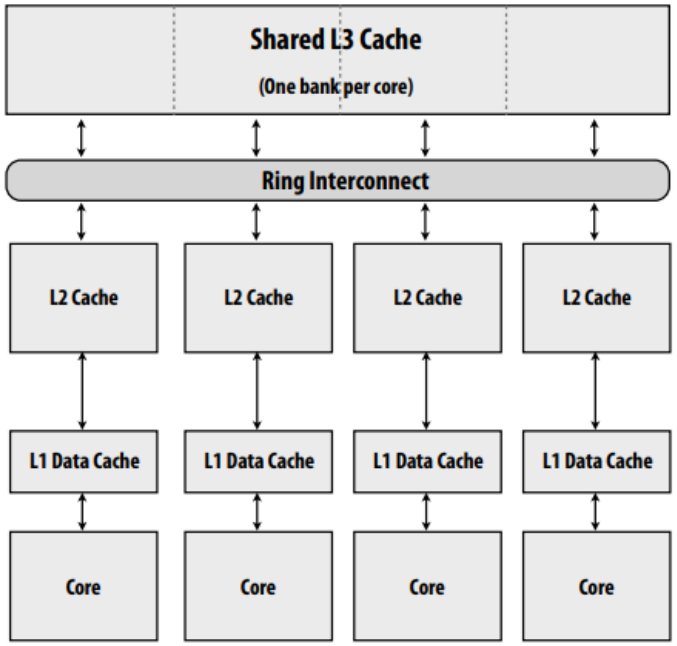
Basado en directorios

El directorio tiene:

- Estado de cada bloque.
  - **Compartido:** uno o más procesadores tienen el bloque de caché.
  - **Uncached:** ningún procesador tiene copia.
  - **Exclusivo:** sólo un procesador tiene datos que han sido modificados.
- Procesadores con copia de bloque.
- Procesador dueño del bloque.

Basado en directorios

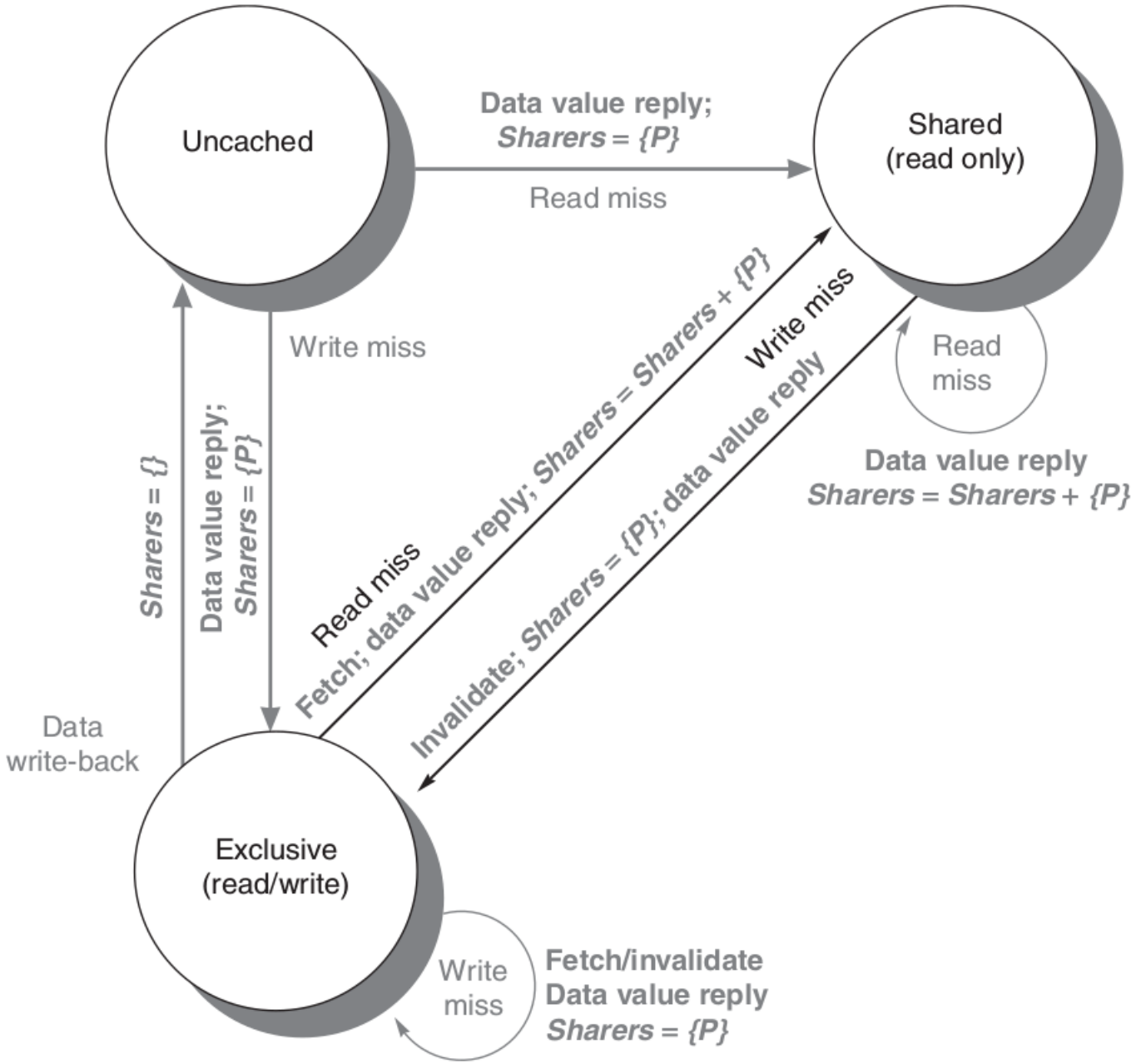
Directory coherence in Intel Core i7 CPU



- L3 serves as centralized directory for all lines in the L3 cache
  (note importance of inclusion property… there will be a directory entry for any line in an L2)
- Directory maintains list of L2 caches containing line
- Instead of broadcasting coherence traffic to all L2's, only send coherence messages to L2's that contain the line
  (Core i7 interconnect is a ring, it is not a bus)
- Directory dimensions:
  - P=4
  - M = number of L3 cache lines

Mensajes entre nodos

Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	Home directory	Local cache	D	Return a data value from the home memory.
Data write-back	Remote cache	Home directory	A, D	Write-back a data value for address A.


Uncached
Data value reply;
Sharers = {P}
Read miss
Shared
(read only)
Write miss
Sharers = {}
Data value reply;
Sharers = {P}
Read miss
Fetch; data value reply; Sharers = Sharers + {P}
Write miss
Invalidate; Sharers = {P}; data value reply
Read
miss
Data value reply
Sharers = Sharers + {P}
Data
write-back
Exclusive
(read/write)
Write
miss
Fetch/invalidate
Data value reply
Sharers = {P}

Consistencia Secuencial de Memoria (SC)

Considere los siguientes *threads*

```
1    Thread 1
2
3    (1) x = 1
4    (2) print(y)
```

```
1    Thread 2
2
3    (3) y = 1
4    (4) print(x)
```

¿Cuáles son las posibles combinaciones de ejecución de (1), (2), (3) y (4)?

```
(1) -> (2) -> (3) -> (4)  We get "0" and "1"
(3) -> (4) -> (1) -> (2)  We get "0" and "1"
```

```
(1) -> (3) -> (2) -> (4)  We get "1" and "1"
(1) -> (3) -> (4) -> (2)  We get "1" and "1"
```

Consistencia Secuencial de Memoria (SC)

¿Es “0” and “0” un caso posible?

```
1    Thread 1
2
3    (1) x = 1
4    (2) print(y)
```

```
1    Thread 2
2
3    (3) y = 1
4    (4) print(x)
```

¿Es esperado que (2) se ejecute antes que (1) ?

¿Es esperado que (4) se ejecute antes que (3) ?

Consistencia Secuencial de Memoria (SC)

Al definir un orden permitido de ejecución *threads* se define un modelo de consistencia de memoria

```
(1) -> (2) -> (3) -> (4)   We get "0" and "1"
(3) -> (4) -> (1) -> (2)   We get "0" and "1"
(1) -> (3) -> (2) -> (4)   We get "1" and "1"
(1) -> (3) -> (4) -> (2)   We get "1" and "1"
```

En el caso anterior definimos que "0" y "0" no es posible en este modelo

"A memory consistency model is a contract between the hardware and software. ***The hardware promises to only reorder operations in ways allowed by the model****, and in return, the* ***software acknowledges that all such reorderings are possible and that it needs to account for them****"*

Consistencia Secuencial de Memoria (SC)

Existen modelos *weak* y *strong*

- Weak: permite casi cualquier reordenamiento de operaciones
  - Pros: permite optimizaciones de HW.
  - Dificultad para programar a bajo nivel.
- Strong: permite reordenamiento muy definido de operaciones
  - Pros: Determinista, simplifica al programador su trabajo
  - Cons: Reduce la capacidad de optimizaciones de HW

Reordenamiento de código

Durante compilación se puede reorganizar el código de forma que se optimice el uso de procesador.

Regla general:
"…Thou shalt not modify the behavior of a single-threaded program.."

```
1  int A, B;
2
3  void foo()
4  {
5      A = B + 1;
6      B = 0;
7  }
```

```
> gcc -S -masm=intel foo.c
> cat foo.s
        ...
        mov     eax, DWORD PTR _B
        add     eax, 1
        mov     DWORD PTR _A, eax
        mov     DWORD PTR _B, 0
        ...
```

```
> gcc -O2 -S -masm=intel foo.c
> cat foo.s
        ...
        mov     eax, DWORD PTR B
        mov     DWORD PTR B, 0
        add     eax, 1
        mov     DWORD PTR A, eax
        ...
```

Reordenamiento de código

Del manual GCC:

O1 Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

O2 Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. The compiler does not perform loop unrolling or function in-lining when you specify -O2. As compared to -O, this option increases both compilation time and the performance of the generated code.

O3 Optimize yet more. -O3 turns on all optimizations specified by -O2 and also turns on the -finline-functions, -funswitch-loops, -fpredictive-commoning, -fgcse-after-reload and -ftree-vectorize options.

O0 Reduce compilation time and make debugging produce the expected results. This is the default.

Os Optimize for size. -Os enables all -O2 optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

Reordenamiento de código

La reorganización de código puede ser evitado mediante instrucciones al compilador

```
1  int A, B;
2
3  void foo()
4  {
5      A = B + 1;
6      asm volatile("" ::: "memory");
7      B = 0;
8  }
```

```
> gcc -O2 -S -masm=intel foo.c
> cat foo.s
        ...
        mov     eax, DWORD PTR _B
        add     eax, 1
        mov     DWORD PTR _A, eax
        mov     DWORD PTR _B, 0
        ...
```

Qualifiers

`volatile`

The typical use of extended `asm` statements is to manipulate input values to produce output values. However, your `asm` statements may also produce side effects. If so, you may need to use the `volatile` qualifier to disable certain optimizations. See Volatile.

Reordenamiento de código

La reorganización de código puede ser evitado mediante *instrucciones al compilador*

Normal

```
> gcc -S -masm=intel foo.c
> cat foo.s
        ...
        mov     eax, DWORD PTR _B
        add     eax, 1
        mov     DWORD PTR _A, eax
        mov     DWORD PTR _B, 0
        ...
```

Optimizado

```
> gcc -O2 -S -masm=intel foo.c
> cat foo.s
        ...
        mov     eax, DWORD PTR B
        mov     DWORD PTR B, 0
        add     eax, 1
        mov     DWORD PTR A, eax
        ...
```

Optimizado con *fence*

```
> gcc -O2 -S -masm=intel foo.c
> cat foo.s
        ...
        mov     eax, DWORD PTR _B
        add     eax, 1
        mov     DWORD PTR _A, eax
        mov     DWORD PTR _B, 0
        ...
```

Reordenamiento de código

6.3.10. Memory barrier and fence instructions

Both ARMv7 and ARMv8 provide support for different barrier operations. These are described in more detail in Chapter 13 *Memory Ordering*:

- *Data Memory Barrier* (DMB). This forces all earlier-in-program-order memory accesses to become globally visible before any subsequent accesses.
- *Data Synchronization Barrier* (DSB). All pending loads and stores, cache maintenance instructions, and all TLB maintenance instructions, are completed before program execution continues. A DSB behaves like a DMB, but with additional properties.
- *Instruction Synchronization Barrier* (ISB). This instruction flushes the CPU pipeline and prefetch buffers, causing instructions after the ISB to be fetched (or re-fetched) from cache or memory.

Reordenamiento de código

Fences x86

- **SFENCE** — Serializes all store (write) operations that occurred prior to the SFENCE instruction in the program instruction stream, but does not affect load operations.
- **LFENCE** — Serializes all load (read) operations that occurred prior to the LFENCE instruction in the program instruction stream, but does not affect store operations.[2]
- **MFENCE** — Serializes all store and load operations that occurred prior to the MFENCE instruction in the program instruction stream.

Note that the SFENCE, LFENCE, and MFENCE instructions provide a more efficient method of controlling memory ordering than the CPUID instruction.

Lecturas recomendadas

Memory Barriers Are Like Source Control Operations ->
https://preshing.com/20120710/memory-barriers-are-like-source-control-operations/

Memory Ordering Models
-> https://preshing.com/20120930/weak-vs-strong-memory-models/
-> https://en.cppreference.com/w/cpp/atomic/memory_order

Cache coherence video
-> https://www.youtube.com/watch?v=aFRboAI_3RU

https://www.arangodb.com/2021/02/cpp-memory-model-migrating-from-x86-to-arm

Referencias

- Stallings, W. (2003). Computer organization and architecture: designing for performance. Pearson Education India.
- Hennessy, J. L., & Patterson, D. A. (2011). Computer architecture: a quantitative approach. Elsevier.
- Murillo, C. & Aguilar, M. (2014). Introducción a Multiprocesadores.
- Yang J. (1998). Lectura de clase Review: Direct mapped caches.
- Snyder L. (2011). Lectura de clase: Cache implementation.

CE4302 – Arquitectura de Computadores II

Coherencia Caché

PROFESOR: ING. LUIS BARBOZA ARTAVIA