

Tema 10: Memoria Caché

Arquitectura de Computadoras

Ing. Nicolás Majorel Padilla (npadilla@herrera.unt.edu.ar)

<http://microprocesadores.unt.edu.ar/arqcom/>

¿Cuáles temas veremos?



- ▶ Primera parte: **Generalidades**
 - ▶ Introducción. Conceptos generales.
 - ▶ *Performance, Price & Power.*
 - ▶ Conceptos de **Paralelismo** y *Pipelining*.
- ▶ Segunda parte: **Procesadores**
 - ▶ ISA: qué es, cómo se diseña.
 - ▶ Diseños de procesadores para *performance*.
- ▶ Tercera parte: **Subsistema de Memoria**
 - ▶ Memoria Caché.
 - ▶ Memoria Virtual.
- ▶ Cuarta parte: **Sistema completo**
 - ▶ Subsistemas de Entrada/Salida y de Comunicaciones.
 - ▶ Consumo de energía.
- ▶ Quinta parte: **Nuevos paradigmas**
 - ▶ Arquitecturas específicas.
 - ▶ Seguridad.

Temas que veremos

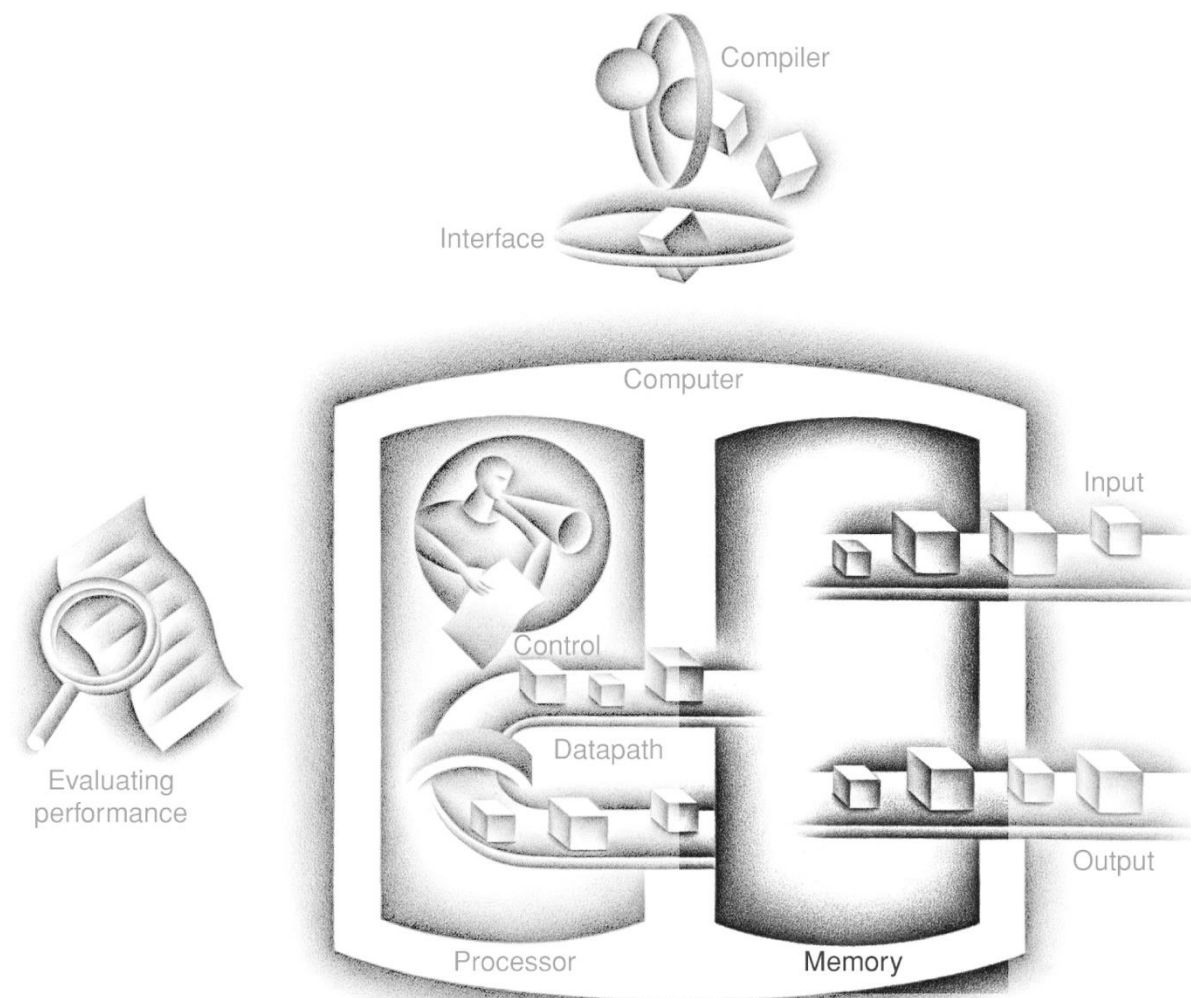
- ▶ Jerarquía de Memorias.
 - ▶ Motivación e importancia.
- ▶ Terminología.
- ▶ Organizaciones de caché.
 - ▶ Parámetros de diseño.
- ▶ Descomposición de direcciones de memoria.
- ▶ Compromisos y Criterios de diseño.

Lectura recomendada

- ▶ Computer Organization and Design, RISC-V Edition (2da ed, 2021)
 - ▶ Sección 5.1: *Introduction*
 - ▶ Sección 5.3: *The Basics of Caches*
 - ▶ Sección 5.8: *A Common Framework for Memory Hierarchy*
- ▶ Para los más curiosos:
 - ▶ Sección 5.2: *Memory Technologies*

Repaso: Partes de una computadora

- ▶ Vimos muchos temas sobre procesadores, sus caminos de datos y su control.
- ▶ Ahora pasamos el foco a la memoria.



Principio de Localidad – Definición

- ▶ Un programa accede a una porción relativamente pequeña del espacio de direcciones, en cualquier intervalo de tiempo.
- ▶ Dos tipos diferentes de localidad.
 - ▶ **Temporal:** cuando un ítem es accedido, es probable que pronto sea accedido nuevamente.
 - ▶ **Espacial:** cuando un ítem es accedido, es probable que pronto sean accedidos ítems cercanos.
- ▶ ¡Aplicable en casi cualquier ámbito!

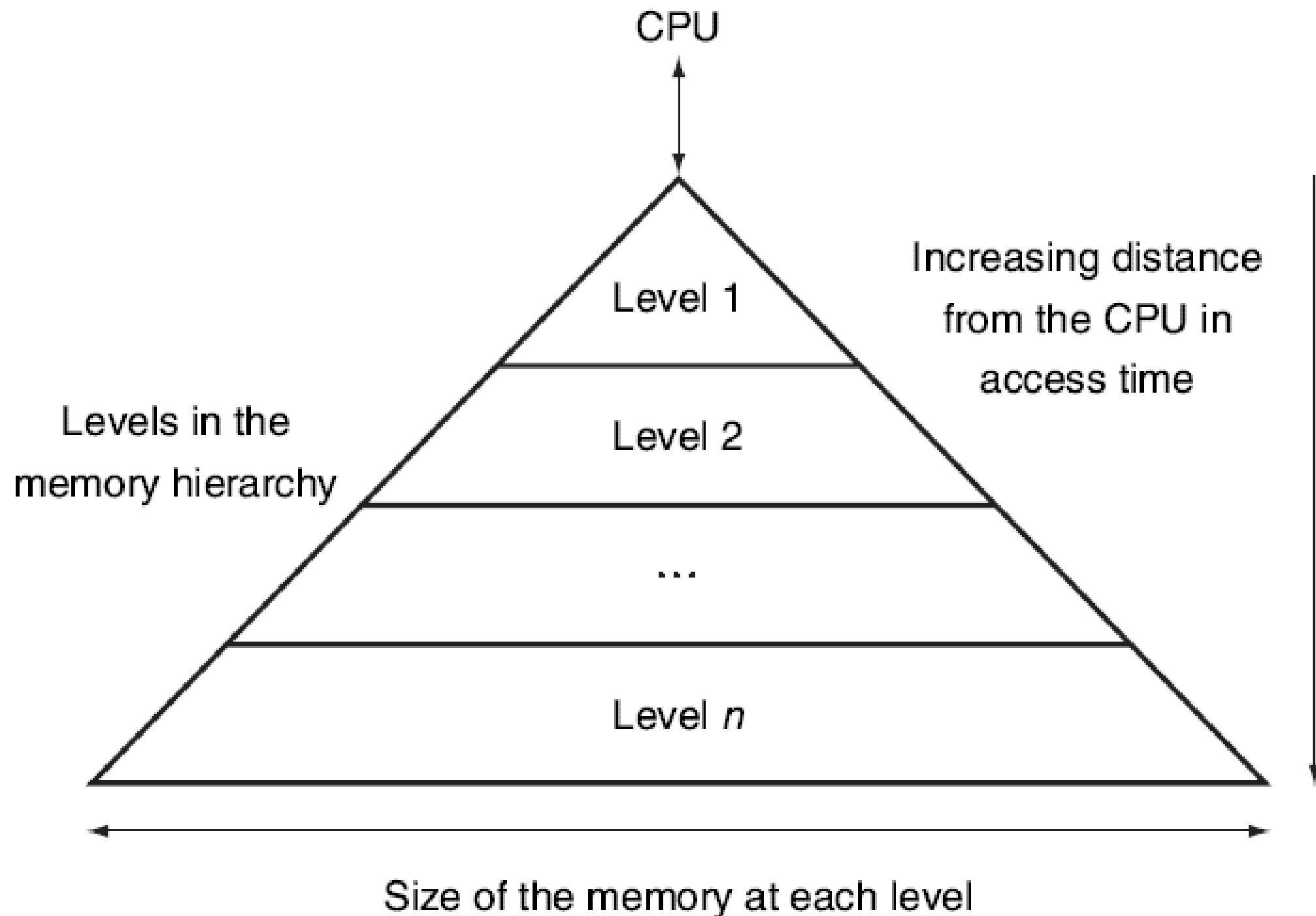
Principio de Localidad – Idea clave

- ▶ El principio de localidad nos da una idea clave:
 - ▶ La información que manejamos en un determinado intervalo de tiempo es acotada
 - ▶ ¡Ideal para memorias rápidas!
- ▶ ¡Las personas funcionamos así!
 - ▶ Cerebro → Notas rápidas → Apuntes → Libros → Bibliotecas
 - ▶ No es transparente (*todavía*).
 - ▶ No hay un sistema automático que detecte nuestra necesidad de información y la incorpore a nuestro cerebro sin que nos demos cuenta.

Pero...

- ▶ Normalmente, las memorias más rápidas (SRAM) son muy caras.
 - ▶ Y por ser caras, son pequeñas.
- ▶ Las memorias más baratas, son lentas.
 - ▶ Y por ser baratas, son grandes.
- ▶ **Objetivo:** brindar al usuario el mayor almacenamiento posible, lo más rápido posible (en promedio), con el menor costo posible. *¿Cómo?*
 - ▶ Mediante otra de las **Grandes Ideas en Arquitectura de Computadoras.**

Jerarquía de Memorias



Jerarquía de Memorias

- ▶ El procesador se comunica solamente con lo más alto de la jerarquía.
- ▶ Los niveles más altos son más rápidos, y a medida que nos alejamos del procesador se van haciendo más lentos.
- ▶ Los niveles más altos son más pequeños, y a medida que nos alejamos del procesador se van haciendo más grandes.
- ▶ El objetivo se cumple si:
 - ▶ **Mayoría de accesos** sea a las memorias **más rápidas**.
 - ▶ **Todo su espacio** de direccionamiento disponible en la memoria **más barata**.
- ▶ *¿Por qué es tan importante la Jerarquía de memoria?*

Tendencias Tecnológicas de DRAM

Year introduced	Chip size	\$ per GiB	Total access time to a new row/column	Average column access time to existing row
1980	64 Kibibit	\$6,480,000	250 ns	150 ns
1983	256 Kibibit	\$1,980,000	185 ns	100 ns
1985	1 Mebibit	\$720,000	135 ns	40 ns
1989	4 Mebibit	\$128,000	110 ns	40 ns
1992	16 Mebibit	\$30,000	90 ns	30 ns
1996	64 Mebibit	\$9,000	60 ns	12 ns
1998	128 Mebibit	\$900	60 ns	10 ns
2000	256 Mebibit	\$840	55 ns	7 ns
2004	512 Mebibit	\$150	50 ns	5 ns
2007	1 Gibibit	\$40	45 ns	1.25 ns
2010	2 Gibibit	\$13	40 ns	1 ns
2012	4 Gibibit	\$5	35 ns	0.8 ns
2015	8 Gibibit	\$7	30 ns	0.6 ns
2018	16 Gibibit	\$6	25 ns	0.4 ns

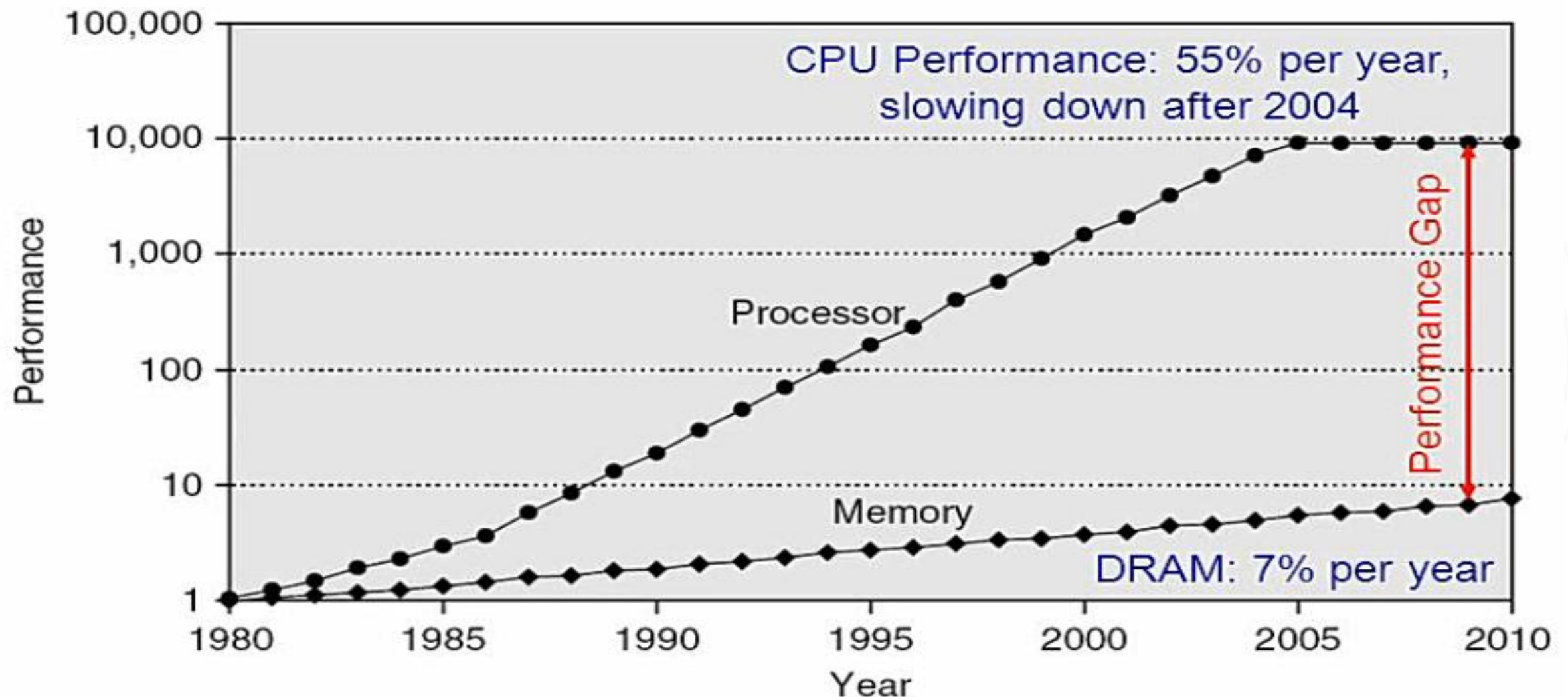
- ▶ Capacidad: aumenta 4x/3años (hasta 1996); aumenta 2x/3 años (desde 1996).
- ▶ Costo/GiB: disminuye continuamente.
- ▶ Tiempo de acceso: disminuye continuamente, pero más lento.

Performance Memoria vs CPU

- ▶ Procesadores duplicaban su performance cada 2 años (impulsados por Ley de Moore).
- ▶ DRAM mejoraba su performance un 40% cada 10 años.
- ▶ Luego, la performance relativa de la memoria con respecto a los procesadores es cada vez menor.

Generación de procesador	T Acceso DRAM	T CPU	ciclos	IPC	Instrucciones
Primera	340 ns	5 ns	68	2	136
Segunda	266 ns	3,3 ns	80	4	320
Tercera	180 ns	1,7 ns	108	6	648

The Memory Wall



- ▶ Durante 20 años la brecha de performance se amplió notablemente, aumentando casi 50% /año.
- ▶ Actualmente, se mantiene.

Impacto en la performance

- ▶ Supongamos nuestro procesador segmentado, con un CPI = 1,1 (por los riesgos) y un 30% de instrucciones que accede a memoria.
 - ▶ Hasta ahora, las memorias eran ideales.
- ▶ Supongamos ahora que la memoria de datos el 10% de las veces no tiene disponible el dato que buscamos, y que traer ese dato “cuesta” 50 ciclos.
 - ▶ Este costo agrega ciclos a nuestro CPI, de manera idéntica a los riesgos.
- ▶ **$CPI = CPI_{ideal} + \text{Paradas promedio por instrucción}$**
 - ▶ $CPI = 1,1 + 0,3 * 0,1 * 50 = 1,1 + 1,5 = 2,6$
- ▶ En este ejemplo simple, el procesador está el **58% del tiempo sin hacer nada, esperando por la memoria.**
- ▶ Si además, la memoria de instrucciones no tiene el dato el 1% de las veces, ¡¡el CPI se incrementa a 3,1!!

Memoria Caché

- ▶ Son los niveles más altos de la Jerarquía de Memoria.
- ▶ Usualmente SRAM, muy rápida y muy costosa.
- ▶ Lo que en los temas anteriores considerábamos como memorias ideales, ahora serán cachés.
 - ▶ Y veremos cómo influye que no sean ideales.
- ▶ No tienen valor por sí mismos.
 - ▶ Sólo sirven para cerrar la brecha de performance entre procesadores y memoria principal.
 - ▶ Ocupan un gran porcentaje del área de procesadores modernos.
 - ▶ Y son responsables de un gran porcentaje del consumo de energía.

Memoria Caché – Idea clave

- ▶ Pero la misma idea es aplicada entre memoria principal y disco.
 - Se denomina Memoria Virtual.
- ▶ También dentro de diversos dispositivos.
 - Y también en programas de Sw.
- ▶ Siempre son transparentes al usuario.
 - Manejados por Hw y por SO.

¿Cómo funciona la Jerarquía?

- ▶ En un principio, almacenar toda la información en la memoria grande y barata.
- ▶ Cuando el procesador necesite un dato, moverlo hasta las memorias más rápidas.
 - ▶ Porque por localidad temporal, es probable que lo volvamos a necesitar.
- ▶ Al moverlo, no movamos solamente el dato solicitado, sino también sus vecinos.
 - ▶ Porque por localidad espacial es probable que también sean accedidos.
- ▶ Independiente de la cantidad de niveles que tenga la jerarquía, los movimientos son siempre entre dos niveles adyacentes.

Jerarquía de Memoria: Terminología

- ▶ **Nivel superior:** el más cercano al procesador.
- ▶ **Bloque:** la mínima cantidad de información que se transfiere entre dos niveles.
- ▶ Cuando un dato solicitado está en algún bloque del nivel superior, se dice que hubo un **acierto** (*hit*).
 - ▶ Se denomina **tasa de aciertos** (*hit rate*) a la fracción de accesos a memoria que se encuentran en el nivel superior.
 - ▶ Se denomina **tiempo de acierto** (*hit time*) al tiempo total en que el nivel superior es capaz de entregar un dato. Consiste en la suma de:
 - ▶ Tiempo de acceso de este nivel.
 - ▶ Tiempo para determinar si es un acierto o no.

Jerarquía de Memoria: Terminología

- ▶ Cuando un dato solicitado no está en el nivel superior, y hay que buscarlo en el nivel inferior (para moverlo), se dice que hubo un **fallo (*miss*)**.
- ▶ Se denomina **tasa de fallos (*miss rate*)** al complemento de la tasa de aciertos.
 - ▶ Tasa de fallos = $1 - \text{Tasa de aciertos}$.
- ▶ Se denomina **penalidad por fallos (*miss penalty*)** al tiempo total para mover un bloque del nivel inferior al superior.
- ▶ Toda la Jerarquía de Memoria se basa en la idea que para un dado nivel, el tiempo de acierto sea mucho menor que la penalidad por fallo.
 - ▶ **Hit time $\ll$ Miss penalty**

Diseño de una Jerarquía de Memorias

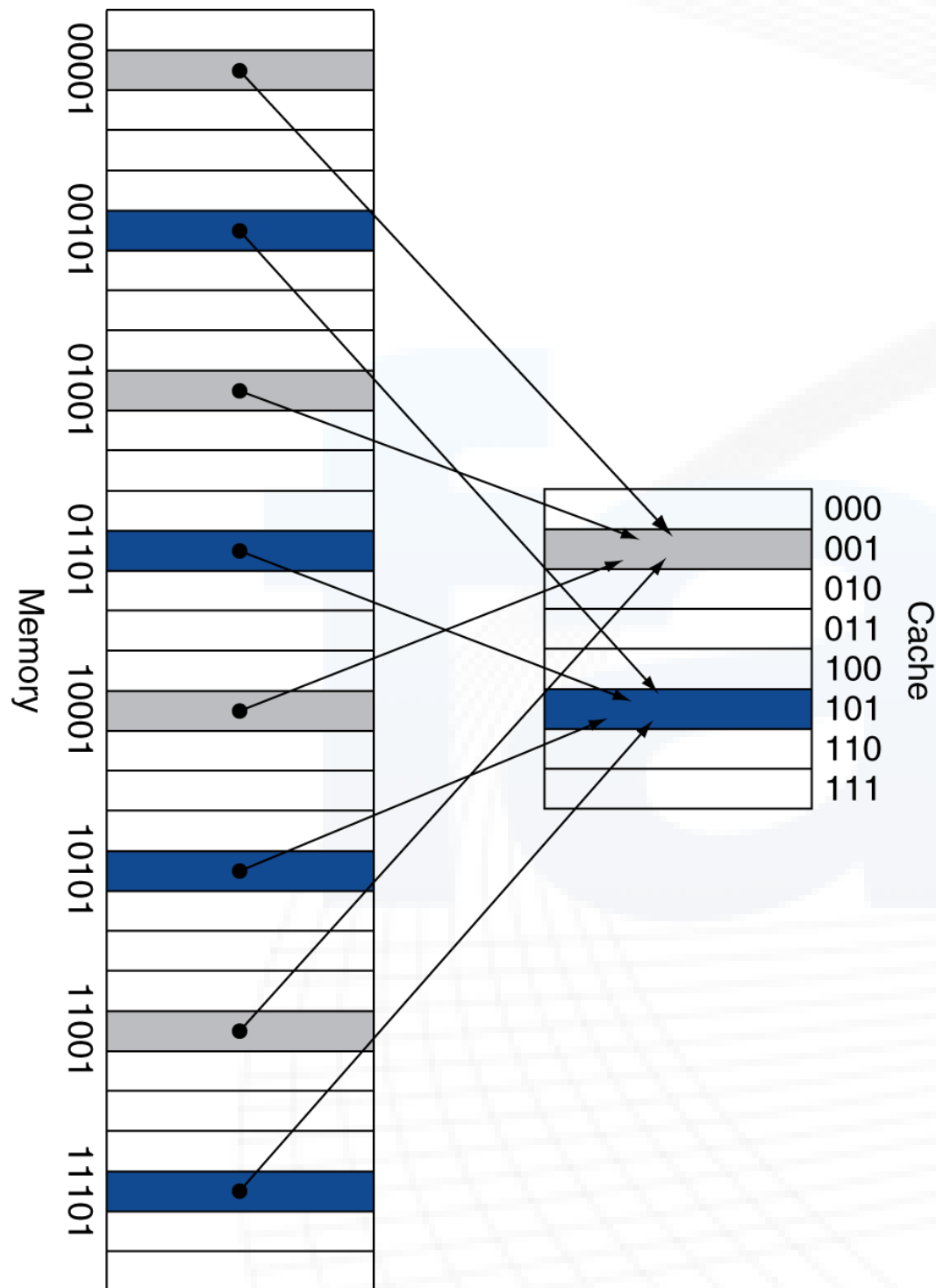
- ▶ Dado un programa de benchmark, se lo ejecuta y se genera un archivo denominado **traza de memoria**.
 - ▶ Consiste en la serie de referencias a memoria realizadas.
 - ▶ Cada referencia a memoria es especificada con un par de valores:
 - ▶ El tipo de acceso: fetch, read, write.
 - ▶ La dirección de acceso.
- ▶ El objetivo es optimizar la Jerarquía para minimizar el tiempo de ejecución del benchmark.

1	0	194	(Fetch)
2	2	33f0	(Read)
3	0	7478	(Fetch)
4	0	747c	(Fetch)
5	0	7480	(Fetch)
6	0	7484	(Fetch)
7	3	9b38	(Write)
8	0	7488	(Fetch)
9	0	748c	(Fetch)
10	0	7490	(Fetch)
11	3	9b3c	(Write)

Cuatro cuestiones básicas de diseño

- ▶ Para cualquier nivel de la Jerarquía de memoria se deben responder las siguientes cuatro preguntas:
 - ▶ ¿Dónde ubico un bloque en el caché?
 - ▶ ¿Cómo encuentro si un bloque está en caché?
 - ▶ ¿Cuál bloque reemplazo en caso de tener un fallo?
 - ▶ ¿Qué se hace en caso de escrituras?
- ▶ Las dos primeras preguntas están relacionadas, y nos dan la pauta que debe existir una forma de “mapear” las direcciones de memoria al caché.

Caché de Mapeo Directo



- ▶ Es la forma más simple de organización.
- ▶ Cada dirección de memoria se mapea directamente a un único bloque del caché.
 - ▶ Se utilizan los bits menos significativos de la dirección, a modo de **índice**.
- ▶ Pero un bloque de caché puede ser ocupado por muchas direcciones de memoria.
 - ▶ *¿Cuál dirección pondríamos?*
 - ▶ *¿Cómo podemos saber cuál de todas está en el caché?*

Etiquetas y bit de Validez

- ▶ Para diferenciar cuál de todas las direcciones de memoria está guardada en el caché, necesitamos almacenar adicionalmente el resto de la dirección, a modo de **identificador único**. Esto se denomina **etiqueta (tag)**.
 - ▶ Hay una etiqueta por bloque de caché.
- ▶ Suponiendo direcciones de memoria de 32 bits, y un caché de mapeo directo de 2^N bytes:
 - ▶ Los N bits menos significativos se usan para el índice.
 - ▶ Los (32-N) bits más significativos se usan para la etiqueta.
- ▶ Como puede ocurrir que en un bloque de caché no haya ningún dato, es necesario también un bit que indica la validez de ese bloque.
- ▶ Ambos son considerados como parte del “estado” de un caché.

Ejemplo de acceso a un caché (1/8)

- ▶ Supongamos un caché de mapeo directo de 8 bloques.
 - ▶ Cada bloque almacena 1 byte de memoria.
 - ▶ **¿Cuántos bits necesitamos para el índice?**
- ▶ Inicialmente, al recibir energía, el caché tiene todos sus bloques vacíos.
 - ▶ El bit de validez de cada bloque no está activo.
- ▶ Accederemos a la siguiente secuencia de ocho direcciones de memoria (en decimal):
 - ▶ 22, 26, 22, 26, 16, 3, 16, 18.
 - ▶ Siempre lecturas.
 - ▶ En binario, necesitaremos 5 bits para estas direcciones.

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Ejemplo de acceso a un caché (2/8)

- ▶ Primer acceso: dirección de memoria 22. *¿Es un acierto o un fallo?*
- ▶ Pasar la dirección de memoria a binario: $22_{10} = 10110_2$.
- ▶ Descomponer la dirección en etiqueta e índice, para saber dónde se ubica.
 - ▶ El índice son los 3 bits menos significativos: 110_2 . El resto de la dirección se usa como etiqueta: 10_2 .
- ▶ Como el bit de validez de ese bloque no está activo, entonces el acceso es un **fallo**.
- ▶ Se trae el contenido de la dirección de memoria desde el nivel inferior de la jerarquía, se lo almacena en el bloque correspondiente, y se actualizan la etiqueta y el bit de validez del bloque.

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Ejemplo de acceso a un caché (3/8)

- ▶ Segundo acceso: dirección de memoria 26. ¿Es un acierto o un fallo?
- ▶ Pasar la dirección de memoria a binario: $26_{10} = 11010_2$.
- ▶ Descomponer la dirección en etiqueta e índice, para saber dónde se ubica.
 - ▶ El índice son los 3 bits menos significativos: 010_2 . El resto de la dirección se usa como etiqueta: 11_2 .
- ▶ Como el bit de validez de ese bloque no está activo, entonces este acceso también es un **fallo**.
- ▶ Se trae el contenido de la dirección de memoria desde el nivel inferior de la jerarquía, se lo almacena en el bloque correspondiente, y se actualizan la etiqueta y el bit de validez del bloque.

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Ejemplo de acceso a un caché (4/8)

- ▶ Tercer acceso: dirección de memoria 22. ¿Es un acierto o un fallo?
- ▶ Pasar la dirección de memoria a binario: $22_{10} = 10110_2$.
- ▶ El índice son los 3 bits menos significativos: 110_2 . El resto de la dirección se usa como etiqueta: 10_2 .
- ▶ Como el bit de validez de ese bloque sí está activo, entonces se compara la etiqueta de la dirección con la etiqueta almacenada en el bloque.
- ▶ Como las etiquetas son iguales, este acceso es un **acierto**.
- ▶ El contenido del caché permanece idéntico.
- ▶ Lo mismo ocurre con el cuarto acceso, a la dirección 26, que también resulta en un **acierto**.
- ▶ Se empieza a notar la Localidad temporal.

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Ejemplo de acceso a un caché (5/8)

- ▶ Quinto acceso: dirección de memoria **16**. *¿Es un acierto o un fallo?*
- ▶ Pasar la dirección de memoria a binario: $16_{10} = 10000_2$.
- ▶ El índice son los 3 bits menos significativos: 000_2 . El resto de la dirección se usa como etiqueta: 10_2 .
- ▶ Como el bit de validez de ese bloque no está activo, entonces este acceso también es un **fallo**.
- ▶ Se trae el contenido de la dirección de memoria desde el nivel inferior de la jerarquía, se lo almacena en el bloque correspondiente, y se actualizan la etiqueta y el bit de validez del bloque.

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Ejemplo de acceso a un caché (6/8)

- ▶ Sexto acceso: dirección de memoria 3. *¿Es un acierto o un fallo?*
- ▶ Pasar la dirección de memoria a binario: $3_{10} = 00011_2$.
- ▶ El índice son los 3 bits menos significativos: 011_2 . El resto de la dirección se usa como etiqueta: 00_2 .
- ▶ Como el bit de validez de ese bloque no está activo, entonces este acceso también es un **fallo**.
- ▶ Se trae el contenido de la dirección de memoria desde el nivel inferior de la jerarquía, se lo almacena en el bloque correspondiente, y se actualizan la etiqueta y el bit de validez del bloque.

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Ejemplo de acceso a un caché (7/8)

- ▶ Séptimo acceso: dirección de memoria 16. Repitiendo el análisis, es un **acierto**.
- ▶ Octavo acceso: dirección de memoria 18. En binario: $18_{10} = 10010_2$.
- ▶ El índice son los 3 bits menos significativos: 010_2 . La etiqueta es: 10_2 .
- ▶ Como el bit de validez de ese bloque está activo, se compara la etiqueta de la dirección con la etiqueta almacenada en el bloque.
- ▶ Como las etiquetas son distintas, este acceso también es un **fallo**.
- ▶ Ese bloque tenía el contenido de la dirección de memoria 26, que será descartado.
- ▶ Se trae el nuevo contenido desde el nivel inferior de la jerarquía, se lo almacena en el bloque, y se actualizan la etiqueta y el bit de validez.

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Ejemplo de acceso a un caché (8/8)

- ▶ Accedimos consecutivamente a las palabras 22, 26, 22, 26, 16, 3, 16, 18.
- ▶ La mayoría fueron fallos.
 - ▶ Muchos fallos al comienzo.
 - ▶ Poca localidad temporal en los accesos.
 - ▶ *¿Y localidad espacial?*
- ▶ Para colocar la palabra 18 hubo que quitar la 26.
 - ▶ A la 26 se la trajo al caché, pero si se la accediese nuevamente, sería un fallo.
 - ▶ Posibilidad que varias direcciones de memoria “compitan” por el mismo bloque de caché.
- ▶ Se hicieron varias simplificaciones.
- ▶ Veremos cómo mejorar cada uno de estos puntos.

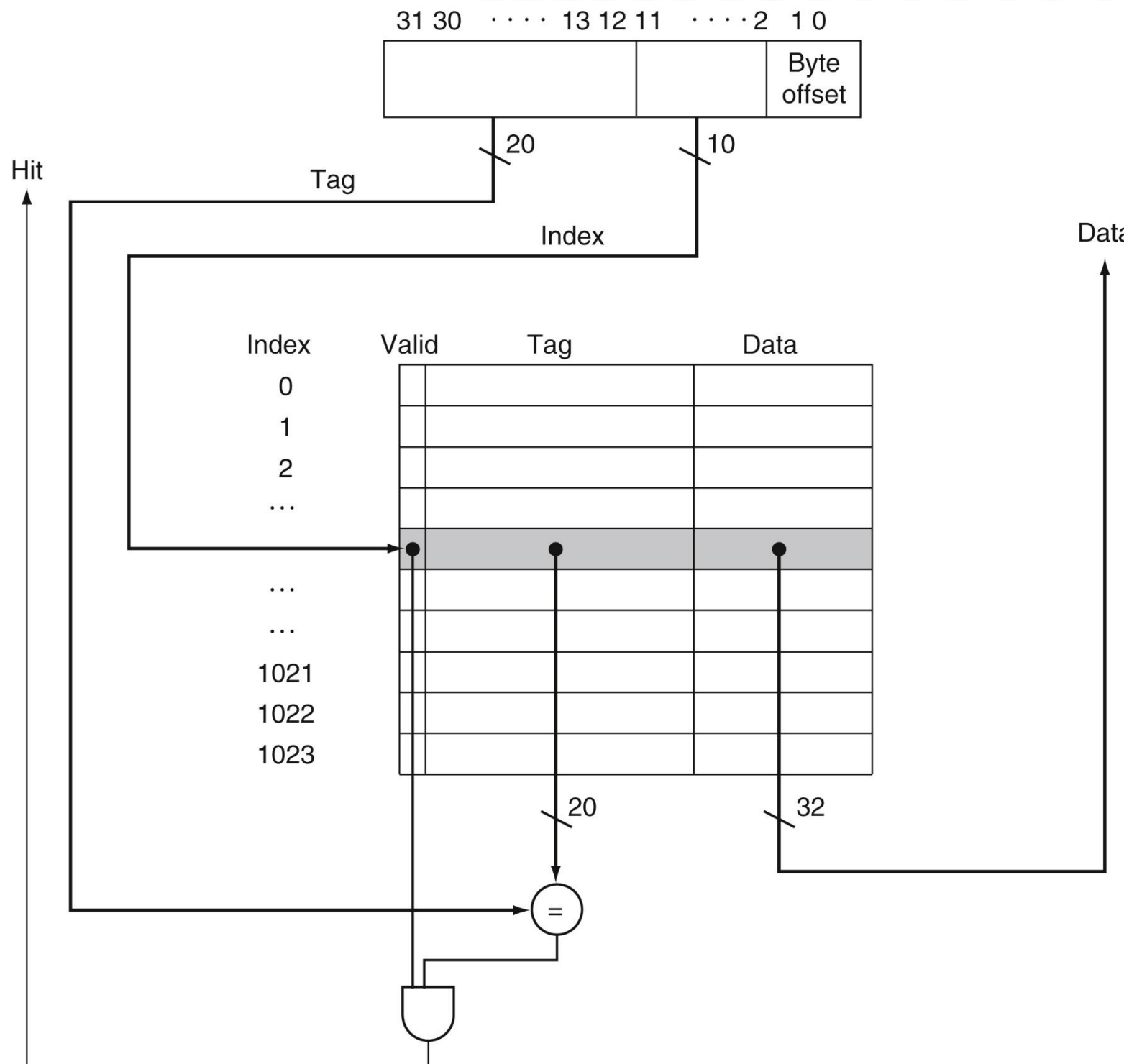
Ejemplo un poco más real (1/3)

- ▶ Consideramos 5 bits para las direcciones, pero normalmente son de 32 bits.
 - ▶ Esto produce que las etiquetas sean más grandes.
- ▶ Consideramos que los bloques almacenan 1 byte, pero normalmente almacenan palabras.
 - ▶ Aunque la memoria es direccionable de a bytes, las direcciones de memoria normalmente buscan traer una palabra.
 - ▶ Las palabras normalmente son de 32 bits (4 Bytes), pero pueden ser de otros tamaños.
 - ▶ Esto produce que los bits menos significativos de la dirección de memoria sean considerados como **offset de byte**.
 - ▶ Y el índice pasan a ser los bits menos significativos, sin contar este offset de byte.
- ▶ Consideramos un caché de 8 bloques, pero normalmente poseen más bloques.

Ejemplo un poco más real (2/3)

- ▶ Supongamos un caché de 1024 bloques de 4 Bytes, accedido por direcciones de 32 bits.
 - ▶ **¿Qué tamaño tiene el caché?**
 - ▶ Cant de bloques * Bytes/bloque
 - ▶ $1024 * 4 \text{ B} = 4 \text{ KiB}$
 - ▶ **¿Cuántos bits tiene el índice?**
 - ▶ $\log_2$ Cant de bloques
 - ▶ $\log_2 1024 = 10 \text{ bits}$
 - ▶ **¿Cuántos bits tiene la etiqueta?**
 - ▶ Tam etiqueta = Tam dirección – tam índice – byte offset
 - ▶ $32 - 10 - 2 = 20 \text{ bits}$
 - ▶ **¿Cómo se lo accede y se determina si es un acierto?**

Ejemplo un poco más real (3/3)



- ▶ Dada una dirección, se la descompone y se obtiene el índice.
- ▶ Con el índice, se selecciona un bloque del caché.
- ▶ Se compara la etiqueta almacenada en el bloque con la que viene en la dirección.
- ▶ Si las etiquetas son iguales, y el bit de validez está activo, se tiene un acierto, y se transfiere el dato del bloque al procesador.

Bloques de caché más grandes

- ▶ En el ejemplo anterior, cada bloque de caché contenía una sola palabra, de 4 bytes.
 - ▶ No aprovechamos localidad espacial.
- ▶ Conviene que cada bloque almacene varias palabras.
 - ▶ Siempre con **una única etiqueta por bloque**.
- ▶ Ahora la dirección se descompone en tres campos:
 - ▶ Los m bits menos significativos para indicar el **offset** de la palabra dentro del bloque.
 - ▶ Los siguientes n bits para determinar el índice del caché.
 - ▶ Todos los bits restantes componen la etiqueta.

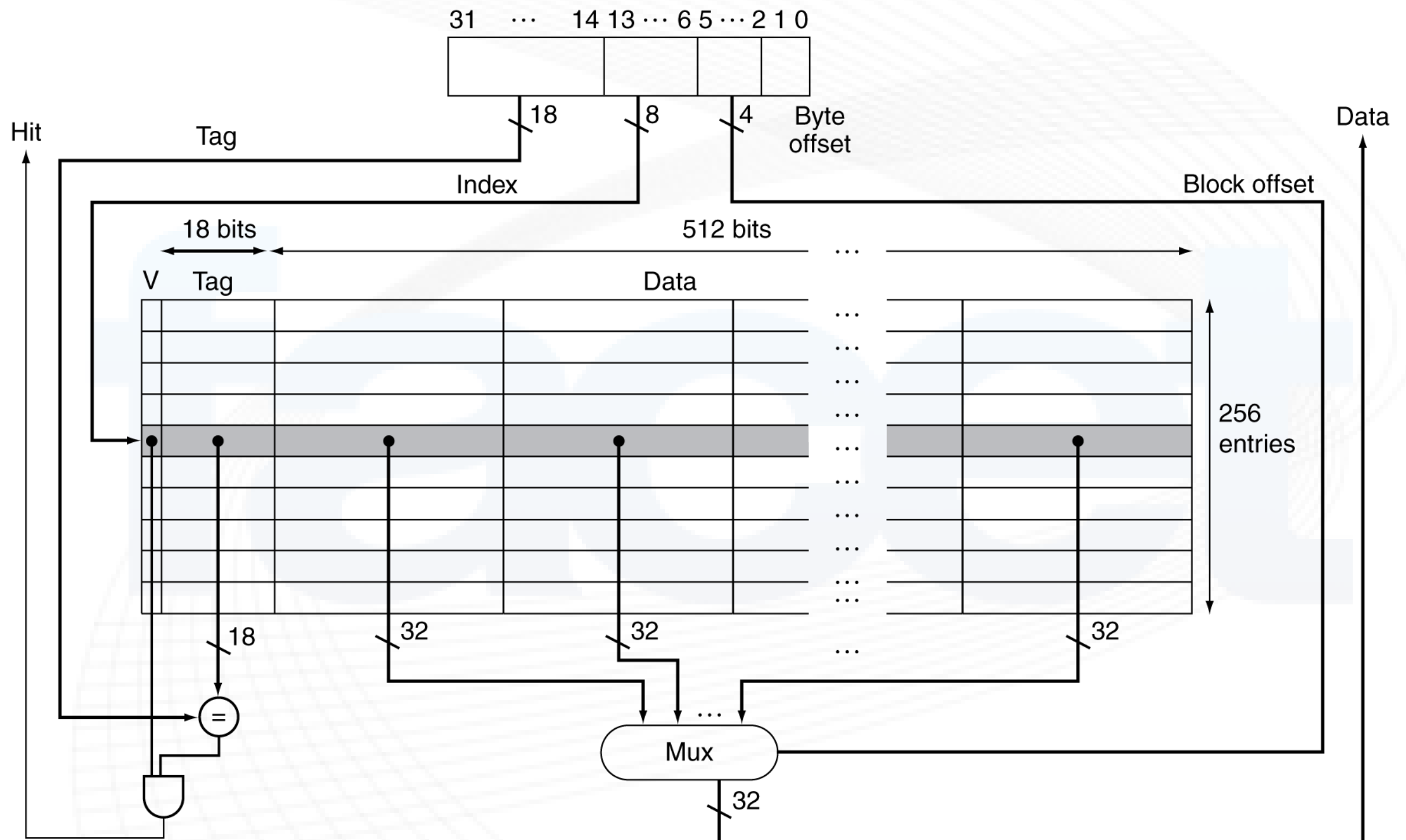
Offset de bloque vs Offset de byte

- ▶ Dentro del offset se debe incluir el offset de byte, para permitir direccionar a los bytes individuales de una palabra.
 - ▶ $\text{Offset} = \text{offset de bloque} + \text{offset de byte}$
- ▶ Ej: un bloque de 16 bytes necesita 4 bits de offset, y puede contener:
 - ▶ 16 palabras de 1 byte (4+0).
 - ▶ 8 palabras de 2 bytes (3+1).
 - ▶ 4 palabras de 4 bytes (2+2).
 - ▶ 2 palabras de 8 bytes (1+3).
 - ▶ 1 palabra de 16 bytes (0+4)

Ejemplo caché MD con bloques grandes

- ▶ Supongamos un caché de mapeo directo de 256 bloques de 16 palabras, de 4 Bytes cada una, accedido por direcciones de 32 bits.
 - ▶ *¿Qué tamaño tiene el bloque?*
 - ▶ *¿Qué tamaño tiene el caché?*
 - ▶ *¿Cuántos bits necesitamos para el offset de bloque?*
 - ▶ *¿Cuántos bits tiene el índice?*
 - ▶ *¿Cuántos bits tiene la etiqueta?*
 - ▶ *¿Cómo se lo accede y se determina si es un acierto?*

Ejemplo caché MD con bloques grandes



Ejemplo caché MD con bloques grandes

- ▶ *¿A cuál bloque de caché se mapea la dirección de memoria 1200?*
 - ▶ Ya no podemos usar la fórmula (dirección mod #bloques), porque varias direcciones se mapean al mismo bloque.
- ▶ Hay que averiguar primero a qué bloque de memoria corresponde la dirección:
 - ▶ Dirección del bloque = dirección de memoria div tamaño del bloque
 - ▶ Dirección del bloque = $1200 \text{ div } 64 = 18$
 - ▶ El bloque 18 de memoria contendrá todas las direcciones comprendidas entre 1152 y 1215.
 - ▶ Se puede hacer el mismo cálculo utilizando palabras:
 - ▶ $300 \text{ div } 16 = 18$ (el bloque contiene las palabras 288 a 303).
- ▶ Luego determinar a cuál bloque de caché se corresponde, con la misma fórmula anterior:
 - ▶ $18 \text{ mod } 256 = 18$

Ejemplo caché MD con bloques grandes

- ▶ *¿Qué cambia si mantenemos el tamaño total, pero aumentando el tamaño del bloque a 32 palabras?*
- ▶ Como el bloque es más grande, se agranda el tamaño del offset.
- ▶ Como el tamaño total se mantiene constante, hay menos bloques.
 - ▶ Entonces se achica el tamaño del índice.
 - ▶ Pero la suma de ambos (índice + offset) se mantiene constante.

Consideraciones sobre el tamaño del bloque

- ▶ Bloques más grandes aprovechan mejor la localidad espacial.
 - ▶ Lo que reduce la tasa de fallos.
- ▶ Pero al aumentar el tamaño de los bloques, manteniendo fijo el tamaño total, disminuye la cantidad de bloques.
 - ▶ Hay menos localidad temporal y aumenta la tasa de fallos.
- ▶ Por lo tanto, hay un **compromiso entre el tamaño del bloque y la tasa de fallos**.
- ▶ Además, bloques más grandes aumentan la penalidad por fallos.
 - ▶ Hay que traer más datos desde el nivel inferior.
 - ▶ Tener menos fallos con alta penalidad puede ser contraproducente.
 - ▶ Da una idea que estas dos variables están relacionadas.

Inconveniente: “competencias”

- ▶ En un ejemplo anterior vimos que es probable que dos direcciones de memoria diferentes “compitan” por un mismo bloque de caché.
 - ▶ Las palabras 18 y 26 se mapeaban al mismo bloque de caché.
- ▶ Efecto Ping Pong
 - ▶ Se cargan continuamente datos en el caché, para ser descartados antes de ser usados nuevamente.
- ▶ Este efecto ocasiona **fallos por conflicto**.
 - ▶ Es común en cachés de mapeo directo, y también en cachés de tamaño reducido.
 - ▶ Dos posibles alternativas de solución:
 - ▶ Aumentar el tamaño del caché (cambia función de mapeo).
 - ▶ Cambiar la organización del caché: Asociatividad.

Caché Asociativo puro

- ▶ Otra organización de caché.
- ▶ No se basa en un índice
 - ▶ Un bloque de memoria puede ubicarse **en cualquier bloque del caché**.
 - ▶ Se comparan las etiquetas de todos los bloques al mismo tiempo para ver si hay un acierto.
 - ▶ Requiere un comparador por cada bloque de caché. Más costoso.
 - ▶ La dirección de memoria se descompone solamente en dos campos: etiqueta y offset.
 - ▶ Las etiquetas son más grandes, más bits para almacenar.
- ▶ Por definición, ¡no posee fallos por conflicto!
 - ▶ Tiene menor tasa de fallos que Mapeo Directo.
- ▶ Tiempo de acierto mayor que en Mapeo Directo.
 - ▶ Porque en Mapeo Directo puedo acceder al bloque mientras evalúo si es un acierto o no.

Caché Asociativo por Conjuntos de N vías

- ▶ Tercera organización de caché.
 - ▶ Combina las ventajas de las dos anteriores.
- ▶ El índice no selecciona un único bloque de caché, sino **un conjunto de N bloques**.
 - ▶ Dentro de este conjunto, los bloques son asociativos.
 - ▶ Dado un valor de índice, el dato de memoria puede ir a cualquier bloque de ese conjunto de bloques.
 - ▶ Analogía: N cachés de mapeo directo en paralelo.
- ▶ Más costoso que Mapeo Directo, pero menos que Asociativo Puro.
- ▶ Menor tasa de fallos que Mapeo Directo, pero más que Asociativo Puro.

Ejemplo organizaciones de caché

**One-way set associative
(direct mapped)**

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

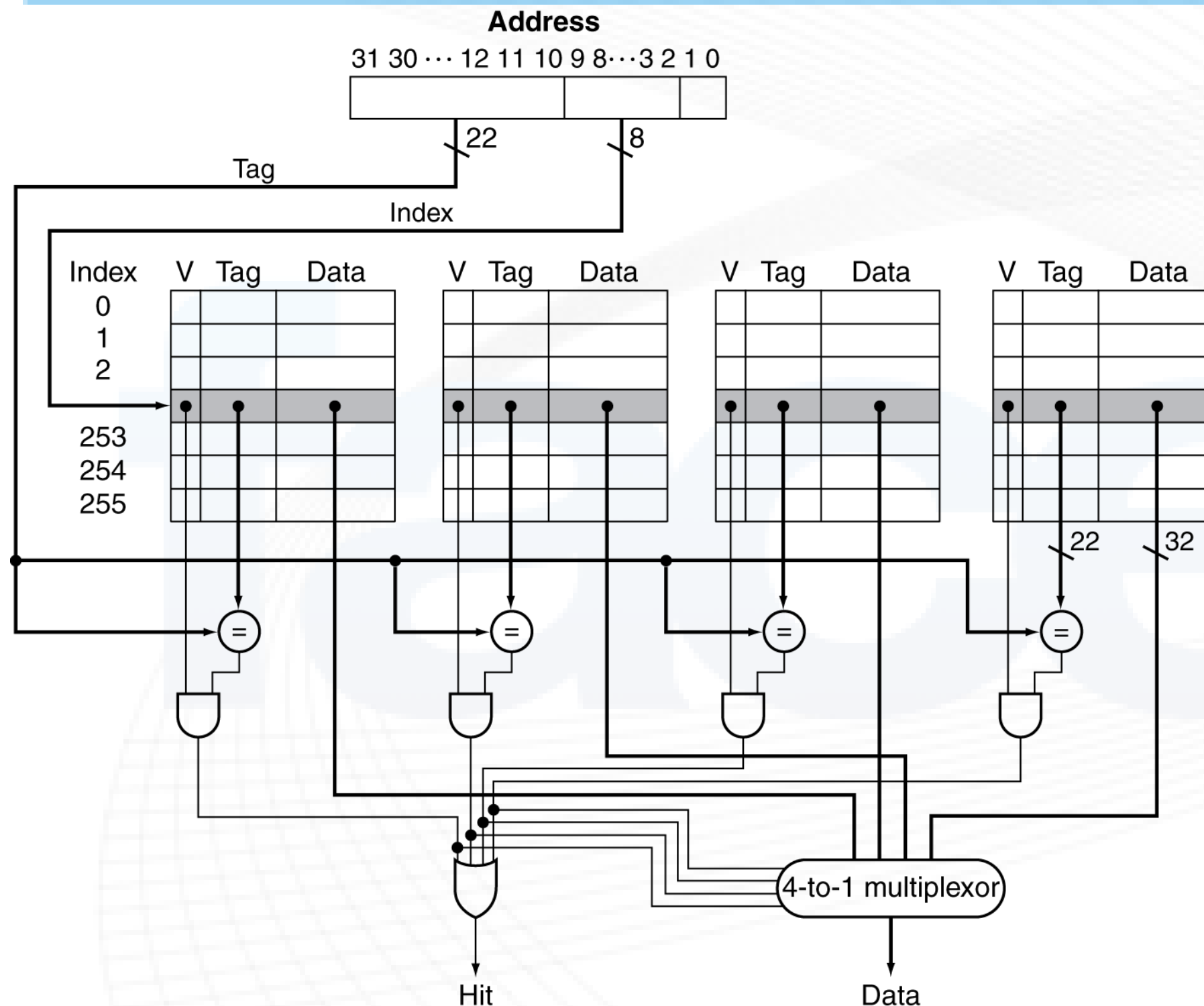
Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

► Dados los 4 cachés posibles de 8 bloques de una palabra: *¿dónde puede ubicarse la palabra 12?*

Ejemplo Asociativo por conjuntos de 4 vías



► *¿Qué tamaño tiene este caché?*

Consideraciones sobre Asociatividad

- ▶ Aumentar la asociatividad disminuye la tasa de fallos.
 - ▶ Pero con rendimientos decrecientes.
 - ▶ En un caché de datos de 64 KiB, con bloques de 16 palabras, ejecutando SPEC2000 se obtiene:

Associativity	Data miss rate
1	10.3%
2	8.6%
4	8.3%
8	8.1%

- ▶ Aumentar la asociatividad aumenta el tamaño y el costo del caché.
- ▶ Aumentar la asociatividad aumenta el tiempo de acierto.

Políticas de Reemplazo

- ▶ Una pregunta básica de diseño era:
 - ▶ ¿Cuál bloque reemplazo en caso de tener un fallo?
 - ▶ En cachés de Mapeo Directo, la respuesta es trivial.
- ▶ Pero en cachés Asociativos Puro o por Conjuntos, hace necesario definir **políticas de reemplazo de bloques**.
 - ▶ Política más simple: reemplazo al azar (random).
 - ▶ Política ideal: reemplazar el que no vaya a usar próximamente.
 - ▶ Imposible de implementar, porque es difícil predecir el futuro.
 - ▶ Política alternativa: reemplazar el menos recientemente usado (LRU, *Least Recently Used*).

Consideraciones sobre Políticas de Reemplazo

- ▶ Al aumentar la asociatividad, LRU se vuelve más complejo.
 - ▶ LRU es el más usado en Asociativo por conjuntos de 2 o 4 vías.
- ▶ Random es más simple, e independiente de la asociatividad.
 - ▶ En conjuntos de 2 o 4 vías, random genera una tasa de fallos 1,1 veces mayor que LRU.
- ▶ Al aumentar la asociatividad, la diferencia relativa entre las tasas de fallo es cada vez menor.
- ▶ Al aumentar el tamaño del caché, la tasa de fallos disminuye para ambas políticas, y disminuye también la diferencia relativa entre ambas.

Políticas de Escritura

- ▶ Nos falta analizar una de las cuatro preguntas básicas: ¿Qué se hace en caso de escrituras?
- ▶ Es mucho más simple manejar las lecturas de memoria, porque no modifican valores.
 - ▶ Un caché de instrucciones es mucho más simple que uno de datos.
- ▶ El problema de las escrituras es cómo se mantienen los datos consistentes entre memoria principal y el caché.
- ▶ Dos alternativas:
 - ▶ Write-Through (escritura “a través”).
 - ▶ Write-Back (escritura “diferida”).

Política de escritura *Write-Through*

- ▶ Idea: al escribir en caché, hacerlo simultáneamente en memoria principal.
 - ▶ Pero...¡la memoria es muy lenta!
 - ▶ Si $CPI_{ideal} = 1$, un acceso a memoria lleva 50 ciclos, y un 10% de las instrucciones son stores, $CPI_{real} = 6$.
- ▶ Se usa un **buffer de escritura**, entre el caché y memoria principal.
 - ▶ El procesador escribe los datos en el caché y en el buffer, y continúa su trabajo.
 - ▶ El buffer vuelca el contenido a memoria cuando puede.
 - ▶ Es una estructura FIFO, típicamente de 4 entradas.
 - ▶ Funciona bien si la frecuencia de escrituras (con respecto al tiempo) no es alta.
 - ▶ Freq máxima = La inversa del tiempo de ciclo de la DRAM.
 - ▶ Si se supera esta frecuencia, el buffer se satura y provoca paradas.

Política de escritura *Write-Back*

- ▶ Idea: escribir sólo en el caché, y marcar el bloque como modificado.
 - ▶ Copiarlo a memoria únicamente cuando esté por ser reemplazado.
 - ▶ Necesita un bit de estado adicional por cada bloque (*dirty bit*).
- ▶ Ventajas:
 - ▶ Reduce mucho el ancho de banda requerido a memoria.
 - ▶ Permite “unificar” varias escrituras en un único acceso a memoria.
 - ▶ Hacen que sea la más usada en la actualidad.
- ▶ Desventaja: aumenta la penalidad por fallo.
 - ▶ Porque ahora no solamente hay que traer un bloque, sino quizás también escribir el bloque a reemplazar si fue modificado.
 - ▶ Se puede “reducir” este problema, y lo veremos en el Tema siguiente.

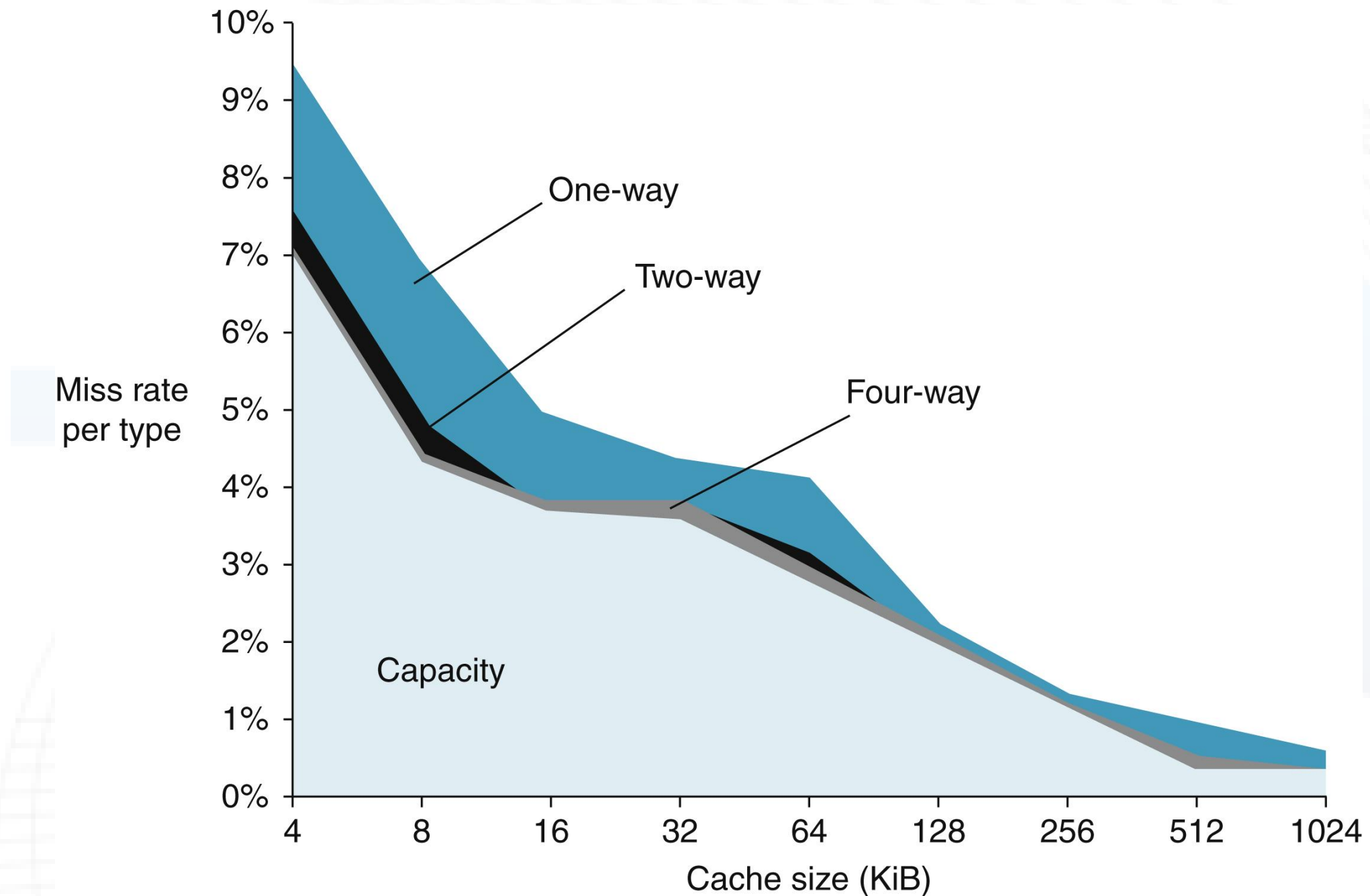
Políticas de Fallo de Escrituras

- ▶ *¿Qué ocurre si al intentar escribir un dato en el caché, el bloque no está?*
 - ▶ ¿Se trae el bloque para escribirlo?
- ▶ En cachés de tipo *Write-Back*, la respuesta suele ser afirmativa.
 - ▶ Se denomina *write allocate*.
- ▶ En cachés de tipo *Write-Through*, la respuesta suele ser negativa.
 - ▶ Porque implica dos accesos a memoria consecutivos.
 - ▶ Directamente, se escribe sólo en el buffer.
 - ▶ Se denomina *write not allocate*.

“Las 3 C”: modelo para evaluar los fallos

- ▶ **Inevitables (Compulsory):** cuando ocurre el primer acceso a un dato o instrucción.
 - ▶ Ojo: no confundir con un caché vacío.
 - ▶ No dependen de la organización, sino del programa a ejecutar.
 - ▶ Disminuyen al incrementar el tamaño del bloque (localidad espacial).
- ▶ **Conflicto:** cuando múltiples direcciones de memoria se mapean al mismo bloque de caché.
 - ▶ Solución 1: incrementar el tamaño del caché.
 - ▶ Solución 2: incrementar la asociatividad.
- ▶ **Capacidad:** cuando el caché no puede contener todos los datos necesarios para el programa.
 - ▶ Solución: incrementar el tamaño del caché.
- ▶ Hay una cuarta “C”, **Coherencia**, que veremos en el Tema siguiente.

Visualizando las 3 C



- Los fallos inevitables representan el 0,006% y no se aprecian en el gráfico.

Consideraciones sobre las 3 C

- ▶ Son un modelo empírico, no teórico.
 - ▶ No explican un fallo en particular, sino un comportamiento general.
 - ▶ Intentan explicar por qué un dato no está en el caché:
 - ▶ Porque nunca estuvo. Porque es la primera vez que se lo necesita.
 - ▶ Porque fue reemplazado porque tuvo un conflicto.
 - ▶ Porque fue reemplazado porque no había más lugar disponible.
- ▶ Incrementar el tamaño del caché disminuye fallos por capacidad.
 - ▶ Quizás también por conflicto, porque cambia la función de mapeo.
 - ▶ Pero incrementa tiempo de acierto, costo y consumo de energía.
- ▶ Aumentar asociatividad disminuye fallos por conflicto.
 - ▶ También incrementa tiempo de acierto, costo y consumo de energía.
- ▶ Aumentar tamaño del bloque disminuye fallos inevitables.
 - ▶ Pero aumenta la penalidad por fallos.

Resumen final

- ▶ Las memorias caché son una parte fundamental de la jerarquía de memorias.
 - ▶ Basados en el principio de localidad.
 - ▶ Sirven para cerrar la brecha de performance entre los procesadores y la memoria principal (*Memory Wall*).
- ▶ Terminología:
 - ▶ Traza de memoria, bloque.
 - ▶ Hit, Miss, Hit Time, Miss rate, Miss Penalty
 - ▶ Índice, etiqueta, offset (de bloque y de byte), bit de validez, bit de ensuciado.
 - ▶ Fallos: inevitables, por conflicto, por capacidad (las tres C).

Resumen final

- ▶ Variables de diseño de cachés:
 - ▶ Tamaño total, Tamaño del bloque, Asociatividad.
 - ▶ Políticas de reemplazo de bloques: LRU, random.
 - ▶ Políticas de escritura: Write-Through y Write-Back.
- ▶ Las distintas variables de diseño interactúan entre sí, generando múltiples compromisos.
 - ▶ Elección óptima dependerá de la traza de memoria.