

JERARQUÍA DE MEMORIA

1 Introducción

Este capítulo está dedicado al análisis de los distintos niveles de memoria existentes en un computador, con especial énfasis en los sistemas de “cache”.

2 Justificación Tecnológica

Históricamente siempre existió un compromiso de diseño en los sistemas de memoria: las memorias rápidas son más costosas que las memorias más lentas. Por tanto un diseño equilibrado es aquel que combina en su justa medida memorias de un tipo y de otro. Este diseño debe, además, lograr la mejor relación entre el coste del sistema y su rendimiento (capacidad de proceso).

La solución a este problema es el concepto de *jerarquía de memoria*. Esta jerarquía básicamente establece un orden en función de la capacidad de memoria y su velocidad. La mayor jerarquía la tiene la memoria más rápida (y por tanto menor en tamaño, para mantener el coste acotado) y la menor la memoria más lenta (y por tanto más económica) pero por tanto más abundante.

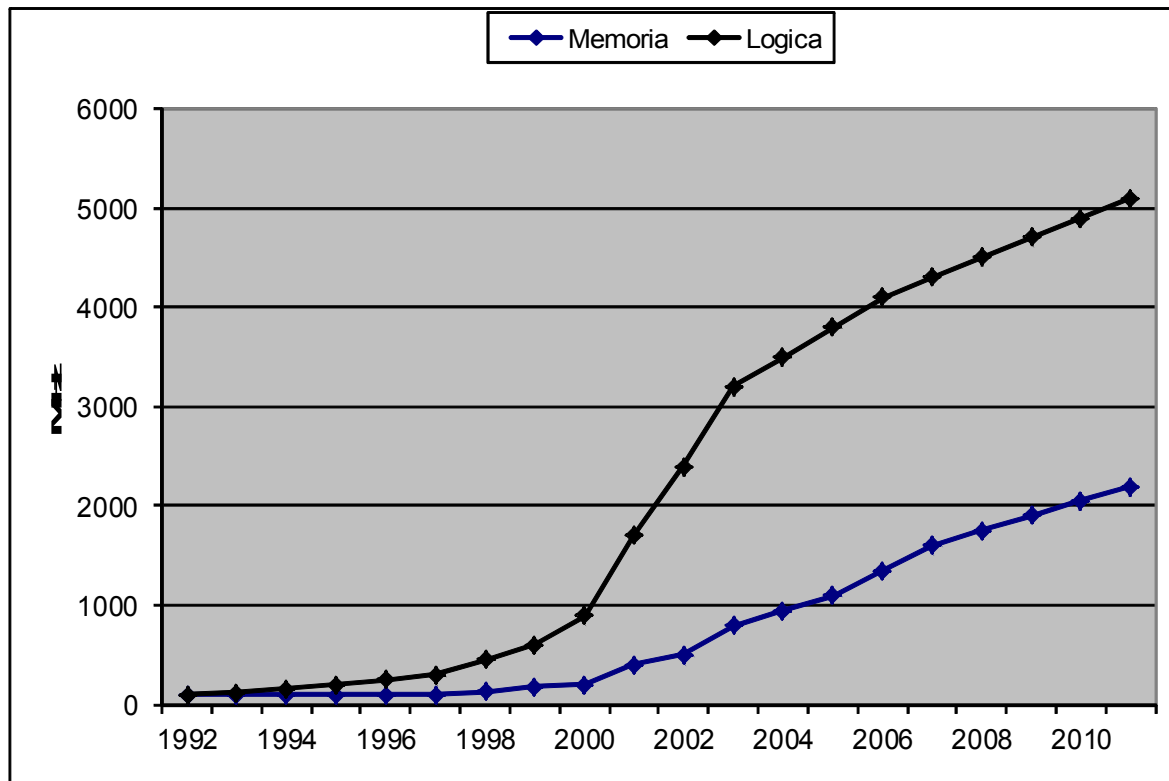
Velocidad		Tamaño (Bytes)
1ns	Registros	100s
1ns	Cache L1	Ks
10ns	Cache L2	Ms
100ns	Memoria Principal (RAM)	Gs
10ms	Memoria Secundaria (Disco)	100Gs
10s	Memoria Terciaria (Cinta)	Ts

Cuanto más descendemos en la jerarquía de memoria, más económica (y por tanto más abundante) es la memoria. Por ejemplo en la actualidad una memoria RAM de 4 GBytes cuesta unos U\$S 40, es decir unos 10 U\$S / GByte. Un disco de 500 GBytes cuesta del orden de U\$S 100, es decir unos 0,2 U\$S / GByte. Aquí vemos que en materia de precio por capacidad de almacenamiento el disco le lleva una ventaja de 50 a 1 a la memoria RAM. Si lo miramos velocidad de acceso el disco es 100000 veces más lento.

Por su lado un chip de memoria SRAM (construido con tecnología de flip-flops) de 4

Mb cuesta unos U\$S 8, mientras que un chip DRAM (construido con tecnología de memorias dinámicas) de 2 Gb sale U\$S 1. Esto significa que la memoria SRAM cuesta 4000 veces más que la memoria DRAM !!. Esto lleva a la inviabilidad económica de construir computadoras con memoria principal basada en tecnología SRAM.

Por otra parte en la década de los 90 del siglo pasado las tecnologías utilizadas para implementar tanto la lógica de los procesadores como la de las memorias era bastante similar en materia de la frecuencia de trabajo. Sin embargo la frecuencia de trabajo de la lógica empezó a aumentar a un ritmo mayor que la de la memoria (en particular los circuitos de memoria utilizados para la memoria RAM principal).



Si bien en la última década la tecnología de memoria logró, por lo menos, no seguir perdiendo terreno, esto se debió básicamente a una ralentización del crecimiento de la frecuencia de trabajo de la lógica de los procesadores, motivada por ciertas barreras físicas difíciles de pasar con la tecnología de fabricación de circuitos integrados actualmente utilizada.

Este fenómeno de diferencia notoria en las frecuencias de trabajo fue la que llevó a la necesidad de mejorar y ampliar el uso de las memorias cache (tipo de memoria que desarrollaremos más adelante) y de hecho se pasó de una sola memoria de este tipo a tener dos (L1 y L2) y hasta tres niveles (L1, L2 y L3), como forma de mejorar el rendimiento de las computadoras sin elevar excesivamente el costo.

3 Principio de Localidad

La organización jerárquica de la memoria se basa en una característica que poseen la mayoría de los programas (al menos dentro de ciertos límites). Esta propiedad se denomina *principio de localidad*.

El principio de localidad establece que los programas acceden a una porción relativamente reducida del espacio de direcciones en un determinado lapso de tiempo.

El principio tiene dos variantes:

- Localidad temporal: si un ítem es referenciado en determinado momento, es común que vuelva a ser referenciado poco tiempo después.
- Localidad espacial: cuando un ítem es referenciado en determinado momento, es común que los ítems con direcciones “cercanas” también sea accedidos poco tiempo después.

Pensemos en las estructuras de control tipo bucle ó en el procesamiento de estructuras vectoriales o matriciales y veremos enseguida como se aplican estos principios. Por ejemplo en un “for” que recorra los elementos de un array tenemos por un lado que las instrucciones dentro del bucle del “for” van a ser referenciadas una y otra vez, cada cierto tiempo, mientras la cuenta no llegue el límite. Del mismo modo el dato que está a continuación del actual será utilizado luego (suponiendo que el array se recorre de a un elemento por vez). En este ejemplo se cumplen ambas variantes del principio de localidad.

¿Cómo aprovechamos el principio de localidad en la jerarquía de memoria?. Lo hacemos manteniendo los datos “recientemente accedidos” en las jerarquías altas, más cerca de quien lo consume (la CPU). También moviendo bloques de memoria contiguos hacia las jerarquías más altas. Con lo primero se apuesta a la localidad temporal, con lo segundo a la localidad espacial.

El principio de localidad es aplicado por distintos actores dentro de un sistema:

- registros <> memoria: lo aplica el compilador ó el programador de bajo nivel
- memoria cache <> memoria: lo aplica el hardware
- memoria <> disco: lo aplica el sistema operativo (memoria virtual) ó el programador (archivos)

4 Términos utilizados en Jerarquía de Memoria

Hay una serie de términos que se utilizan cuando se habla de jerarquía de memoria.

Hit

Se dice que ocurre un *hit* cuando un objeto de información se encuentra en el lugar de la jerarquía donde se lo está buscando.

Miss

Se dice que ocurre un *miss* cuando el elemento de información no es encontrado en el lugar de la jerarquía donde se lo está buscando. En este caso es necesario ir a buscar el objeto a un nivel de jerarquía inferior.

Hit Rate

Es la tasa de acierto de encontrar un elemento de información en el lugar de la jerarquía en que se lo busca.

Miss Rate

Es la tasa de fallos en encontrar un elemento de información en el lugar buscado (y coincide con $1 - \text{Hit Rate}$).

Hit Time

Tiempo de acceso promedio en el nivel de jerarquía considerado (donde se da el hit).

Miss Penalty

Tiempo de acceso promedio adicional requerido para acceder al elemento de información en el nivel de jerarquía inferior. Típicamente ocurre que $\text{Hit Time} \ll \text{Miss Penalty}$.

Tiempo promedio de acceso a memoria

Si consideramos el nivel de jerarquía “memoria caché” podemos establecer que el tiempo promedio de acceso a memoria (es decir a elementos de información almacenados en la memoria principal de la computadora) cumple la relación:

$$\text{Tiempo promedio de acceso a memoria} = \text{Hit Time} + \text{Miss Rate} * \text{Miss Penalty}$$

5 Rendimiento de la Memoria

La velocidad de acceso a los elementos de información, incluyendo las instrucciones de los programas condiciona la capacidad de proceso de las computadoras. Es por esto que el rendimiento del sistema de memoria tiene un impacto significativo sobre el rendimiento general.

Los sistemas de memoria tienen distintos parámetros que tienen relación con su capacidad de movilizar elementos de información:

Tiempo de Acceso: es el tiempo que transcurre entre que la dirección se presenta estable y los datos pueden ser manipulados en forma confiable. Puede haber diferencias entre el tiempo de acceso de lectura y el de escritura.

Tiempo de Ciclo: es el tiempo mínimo que debe pasar entre un acceso y el siguiente (las memorias requieren de un cierto tiempo de recuperación antes de poder iniciar un nuevo ciclo de acceso). Es la suma del tiempo de acceso más el de recuperación.

Tasa de Transferencia: es la velocidad de movimiento de datos de la memoria.

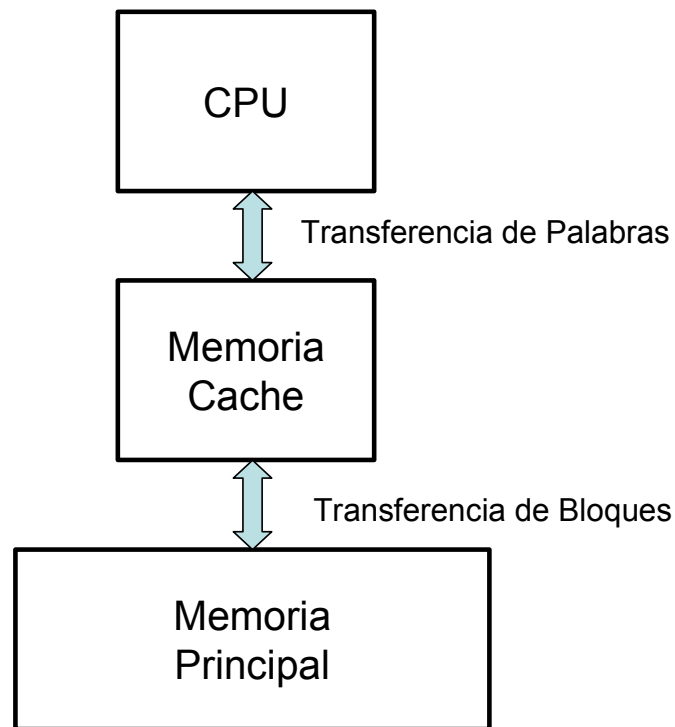
La tasa de transferencia está afectada por el tiempo de acceso y por el tiempo de ciclo, ya que ambos condicionan la velocidad con que los datos pueden ser obtenidos.

6 Memoria Caché

6.1 Introducción

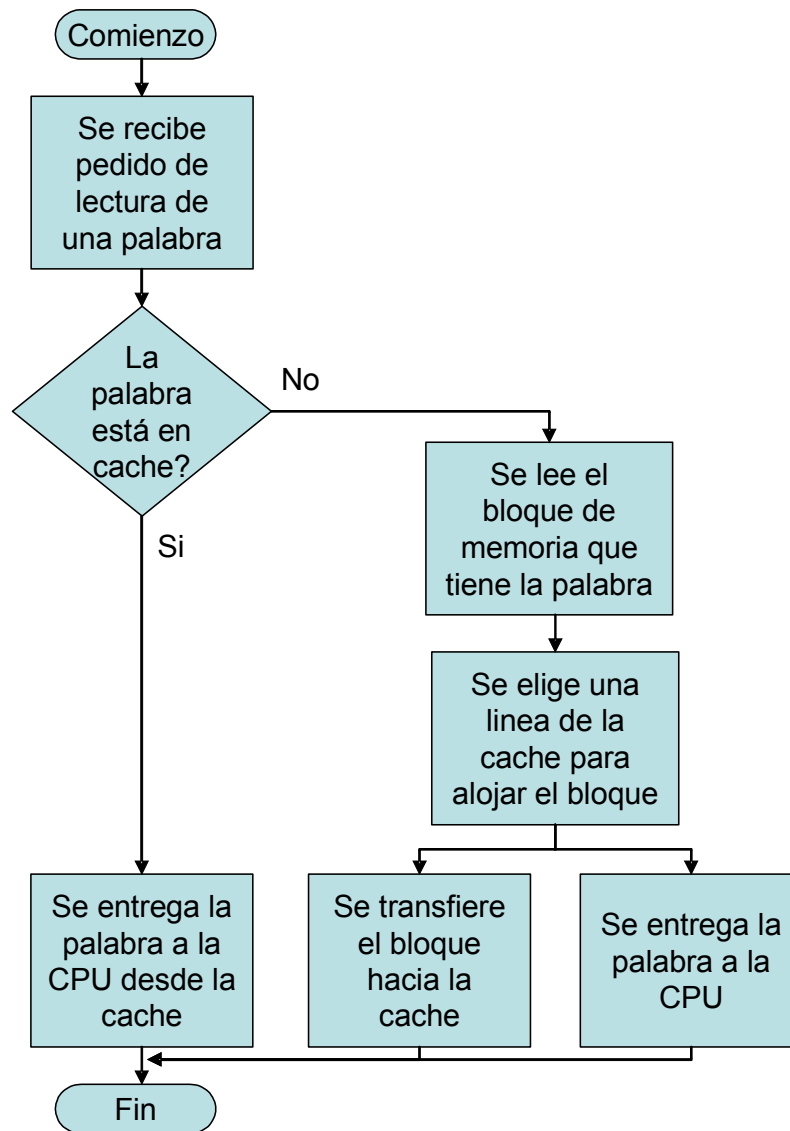
La memoria caché es una memoria de tamaño reducido, de alta velocidad, que se coloca entre la memoria principal y la CPU. Utilizando el principio de localidad mantiene copias de los bloques de memoria principal más accedidos, de manera que cuando la CPU requiere una palabra que está en uno de los bloques almacenados en la memoria caché, el

requerimiento es satisfecho desde la memoria caché, con un tiempo de acceso mucho menor que si debiera ser satisfecho desde la memoria principal.



Desde la memoria cache se accede a la memoria principal en bloques de k palabras (normalmente $k > 1$). Cuando un bloque es leído desde memoria se almacena en una **línea** de la cache.

La lógica de funcionamiento de la memoria cache es la que se esquematiza en el siguiente diagrama de flujo:



En definitiva si cuando la CPU requiere una palabra de memoria (para lectura) ocurre un “hit” en la memoria de mayor jerarquía (la cache) entonces el requerimiento se satisface desde dicha memoria, con un menor tiempo de acceso. Si por el contrario ocurre un “miss”, se dispara un mecanismo por el cual se trae desde la memoria el bloque que contiene la palabra buscada para simultáneamente escribirlo en la línea de la memoria cache que corresponda y entregar la palabra solicitada a la CPU.

En realidad luego veremos que es un poco más complejo el algoritmo, ya que al seleccionar una línea de la memoria cache para almacenar el bloque leído desde la memoria principal, puede significar la necesidad de escribir el contenido actual de la línea en el bloque de memoria principal correspondiente, dependiendo de la estrategia de “write” que se siga.

Notemos que cuando ocurre un “miss” (la palabra buscada no está en la memoria cache) al realizar la transferencia desde memoria principal del bloque entero que contiene la palabra buscada hacia la memoria cache, estamos aprovechando el principio de localidad ya que lo mas probable es que el próximo requerimiento de acceso a la memoria sea una palabra próxima (seguramente la de la dirección siguiente), por lo que probablemente estará contenida en el bloque recién leído desde memoria. Es decir, luego de un “miss” lo

más probable es que ocurra un “hit” por la estrategia de mover bloques (y no palabras) desde la memoria principal a la memoria cache.

6.2 Diseño de la Cache

En el diseño de un sistema de memoria cache hay que tomar una serie de decisiones sobre distintos aspectos que tienen impacto sobre su rendimiento, entendido como el porcentaje de “hits” respecto al total de accesos.

Primero haremos un repaso de todos estos aspectos para luego detenernos en aquellos que merecen una explicación con mayor detalle.

Tamaño

Uno de los elementos a considerar es el tamaño de la memoria cache. Ya sabemos que la memoria utilizada para este subsistema es un elemento comparativamente muy caro, por lo que siempre deberemos lograr un compromiso entre la cantidad de memoria cache y la cantidad de memoria principal.

Si bien el punto de equilibrio depende fuertemente del tipo de programas que se ejecuten en el sistema, existirá una cierta cantidad de memoria a partir de la cual el incremento del rendimiento obtenido no compensa el costo adicional de agregar más memoria cache. Actualmente la memoria cache se coloca en el propio “chip” del procesador, por lo cual también existe una limitante adicional: la cantidad de transistores disponibles en el circuito integrado es finita y compite con todos los demás sub-sistemas que deben ser construidos.

El estado del arte actual de los procesadores comerciales disponibles indican que se utilizan 3 niveles de cache: el L1 de unos 10KB a 20KB, el L2 de 128KB a 512 KB y el L3 de 4MB a 12MB. Si consideramos que una computadora actual puede tener 4GB y más de memoria, vemos que la relación entre el tamaño de la memoria principal y la cache (L3) es de 1000 a 1.

Función de Correspondencia (Mapping)

Este elemento de diseño determina como se asocia una cierta posición de memoria con su posible ubicación en la memoria cache. La memoria cache está formada por un cierto conjunto de líneas. En cada línea se puede almacenar un bloque de memoria. Por tanto la función de correspondencia establece para cada bloque de memoria cuales son las líneas posibles de ser utilizadas en la cache para almacenarlo. La relación puede ser desde que cualquier línea de la cache puede recibir a un bloque dado de memoria (denominada correspondencia “completamente asociativa”) hasta que solo una línea determinada puede recibir un bloque dado (denominada correspondencia “directa”), pasando por que cierto conjunto de n líneas puede alojar un bloque dado (es la correspondencia “asociativa por conjuntos de n vías”).

Algoritmo de Sustitución

Al momento que ocurre un “miss” es necesario traer un nuevo bloque de la memoria principal a la cache. Para los casos en que hay más de un lugar posible de colocación del bloque (casos de correspondencia “totalmente asociativa” o “asociativa por conjuntos”) es necesario tener un algoritmo para seleccionar cual de las líneas utilizar (y por tanto reemplazar su contenido en caso que no haya lugares libres, lo que va a ser la situación

habitual).

Política de Escritura

Hasta ahora hemos analizado como se comporta la memoria cache en la operación de lectura, pero también debemos analizar qué pasa cuando la CPU escribe en la memoria.

En las operaciones de escritura (*write*) existen dos grandes estrategias ***write through*** y ***write back***. La primera (*write through*) implica que, de existir un hit, la operación de escritura se hace en la memoria cache y también en la memoria principal. En la segunda en el caso del hit la escritura se realiza solamente en la memoria cache.

Naturalmente la estrategia *write back* permite un mejor desempeño del sistema en escritura, pero tiene la complejidad adicional de determinar si es necesario actualizar la memoria principal con el contenido de una línea de cache que va a ser reemplazada como consecuencia de un miss, si es que el bloque tuvo algún cambio desde que se trajo a la cache. También tiene la tarea adicional de velar por la ***coherencia de la cache***, aspecto que veremos más adelante.

Tamaño del bloque

Un aspecto que debe analizarse es el tamaño del bloque (y por tanto de la línea del cache). Para un tamaño de memoria cache dado, ¿qué conviene más? ¿una línea más grande o más cantidad de líneas?.

En el primer caso por el principio de localidad la probabilidad de un *hit* luego de un *miss* aumenta, pero se puede sufrir ante cambios de contexto en ambientes de multiprogramación.

6.3 Función de Correspondencia Directa

La función de correspondencia directa (*Direct Mapping*) a cada bloque de memoria le asocia una única línea de cache. Esto significa que hay varios bloques de memoria que tienen la misma línea de cache asignada en la función de correspondencia.

Una forma de ver esta forma de correspondencia es que cada región de memoria principal de tamaño igual a la memoria cache puede estar contenida completamente en la memoria cache. Pero dos bloques ubicados en regiones distintas, separados un múltiplo del tamaño de la memoria cache competirán por la misma línea de la cache.

La principal ventaja de esta función de correspondencia es su simplicidad y por tanto su bajo costo de implementación. La desventaja es que un programa que se mueva por regiones de memoria muy distantes, posiblemente genere una gran cantidad de *misses* repercutiendo negativamente en el rendimiento de la memoria.

6.4 Función de Correspondencia Completamente Asociativa

En este caso la función de correspondencia (denominada *Fully Associative* en inglés) vincula un bloque de memoria con cualquier línea de la memoria cache. Es decir que un bloque dado puede estar en cualquier lugar de la cache.

6.5 Función de Correspondencia Asociativa por Conjunto de N Vías

En este caso la función de correspondencia (*N-way Set Associative* en inglés)

establece que cada bloque de memoria tiene asociado un conjunto de líneas de la memoria cache y puede estar almacenado en cualquiera de las líneas del conjunto. La cantidad de *vías* es la cantidad de líneas contenida en un conjunto. Ejemplo: una memoria cache asociativa por conjuntos de 4 vías tiene 4 líneas en cada conjunto. Un bloque dado podrá estar en cualquiera de las 4 líneas de su correspondiente conjunto.

6.6 Algoritmos de Sustitución

Cuando ocurre un *miss* y es necesario leer un nuevo bloque a la memoria cache, será necesario determinar cual línea ocupar, en caso que exista más de una posibilidad.

Para el caso de la función de correspondencia directa hay una sola línea posible por lo que este problema no existe. Pero para los casos de correspondencia totalmente asociativa o asociativa por conjuntos de n vías hay más de una posibilidad y se requiere definir la manera en que se determinará la selección.

Un aspecto a tener en cuenta cuando se piensan en estos algoritmos es que los mismos deben ser posibles de ser implementados en hardware, porque no pueden significar un impacto negativo en el rendimiento del sistema. Los algoritmos más utilizados son los siguientes:

Menos Recientemente Usado (LRU = Least Recently Used)

Este algoritmo selecciona para reemplazar, dentro de las líneas posibles, la que tenga el bloque que haya sido accedido hace más tiempo. Una implementación sencilla de este algoritmo para una memoria cache asociativa por conjuntos de 2 vías es usar un bit que indica cual fue la última línea que se accedió. En cada acceso a un conjunto se actualizan los bits de ambas líneas del conjunto.

FIFO (First In First Out)

En este caso se selecciona la línea que contiene el bloque que haya sido traído primero desde la memoria principal (el más antiguo), sin importar si fue accedido ni que tantas veces lo fue. Una manera de implementar este algoritmo en hardware es mediante un buffer circular (con un puntero de circular por conjunto que señala la línea a reemplazar que se actualiza en cada reemplazo).

Menos Frecuentemente Utilizado (LFU = Least Frequently Used)

Este algoritmo utiliza la cantidad de veces que han sido accedidos los bloques de las líneas candidatas a ser reemplazadas. Para su implementación se pueden utilizar contadores en cada línea.

Random

El algoritmo random selecciona el bloque a reemplazar mediante una técnica aleatoria (normalmente pseudo-aleatoria por razones de implementación).