Memoria Caché

Organización de Computadoras

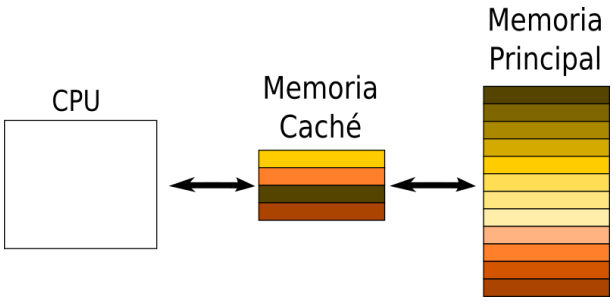
Universidad Nacional de Quilmes

Durante la ejecución de un programa se accede a memoria con un criterio que no es aleatorio, pero tampoco absolutamente predecible. Los accesos respetan cierta lógica que tiene relación con la naturaleza de la ejecución de los programas, donde se pueden distinguir dos tipos de accesos a memoria: por lectura de instrucciones y por lectura de datos.

La lectura de instrucciones es en su mayoría un recorrido de celdas consecutivas de memoria, y asi mismo la lectura de los datos de un arreglo también lo es. Por otro lado, las instrucciones de una rutina o de una repetición se utilizan más de una vez.

Estos patrones de acceso se describen con el nombre de **principios de localidad**: El principio de **localidad espacial** enuncia que las posiciones de memoria cercanas a alguna accedida recientemente son más probables de ser requeridas que las más distantes, y el principio de **localidad temporal** dice que cuando un programa hace referencia a una posición de memoria que contiene una instrucción, es probable que lo vuelva a hacer en poco tiempo.

A partir de identificar lo anterior es que se pensó una memoria intermedia (entre la CPU y la memoria principal) que contenga las celdas más usadas (y no todas), denominada ***memoria caché*** (del francés, escondida).



De manera general el funcionamiento con una memoria caché requiere que la caché resuelva qué celdas son aquellas mas accedidas, y lo haga de manera transparente a la CPU. En otras palabras, la CPU se debe abstraer de la lógica de funcionamiento de la memorica caché.

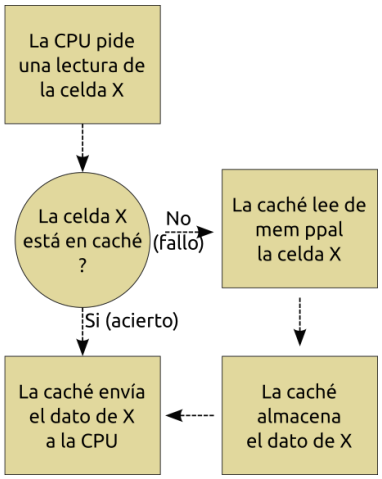


Figura 1: Mecanismo de acceso a Memoria Caché

1. Función de correspondencia

Las funciones de correspondencia proponen distintos criterios para corresponder los bloques de memoria principal con las líneas de memoria caché, es decir: con que flexibilidad se almacenan los bloques en las líneas.

Cuando la Unidad de Control pide una determinada celda, la memoria caché debe, en primer lugar, determinar si la dirección corresponde a una celda cacheada. Para responder esta pregunta, debe aplicar su **función de correspondencia**

1.1. Correspondencia asociativa

El enfoque mas natural es la **correspondencia completamente asociativa**, donde cada línea de caché se puede llenar con cualquier dato de la memoria principal, y por lo tanto se requiere almacenar una **etiqueta o tag** que permita identificar al dato que se almacena.

Dicho en otras palabras, el contenido de cada celda leída se **asocia** a una **etiqueta** que identifica su origen, es decir, su dirección de memoria principal. Esto permite que las celdas puedan ser almacenadas en cualquier ranura,y por lo tanto se debe chequear en todas ellas si alguna contiene la dirección buscada como tag.

Por ejemplo, considerá una memoria principal con direcciones de 6 bits (64 celdas) y donde cada celda de almacena 1 byte. En la siguiente memoria caché las celdas cacheadas son las `010000` a `010011`:

00	010000	11111111
01	010001	11111111
10	010010	11111111
11	010011	11111111

Normalmente, las memorias caché tienen capacidad para almacenar varias

celdas en un misma linea[1], para aprovechar el principio de localidad espacial. En estos casos el tag ya no es simplemente la dirección de una celda, sino el número de bloque (que está relacionado con las direcciones que lo componen). Siguiendo el ejemplo anterior, si utilizamos una caché con bloques de 4 celdas, se tienen $\frac{64}{4} = 16$ bloques, y entonces el número de bloque tiene 4 bits, pues $2^4 = 16$. De aquí que el bloque número `0000` tiene las celdas `000000` a `000011`. bloque número `0001` tiene las celdas `000100` a `000111`. Notar que se subrayó el número de bloque dentro de la dirección de la celda.

00	1100	01010101111111110101010111111111
01	1010	11111111010101011111111110101010
00	0000	01010101111111110101010111111111
01	0110	11111111010101011111111110101010

Veamos una lectura de ejemplo: ¿cómo determina la caché si la celda `011010` está cacheada? Para eso utiliza los primeros 4 bits que corresponden al número de bloque (`0110`). ¿Qué parte de la linea envía como respuesta a la CPU? Para eso utiliza los últimos 2 bits (`10`), que se denominan **bits de palabra**, para extraer una porción de la linea.

$$\frac{11111111}{00}\ \frac{10101010}{01}\ \frac{11111111}{10}\ \frac{10101010}{11}$$

De esta manera, el dato que envía es el tercer byte (`11111111`)

1.2. Correspondencia Directa

La implementación de una correspondencia asociativa tiene un costo elevado debido a que se requiere la búsqueda de la clave (dirección de la celda o número de bloque) en todas las líneas en forma simultánea y esto requiere una tecnología de acceso aleatorio. Para reducir este costo es posible presentar otro mecanismo de correspondencia donde cada bloque de memoria principal esté asignado a una línea determinada. Como la cantidad de líneas es mucho menor a la cantidad de bloques (porque en otro caso la memoria principal y la caché tendrían la misma capacidad), hay un conjunto de bloques candidatos para una misma línea que "compiten" entre si. Este mecanismo se denomina **correspondencia directa** y cada línea de caché es un recurso a compartir por un conjunto de bloques de memoria. Es importante marcar que este conjunto de bloques no es consecutivo, con el objetivo de aprovechar el principio de localidad y maximizar la tasa de aciertos. El criterio que se usa para corresponder bloques con lineas es circular, como describe la figura 2.

A la hora de verificar si una celda está cacheada, la memoria caché compara solamente en una línea (la que corresponde al bloque) para determinar si contiene el tag buscado, simplificando la tecnología que se necesita para implementar la caché.

Continuando el ejemplo de la memoria descripta en la sección anterior, considerar una caché con 8 lineas y bloques de 4 celdas. Esto permite simplificar la búsqueda de un tag en la caché, pues se debe buscar en una sóla línea, la que corresponde al bloque buscado. ¿Cómo determina qué línea es esa? Notar primero que el *tag* es mas pequeño que en el caso del mapeo asociativo: Se

[1] ranura=línea=slot

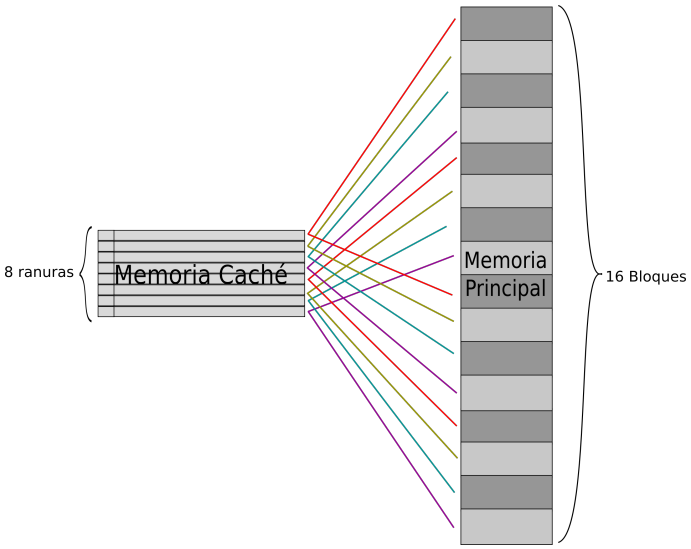


Figura 2: Correspondencia directa

tienen $\frac{16}{8} = 2$ bloques por ranura. Es decir que con un bit (pues $2^1 = 2$) alcanza para identificar cual de esos 2 bloques está almacenado en la ranura en cuestión. Notar también que la ranura `000` almacena el bloque `0000` o el bloque `1000`, y de aquí que el bit mas significativo dentro del número de bloque puede usarse como *tag*.

Analicemos una lectura de ejemplo:

(I) ¿cómo determina la caché si la celda `011010` está cacheada? El número de bloque son los primeros 4 bits (`0110`) y a ese bloque le corresponde la linea `110`. Entonces allí debe verificarse que contenga el tag `0`.

(II) ¿Qué parte de la linea envía como respuesta a la CPU? Como en el caso asociativo, utiliza los **bits de palabra** (`10`) para extraer una porción de la linea.

2. Correspondencia asociativa por conjuntos

A la hora de comparar los enfoques anteriores, se ve que la correspondencia directa es mas económica en su construcción pero la correspondencia asociativa es mas flexible y maximiza el porcentaje de aciertos. Para hacerlo evidente, suponer una secuencia de accesos que requiera repetidamente cachear bloques que corresponden a una misma línea, causando repetidos fallos. En una correspondencia asociativa, esos bloques no compiten y no se darían mas fallos que los que producen bloques nuevos.

En una búsqueda de balancear los dos aspectos mencionados (eficiencia vs. costo), se propone una **correspondencia asociativa por conjuntos**, donde la memoria caché se divide en conjuntos de N líneas y a cada bloque le corresponde uno de ellos. Es decir que dentro del conjunto de líneas asignado, un bloque de

memoria principal puede ser alojado en cualquiera de las N líneas que lo forman, es decir que dentro de cada conjunto la caché es totalmente asociativa.

Esta situación es la más equilibrada, puesto que se trata de un compromiso entre las técnicas anteriores: si N es igual a 1, se tiene una caché de mapeo directo, y si N es igual al número de líneas de la caché, se tiene una caché completamente asociativa. Si se escoge un valor de N apropiado, se alcanza la solución óptima.

En la figura 3 se ejemplifica esta nueva forma de correspondencia.

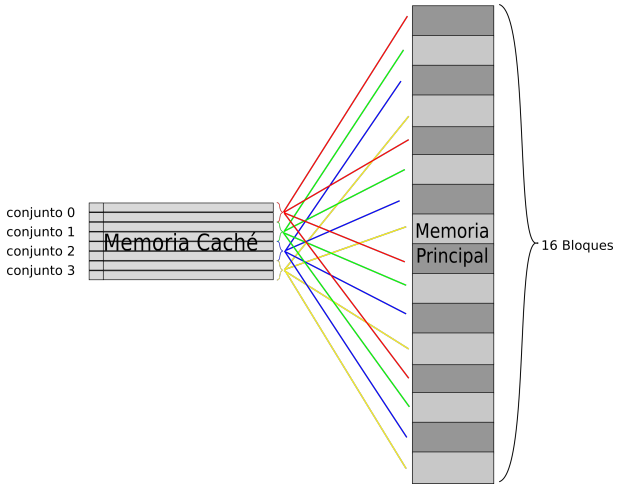


Figura 3: Correspondencia asociativa por conjuntos

Con este mecanismo, las lineas se agrupan en conjuntos, y cada conjunto se corresponde con determinados bloques de la memoria principal. Además, dentro de cada conjunto los bloques se almacenan con un criterio asociativo. Cuando la caché recibe una dirección, debe determinar a que conjunto corresponde el bloque de la celda buscada y luego buscar asociativamente dentro del conjunto el tag que corresponde.

Suponer una computadora como la de la figura, con 16 bloques de 4 celdas en la memoria principal, y una memoria caché con 4 conjuntos de 2 linea cada uno.

Por ejemplo, el conjunto `00` puede almacenar los bloques `0000`,`0100`,`1000` y `1100`. Si se pide la lectura de la celda `110011`, que corresponde al bloque `1100`, entonces se deben analizar las lineas del conjunto `00`, buscando el tag `11`.

3. Fallos y reemplazos

Cuando se produce un **fallo** y un nuevo bloque debe ser cargado en la caché, debe elegirse una ranura que puede o no contener otro bloque para ocupar. En el caso de correspondencia directa, no se requiere hacer tal elección, pues existe

solo una posible ranura para un determinado bloque.

Sin embargo, en los casos de las correspondencia asociativa y correspondencia asociativa por conjuntos, dado que ambos mecanismos aplican mayor o menor nivel de asociatividad, se necesita un criterio para elegir el bloque que será reemplazado. Con este objetivo se diseñan los **algoritmos de reemplazo** descriptos a continuación.

3.1. Algoritmo LRU (Least Recently Used)

El algoritmo más usado es el algoritmo *LRU*. Tiene la característica de que luego de cada referencia se actualiza una lista que indica cuan reciente fue la última referencia a un bloque determinado. Si se produce un fallo, **se reemplaza aquel bloque cuya última referencia se ha producido en el pasado más lejano**.

Para poder implementarlo, es necesario actualizar, luego de cada referencia, una lista que ordena los bloques por ultimo acceso, indicando cuan reciente fue la última referencia a cada bloque. Si se produce un fallo, se reemplaza aquel bloque que está ultimo en la lista, pero si ocurre un acierto, el bloque accedido debe convertirse en el primero de la lista.

Por ejemplo, considerar una memoria principal de 64 celdas (direcciones de 6 bits), y una memoria caché con 4 líneas y 8 celdas por bloque. La política LRU para una lista de accesos de ejemplo se muestra en la tabla 1

Dirección	N. Bloque	A/F	Lista de bloques
111000	111	Fallo	111
011001	011	Fallo	011,111
001111	001	Fallo	001,011,111
110000	110	Fallo	110,001,011,111
011111	011	Acierto	011,110,001,111
111111	111	Acierto	111,011,110,001
000111	000	Fallo	000,111,011,110
001111	001	Fallo	001,000,111,011

Cuadro 1: Ejemplo LRU

En una caché completamente asociativa la lista es una sola, pero en una caché asociativa por conjuntos, cada conjunto debe mantener su propia lista ordenada. En ambos casos se necesita almacenar información extra para simular cada lista encadenada.

Diversas simulaciones indican que las mejores tazas de acierto se producen aplicando este algoritmo.

3.2. Algoritmo FIFO (First In-First Out)

La política **FIFO** (el primero en entrar es el primero en salir) hace que, frente a un fallo, se siga una lista circular para elegir la línea donde se alojará el nuevo bloque. De esta manera, se elimina de la memoria cache aquel que fue cargado en primer lugar y los bloques siguientes son removidos en el mismo orden, analogamente a lo que ocurre en una cola de espera en un banco.

Similarmente para el caso LRU, en una memoria totalmente asociativa por conjuntos se debe mantener el registro de una sola lista, y en una asociativa por conjuntos, una por cada conjunto.

Es posible observar que esta política no tiene en cuenta el principio de localidad temporal. Para hacerlo, considerar el caso donde un bloque es requerido repetidamente, en una memoria caché como la del caso LRU. Cada 4 fallos, dicho bloque es reemplazado por otro, sin importar si es muy usado.

La implementación de la lista puede resolverse a través de un bit que se almacena en cada línea (además del tag y de los datos del bloque) que indica cuál es el bloque candidato a ser reemplazado. De esta manera, solo uno de los bloques tiene un 1, y los restantes deben tener 0. Cuando ocurre un fallo, el bloque que tenía un 1 se reemplaza, se resetea el bit y se setea el bit (de 0 a 1) de la siguiente línea.

3.3. Algoritmo LFU (Least Frequently Used)

En el método **LFU** (*Least Frequently Used*) el bloque a sustituir será aquel que se acceda **con menos frecuencia**. Será necesario entonces registrar la frecuencia de uso de los diferentes bloques de caché. Esta frecuencia se debe mantener a través de un campo contador almacenado en cada línea, pero esto presenta una limitación en el rango de representación. Para mejorar esto, una posible solucion es ir recalculando la frecuencia cada vez que se realice una operación en caché, dividiendo el número de veces que se ha usado un bloque por el tiempo que hace que entró en caché. La contraparte de este método es que el cálculo mencionado presenta un costo adicional elevado.

3.4. Algoritmo Aleatorio

Con esta política, el bloque es elegido al azar. En una memoria asociativa por conjuntos, el azar se calcula entre los bloques que forman el conjunto en el cual se ha producido la falla.

Esta política es contraria al principio de localidad, pues lo desconoce; sin embargo, algunos resultados de simulaciones indican que al utilizar el algoritmo aleatorio se obtienen tazas de acierto superiores a los algoritmos FIFO y LRU.

Posee las ventajas de que por un lado su implementación requiere un mínimo de hardware y por otro lado no es necesario almacenar información alguna para cada bloque.

Fuente

1. *Organización y Arquitectura de Computadoras , William Stallings, Capítulo cuarto*