Subsistema de memoria

Organización de Computadoras 2019

Universidad Nacional de Quilmes

Índice

1. Subsistema de memoria

El sistema de computos necesita tener varios tipos de memorias para ser utilizadas en diferentes situaciones cumpliendo distintos objetivos. Antes de mostrar porque es esto cierto, veamos cómo caracterizar las memorias según los diferentes aspectos.

En primer lugar, una unidad puede ser interna o externa al sistema según si es fija o desmontable (esto es, portable a otro sistema). Además puede ser volátil, si su contenido se pierde al perder la alimentación eléctrica, o persistente, si no necesita electricidad para mantener la información. Puede ser de lectura y escritura, como se requiere en una memoria principal, o bien de solo lectura para almacenar información estática que no necesita modificarse.

Volatilidad

Por último, se las puede clasificar segun su método de acceso, en secuencial, directo, aleatorio o asociativo. El **método de acceso secuencial** requiere que la dirección (o identificación) de cada dato esté almacenada junto con él,

Métodos de Acceso

y por lo tanto el dispositivo debe recorrerse secuencialmente hasta encontrar la identificación buscada. El **método de acceso directo** es el utilizado por los discos magnéticos, donde la direccion del dato se basa en su ubicación física, en particular la busqueda se da en dos etapas: se accede primero a la zona que incluye al dato y luego se busca secuencialmente dentro de esa zona. En el **método aleatorio**cada dato tiene un mecanismo de acceso único y cableado físicamente (cada acceso es independiente de los accesos anteriores). Finalmente, el **método asociativo** organiza cada unidad de información con una etiqueta que la describe en función de su contenido. Entonces, para recuperar un dato se debe analizar un determinado conjunto de bits dentro de la celda, buscando la coincidencia con la clave o patrón buscado. Esta comprobación del contenido de las celdas se lleva a cabo de manera simultánea en todas las celdas.

1.1. Memorias ROM

Para algunas aplicaciones, el programa no necesita ser modificado y entonces puede ser almacenado de manera permanente en una memoria de solo lectura ó ROM (*Read Only Memory*). Ejemplos de memorias ROM pueden encontrarse en videojuegos, calculadoras, hornos de microondas, computadoras en automóviles, etc. Además casi todas las computadoras personales necesitan una memoria ROM donde almacenar el primer programa que da arranque al sistema operativo a partir del acceso (en la mayoria de los casos) a un disco rígido primario.

1.2. Jerarquía de memorias

En la arquitectura de Von Neumann la **memoria principal** es volátil y de acceso aleatorio. Si es volátil entonces se necesita otra posibilidad de almacenamiento donde los programas puedan almacenarse de manera persistente y desde donde se recuperen bajo demanda del usuario (cuando dispara la ejecución de un programa).

Memoria Secundaria

Esta **memoria secundaria** puede ser resuelta con un disco rígido o un disco de estado sólido (SSD) donde se mantengan persistentes (instalados) los programas. Cuando se necesita de la ejecución de un programa, hecho que puede darse a partir del requerimiento de un usuario o por invocación de otro programa, este debe ser cargado en memoria y permanece allí hasta que la memoria se sobreescribe o se apaga el sistema.

La pregunta que puede surgir es ¿porque no montar la memoria principal en una tecnología persistente, como puede ser un disco de estado sólido? Para responderla es importante destacar que existe una relación de compromiso entre el tiempo de acceso, el costo por bit y la capacidad de almacenamiento (ver figura 1). Un disco tiene mayor capacidad (y por lo tanto menor costo por bit) pero un tiempo de acceso mucho mayor. Esto impacta directamente en el desempeño de la CPU pues el tiempo de ejecución de una instrucción está condicionado por el tiempo de acceso a memoria principal. Además, en este punto es importante notar que algunos programas pueden no ser ejecutados nunca y algunos datos pueden no ser accedidos, aunque se los tenga disponibles en el sistema.

Es por este motivo que se incorpora una memoria mas pequeña y rápida (de acceso aleatorio) donde se cargan los programas a la hora de ser ejecutados y

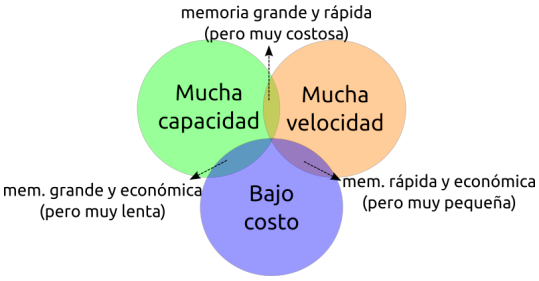

Figura 1: Relación de compromiso entre capacidad, velocidad y costo

los datos cuando se los necesita. Si lo que se necesita es asegurar una velocidad aceptable y no se necesita gran capacidad, ¿Porqué no implementar la memoria principal con regitros de la CPU? Para responder esto se debe tener en mente que el costo sea razonable. En general las arquitecturas cuentan con pocas decenas de registros debido al costo que eso implica. Por otro lado, si se implementa la memoria principal con una RAM se otorga flexibilidad para extender su tamaño pues se trata de un componente externo a la CPU y en cambio los registros suelen estar definidos en el diseño electrónico de la misma. Este esquema se describe en la figura 2.

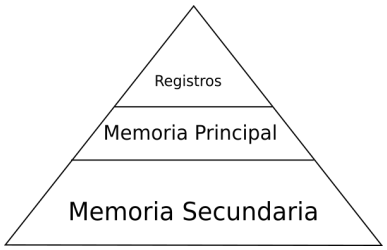

Figura 2: Jerarquía de memoria

2. Memoria Caché

Desde hace un par de décadas el progreso tecnologico de las computadoras personales esta marcando una tendencia a duplicar, cada año y medio, la velocidad de los procesadores y el tamaño de la memoria principal pero la velocidad de las memorias apenas crece un 10 %. Esto ocasiona un crecimiento de la brecha entre la velocidad del procesador y la velocidad de la memoria, causando un mayor impacto en el tiempo de ejecución de los programas, pues la velocidad con la que la CPU puede ejecutar instrucciones está limitada por el tiempo de acceso a memoria, dado que al menos una vez es necesario accederla dentro del ciclo de ejecución de instrucción.

Un enfoque para buscar una solución a esta brecha es incorporar una memoria mas pequeña que la memoria principal pero mas rápida, teniendo en mente que

no es viable construir la memoria principal a partir de registros internos a la CPU.

Esta idea se basa en que los accesos que se solicitan no son al azar, pero tampoco absolutamente predecible. Los accesos respetan una cierta lógica que tiene relación con la naturaleza de la ejecución de los programas, pudiendose distinguir dos tipos de accesos: lectura de instrucciones y 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 piensan para ser utilizadas mas de una vez.

Principios de localidad espacial y temporal

Estos patrones de acceso se describe con los **principios de localidad** : el principio de **localidad espacial** enuncia que las posiciones de memoria cercanas a alguna accedida recientemente son mas probables de ser requeridas que las mas distantes, y el principio de **localidad temporal** dice que cuando un programa hace referencia a una posicion de memoria, se espera que vuelva a hacerlo en poco tiempo.

A partir de esta idea, se busca tener las celdas mas usadas (y no todas) en una memoria mas inmediata, intermedia a la cpu y la memoria principal (ver figura 3), denominada **Memoria Caché** (del francés: 'escondida').

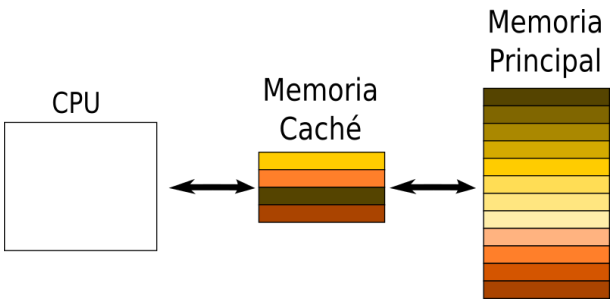

Figura 3: Memoria Caché como intermediaria

De esta manera podemos reformular la pirámide de la figura 2 incorporando una memoria mas costosa que los registros, y con menos capacidad que la memoria principal, como se muestra en la figura 4.

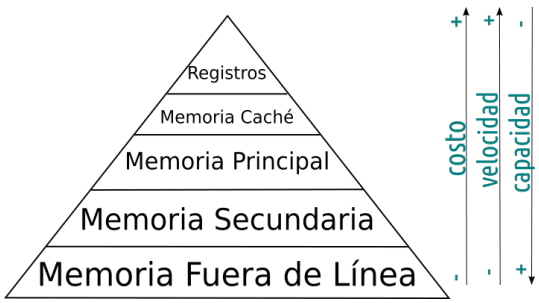

Figura 4: Jerarquía con Memoria Caché

La memoria caché tiene como objetivo proveer una velocidad de acceso cercana a la de los registros pero al mismo tiempo proveer un mayor tamaño de memoria. Para esto, la caché contiene una copia de porciones de la memoria principal, y en el momento de leer una celda, primero se verifica si se encuentra en la caché. **Si esto no ocurre** debe traerse de la memoria principal un bloque de celdas a la caché y luego la celda se entrega a la CPU (ver figura 5). El manejo de bloques respeta el principio de localidad espacial, que dice que cuando un bloque de datos es traído a la caché porque se necesitó una de sus celdas, entonces es probable que próximamente se necesiten otras celdas del mismo bloque.

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.

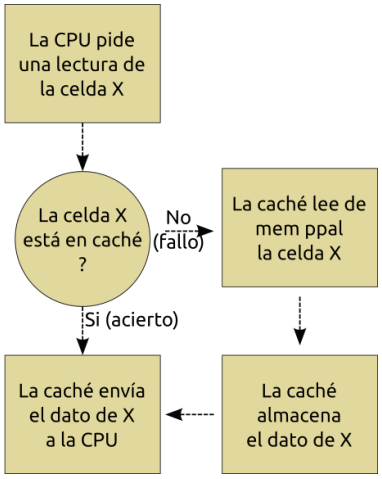

Figura 5: Mecanismo de acceso a Memoria Caché

2.1. 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 una determinada **función de correspondencia**

2.1.1. 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	11111111101010101111111110101010
00	0000	01010101111111110101010111111111
01	0110	11111111101010101111111110101010

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`)

2.1.2. 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**

[1] ranura=línea=slot

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 6.

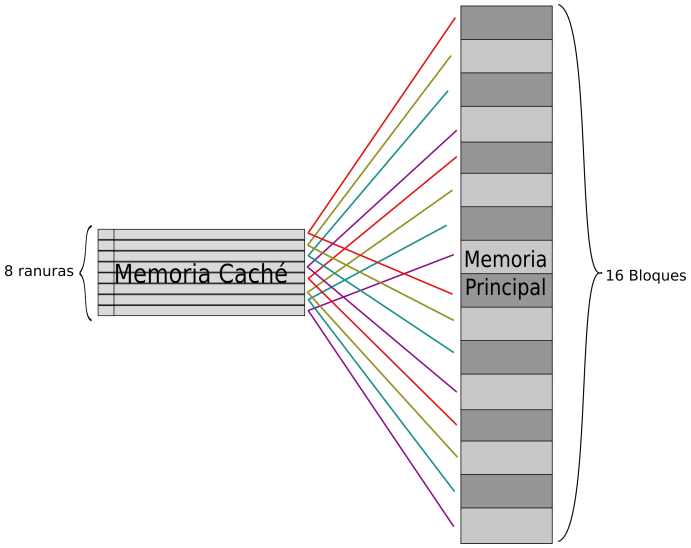

Figura 6: Correspondencia directa

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 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.

2.2. 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 7 se ejemplifica esta nueva forma de correspondencia.

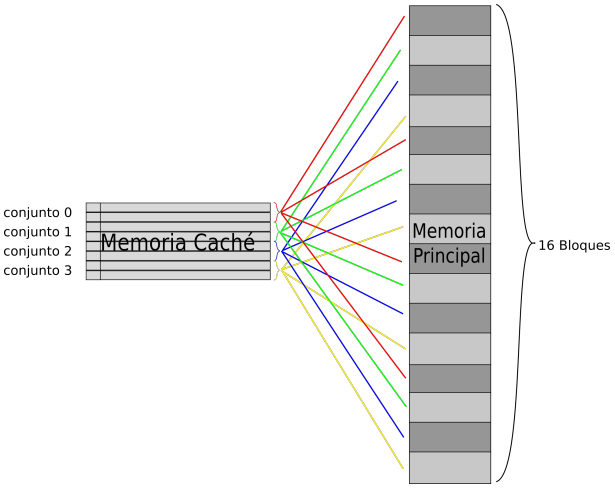

Figura 7: Correspondencia asociativa por conjuntos

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`.

2.3. 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.

2.3.1. 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.

2.3.2. 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.

2.3.3. 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.

2.3.4. 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.

2.4. Desempeño (performance) de la caché

El desempeño de una memoria Caché está dado por la cantidad de aciertos y fallos en un conjunto representativo de programas. Al diseñar una caché se debe proponer dimensiones (cantidad de líneas, capacidad de las líneas), asi como funciones de correspondencia y algoritmos de reemplazo que permitan maximizar los aciertos para un conjunto de programas de prueba. La condición mínima es al ejecutar esos programas en una arquitectura con la caché diseñada el tiempo de acceso sea menor que al ejecutarlos sin una memoria caché.

Suponer una máquina con arquitectura **Q6** con una memoria caché de correspondencia directa de 64 líneas y 4 celdas por bloque. Se tiene el siguiente programa ensamblado a partir de la celda B110 y se sabe que R1 = AC00, R3 = A702, SP=FFEE y que la celda A702 tiene el valor 1.

	...
B110	CMP [R3],
B111	0x0000
B112	JE fin
B113	ADD R0, [R1]
B114	ADD R2, [R3]
B115	fin: RET
	...

Se necesita analizar la performance de la caché en cuanto a la cantidad de fallos que se producen durante la ejecución del programa. Para eso se lleva registro de las direcciones que se solicitan a la memoria principal, y para cada una si produjo un acierto o fallo. Ver la siguiente tabla:

dir(hexa)	dir (binario)	tag	linea	F/A
B110	1011000100010000	10110001	000100	F
B111	1011000100010001	10110001	000100	A
B112	1011000100010010	10110001	000100	A
B113	1011000100010011	10110001	000100	A
AC00	1010110000000000	10101100	000000	F
B114	1011000100010100	10110001	000101	F
A702	1010011100000010	10100111	000000	F
B115	1011000100010101	10110001	000101	A

En la ejecución se produjeron 8 accesos, de los cuales 4 fueron fallos, es decir que se tiene una **tasa de fallos** de $\frac{4}{8} = 0,5$. Si el tiempo de acceso a la memoria es de 0,5ms y el de la cache es 0,05ms, se sabe que la ejecución de ese programa llevó en total:

$$4 * (0{,}5ms + 0{,}05ms) + 4 * 0{,}05ms = 2,4ms$$

3. Memoria Secundaria

3.1. Discos Magnéticos

Para la lectura o la escritura del disco, el cabezal debe estar posicionado al comienzo del sector deseado. En el caso de los dispositivos de cabezal móvil, el

posicionamiento implica mover el cabezal hasta la pista deseada (*seek*) y luego esperar a que el sector deseado pase por debajo del cabezal (*latency*). De esta manera, el **tiempo de acceso al disco** es la suma del seek time, el latency time y el tiempo de transmisión.

Tiempo de Búsqueda (seek time): Es el tiempo que le toma a las cabezas de Lectura/Escritura moverse desde su posición actual hasta la pista donde esta localizada la información deseada. Como la pista deseada puede estar localizada en el otro lado del disco o en una pista adyacente, el tiempo de búsqueda variara en cada búsqueda. En la actualidad, el tiempo promedio de búsqueda para cualquier búsqueda arbitraria es igual al tiempo requerido para mirar a través de la tercera parte de las pistas. Los HD de la actualidad tienen tiempos de búsqueda pista a pista tan cortos como 2 milisegundos y tiempos promedios de búsqueda menores a 10 milisegundos y tiempo máximo de búsqueda (viaje completo entre la pista más interna y la más externa) cercano a 15 milisegundos .

Latencia (latency): Cada pista en un HD contiene múltiples sectores una vez que la cabeza de Lectura/Escritura encuentra la pista correcta, las cabezas permanecen en el lugar e inactivas hasta que el sector pasa por debajo de ellas. Este tiempo de espera se llama latencia. La latencia promedio es igual al tiempo que le toma al disco hacer media revolución y es igual en aquellos drivers que giran a la misma velocidad. Algunos de los modelos más rápidos de la actualidad tienen discos que giran a 10000 RPM o más reduciendo la latencia.

Command Overhead: Tiempo que le toma a la controladora procesar un requerimiento de datos. Este incluye determinar la localización física del dato en el disco correcto, direccionar al .ªctuador"para mover el rotor a la pista correcta, leer el dato, redireccionarlo al computador.

Transferencia: Los HD también son evaluados por su transferencia, la cual generalmente se refiere al tiempo en la cual los datos pueden ser leídos o escritos en el drive, el cual es afectado por la velocidad de los discos, la densidad de los bits de datos y el tiempo de acceso. La mayoría de los HD actuales incluyen una cantidad pequeña de RAM que es usada como cache o almacenamiento temporal. Dado que los computadores y los HD se comunican por un bus de Entrada/Salida, el tiempo de transferencia actual entre ellos esta limitado por el máximo tiempo de transferencia del bus, el cual en la mayoría de los casos es mucho más lento que el tiempo de transferencia del drive.

3.2. Discos de estado sólido

Una unidad de estado sólido o SSD (acrónimo en inglés de *solid-state drive*) es un dispositivo de almacenamiento de datos que usa una memoria no volátil, como la memoria flash, para almacenar datos, en lugar de los platos giratorios magnéticos encontrados en los discos mencionados antes.

En comparación con los anteriores, las unidades de estado sólido son menos sensibles a los golpes, son prácticamente inaudibles y tienen un menor tiempo de acceso y de latencia.

La mayoría de los SSD utilizan memoria flash basada en compuertas NAND, que retiene los datos sin alimentación. Para aplicaciones que requieren acceso rápido, pero no necesariamente la persistencia de datos después de la pérdida de potencia, los SSD pueden ser construidos a partir de memoria de acceso aleatorio (RAM). Estos dispositivos pueden emplear fuentes de alimentación independientes, tales como baterías, para mantener los datos después de la desconexión de la corriente eléctrica.

3.3. Redundant Array of Inexpensive Drives

RAID es una tecnología que emplea el uso simultáneo de dos o mas discos duros para alcanzar mejor performance, mayor fiabilidad y mayor capacidad de almacenamiento. Los diseños de RAID involucran dos objetivos de diseño: mejor fiabilidad y mejor performance de lectura y escritura. Cuando un conjunto de discos físicos se utilizan en un RAID, se los denomina conjuntamente **vector RAID**. Este vector distribuye la información a través de varios discos, pero esto es transparente para el sistema operativo, pues lo maneja como si se tratara de un solo disco.

Algunos vectores RAID son redundantes en el sentido que se almacena información extra, derivada de la información original, de manera que el fallo de un disco en el vector no provocará pérdida de infomación. Una vez reemplazado el disco, su información se reconstruye a partir de los otros discos y la información extra. Claramente, un vector redundante tiene menos capacidad de almacenamiento.

Existen varias combinaciones de estos enfoques dando diferentes relaciones de compromiso entre proteccion contra la perdida de datos, capacidad y velocidad. Estas combinaciones se denominan niveles, y los niveles 0,1 y 5 son los mas comunmente encontrados pues cubren la mayoría de los requerimientos.

RAID 0 (*striped disks*): Distribuye la información a través de distintos discos de manera que se mejore la velocidad y la capacidad total, pero toda la información se pierde si alguno de los discos falla.

RAID 1 (*mirrored disks*): Utiliza dos (en algunos casos mas) discos que almacenan exactamente la misma información. Esto permite que la información no se pierda mientras al menos un disco funcione. La capacidad total de este esquema es la misma que la de un disco, pero el fallo de un dispositivo no compromete el funcionamento del arreglo de discos.

RAID 5 (*striped disks with parity*): Este esquema combina tres o mas discos de una manera que se protege la información contra la perdida de un disco y la capacidad de almacenamiento del arreglo se reduce en un disco.

La tecnología RAID involucra importantes niveles de cálculo durante lecturas y escrituras. Si el RAID está implementado por hardware, esta tarea es llevada a cabo por el controlador. En otros casos, el sistema operativo requiere que el procesador lleve a cabo el mantenimiento del RAID, lo que impacta en la performance del sistema.

Los RAID redundantes pueden continuar trabajando sin interrupcion cuando ocurre una falla, pero son vulnerables a fallas futuras. Algunos sistemas de alta disponibilidad permiten que el disco con fallas sea reemplazado sin necesidad de reiniciar el sistema.

Nivel de redundancia

$$red = capTotal/capU$$