GUÍA PARA APROBAR EC – MEMORIA CACHÉ

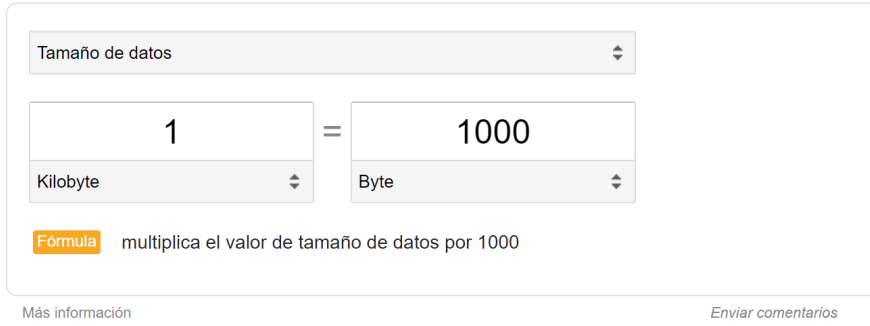
Para sacar estos ejercicios hay que saber principalmente lo siguiente:

1. Sacar el formato de la dirección de la MC
2. Calcular el número de fallos y optimizar código
3. Calcular el rendimiento
4. Cachés con Pre-Búsqueda
5. Cachés víctimas
6. Cachés multinivel
7. Cachés segmentadas

Antes de comenzar, un aviso:

IMPORTANTE

Si buscáis las unidades de medida de la memoria en Google, ¡CUIDADO! Os saldrá lo siguiente:



Y es que Google está en lo cierto: 1KB no son 2^{10} bytes sino 10^3 bytes. ¿Entonces por qué usamos en los problemas la primera unidad de medida?

En verdad, aunque se hable de KiloByte (KB), MegaByte (MB), GygaByte (GB)… en realidad a lo que se hace referencia es a **KibiByte (KiB), MebiByte (MiB), GybiByte (GiB)…** que son unas nuevas unidades de medida que se propusieron para facilitar los cálculos (las potencias de dos son más cómodas) y que han acabado por sustituir completamente a las otras., robándoles hasta el nombre xd. Así que eso, tenedlo en cuenta si vais a buscarlo:

$*1024 = 2^{10}$

Sacar el formato de la dirección de la MC

El problema normalmente siempre nos dará tres tipos de datos: **el tamaño de la memoria principal, el tamaño de la memoria caché y el tamaño de bloque**. Con estos tres datos ya es posible sacar el formato de la dirección de la MC.

El tamaño de cada memoria vendrá expresado normalmente en KB, Bytes (B), y más ocasionalmente en GB y MB. Es **ESENCIAL** saber pasar todas estas unidades a una potencia de dos en Bytes. Sin ello no podremos sacar cuantos bits necesita cada memoria, ya que estos serán el exponente de la potencia de 2.

Unidad	Bytes
1 KB	2^{10}
1 MB	2^{20}
1 GB	2^{30}

Según el tipo de emplazamiento que use la caché, la dirección tomará un formato u otro.

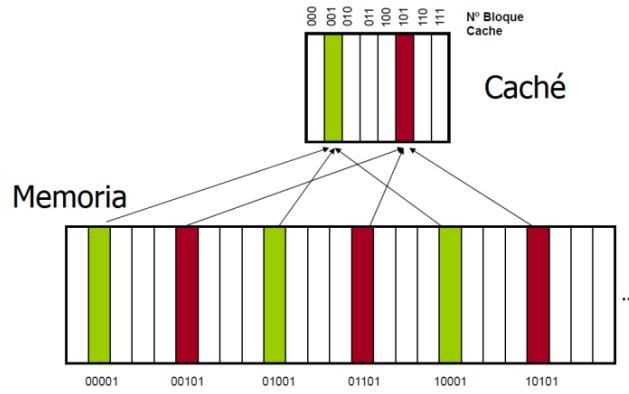
¿Qué es el emplazamiento?

Es la forma de decidir para cada bloque de memoria de la memoria principal (MP) qué bloque de la memoria caché (MC) le corresponde. Existen tres tipos de emplazamientos:

1. Emplazamiento directo
2. Emplazamiento completamente asociativo
3. Emplazamiento asociativo por conjuntos

Emplazamiento directo

En este tipo de cachés se asigna a cada bloque de memoria principal un solo bloque de caché en el que puede ubicarse. El ***bloque*** vendrá determinado por ciertos bits de la propia dirección. Al haber menos bloques en la caché que en la memoria principal**, habrá bloques de la MP que compitan por un mismo bloque de la MC**, ocasionando fallos de caché, como veremos más adelante. Por último, para saber si un bloque de memoria principal está en memoria caché se usará una serie de bits de la propia dirección para diferenciar unos de otros (***etiqueta***).



Ejemplo:

- Memoria principal de 64KB

$$64KB = 2^6 * 2^{10} = 2^{16}\ bytes \rightarrow 16\ bits\ de\ dirección\ de\ MP$$

- Memoria caché de 64B

$$64B = 2^6\ bytes \rightarrow 6\ bits\ de\ dirección\ de\ MC$$

- Tamaño de bloque de 16B

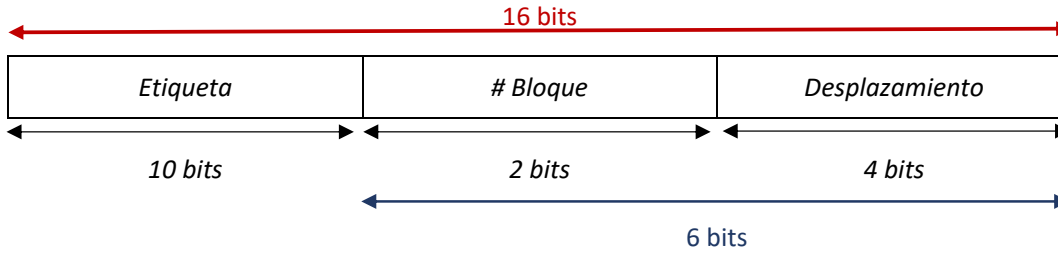
$$16B = 2^4\ bytes \rightarrow 4\ bits\ de\ desplazamiento\ dentro\ del\ bloque$$

- Bits de bloque

$$6\ bits - 4\ bits = 2\ bits\ de\ bloque$$

- Bits de etiqueta

$$16\ bits - 6\ bits = 10\ bits\ de\ etiqueta$$



Como se ve arriba, el formato de la dirección estará dado por:

1. **Los bits de dirección de la MP**. Son los bits totales que tendrá la dirección del procesador y es la suma de los bits de etiqueta + bloque + desplazamiento.
2. **Los bits de dirección de la MC**. Son los bits que lee la memoria caché para ubicar un dato. Es la suma de los bits de bloque + desplazamiento.
3. **Los bits de etiqueta**. Indica qué bloque de MP está ocupando ese bloque de la caché.
4. **Los bits de bloque**. Definen el número de bloque que le corresponde a la dirección en la MC.
5. **Los bits de desplazamiento**. Normalmente nunca se usan y definen el dato o palabra a consultar dentro del bloque.

Ejemplo de consulta

	Memoria Caché			
	# Bloque	Etiqueta	Posiciones Datos	
Bloque 1	00	00 0000 1111	…	
Bloque 2	**01**	**00 0000 1010**	0000	
			0001	
			0010	
			0011	**¡Acierto!**
			0100	
			…	
Bloque 3	10	01 0101 0101	…	
Bloque 4	11	11 1000 1110	…	**Fallo**

Al consultar esta dirección:

00 0000 1010	*01*	*0011*
Etiqueta	*Bloque*	*Desplazamiento*

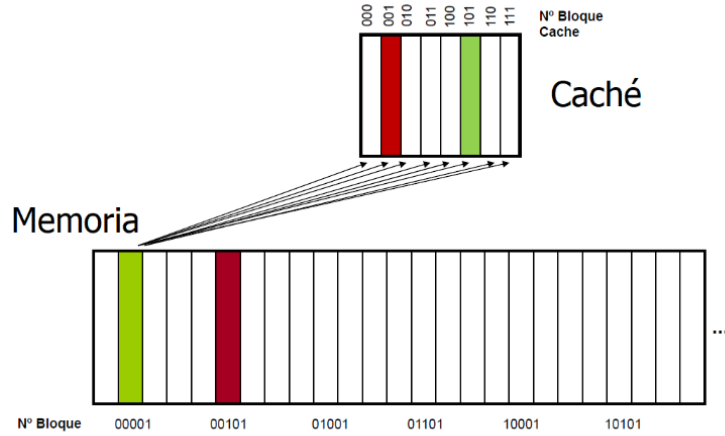
Se producirá un acierto ya que el bloque “01” de la caché está ocupado por la etiqueta “00 0000 1010”. Sin embargo, si queremos consultar la dirección:

11 1111 1111	*11*	*1111*
Etiqueta	*Bloque*	*Desplazamiento*

Como el bloque de caché “11” no contiene la etiqueta “11 1111 1111”, se producirá un fallo.

Emplazamiento completamente asociativo

En este tipo de emplazamiento no hay ninguna regla que diga dónde deben colocarse los bloques de memoria principal en la memoria caché. Por lo tanto, si hay huecos libres, se asignarán. Por el otro lado, si todos los bloques de la caché están llenos, se seguirán unas ciertas reglas para determinar **qué bloques deben salirse de la MC** *(políticas de reemplazamiento).*



Políticas de reemplazamiento:

- Aleatorio: se escoge un bloque aleatorio y se sustituye.
- FIFO: se quita el primer bloque en entrar (el más antiguo).
- LRU: se quita el bloque menos usado recientemente (*Least Recently Used*).
- LFU: se quita el bloque menos frecuentemente usado (*Least Frequently Used*).

Ejemplo:

- Memoria principal de 64KB

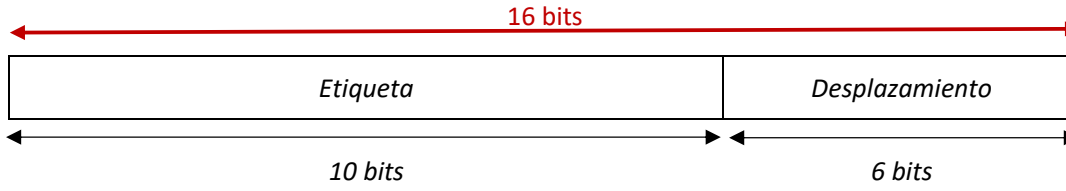
$$64KB = 2^6 * 2^{10} = 2^{16}\ bytes \rightarrow 16\ bits\ de\ dirección\ de\ MP$$

- Memoria caché de 64B

$$64B = 2^6\ bytes \rightarrow 6\ bits\ de\ dirección\ de\ MC$$

- Bits de desplazamiento = dirección MC = 6 bits
- Bits de etiqueta

$$16\ bits - 6\ bits = 10\ bits\ de\ etiqueta$$



El significado de cada término es el mismo que en el emplazamiento directo

Ejemplo de consulta

	Memoria Caché		
	Etiqueta	Posiciones Datos	
Bloque 1	00 0000 1111	…	
Bloque 2	**00 0000 1010**	00 0000	
		00 0001	
		00 0010	
		00 0011	**¡Acierto!**
		00 0100	
		…	
Bloque 3	01 0101 0101	…	
Bloque 4	11 1000 1110	…	

Al consultar esta dirección:

00 0000 1010	*00 0011*
Etiqueta	*Desplazamiento*

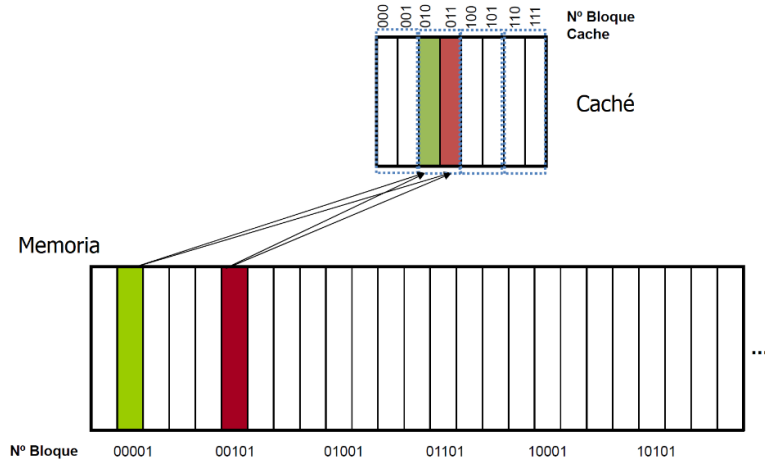
Se producirá un **acierto** ya que la etiqueta "00 0000 1010" existe en la memoria caché. Sin embargo, si queremos consultar la dirección:

11 1111 1111	*11 1111*
Etiqueta	*Desplazamiento*

Producirá un **fallo**, porque en TODA la memoria caché no hay ni un solo bloque con la etiqueta "11 1111 1111".

Emplazamiento asociativo por conjuntos

En este tipo de emplazamiento cada bloque de MP tiene asignado un **conjunto** de bloques fijo en MC, y dentro de ese conjunto puede ubicarse en cualquiera de los bloques que componen dicho conjunto. Es decir, es como una mezcla entre el **emplazamiento directo y el emplazamiento asociativo.** Las reglas para seleccionar qué bloque dentro de un conjunto debe eliminarse son las mismas que en el emplazamiento asociativo.



Ejemplo:

- Memoria principal de 64KB

$$64KB = 2^6 * 2^{10} = 2^{16}\ bytes \rightarrow 16\ bits\ de\ dirección\ de\ MP$$

- Memoria caché de 64B de dos vías (conjuntos de dos bloques)

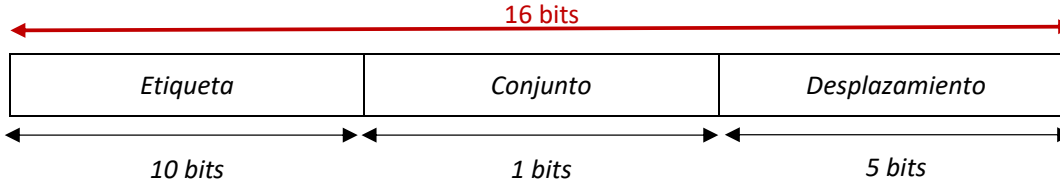
$$64B = 2^6\ bytes \rightarrow 6\ bits\ de\ dirección\ de\ MC$$

- Bits de conjunto: 1 bits (dos conjuntos: 0 y 1)
- Bits de desplazamiento (16B de tamaño de bloque):

$$16B = 2^4 \rightarrow 4\ bits + 1\ bit\ (el\ que\ sobra) = 5\ bits\ de\ desplazamiento\ dentro\ del\ bloque$$

- Bits de etiqueta

$$16\ bits - 6\ bits = 10\ bits\ de\ etiqueta$$



El significado de cada término es el mismo que en el emplazamiento directo

Ejemplo de consulta

Memoria Caché

Conjunto	Etiqueta	Posiciones Datos	
0	00 0000 1111	…	
	00 0000 1010	0 0000	
		0 0001	
		0 0010	
		0 0011	**¡Acierto!**
		0 0100	
		…	
1	01 0101 0101	…	
	11 1000 1110	…	

Al consultar esta dirección:

00 0000 1010	*0*	*0 0011*
Etiqueta	*Conjunto*	*Desplazamiento*

Se producirá un **acierto** ya que en el conjunto "0" existe la etiqueta "00 0000 1010". Sin embargo, si se consulta la dirección:

11 1111 1111	*1*	*1 1111*
Etiqueta	*Conjunto*	*Desplazamiento*

Producirá un **fallo**, porque en el conjunto "1" no hay ni un solo bloque con la etiqueta "11 1111 1111".

Calcular el número de fallos y optimizar código

En estos apartados el ejercicio suele dar un código (normalmente con un bucle) y te pide que cuentes el número total de fallos de caché que tiene.

Un fallo, como ya hemos visto**, se produce cuando intentamos leer un bloque de la MP que no se encuentra en la MC**. Al producirse un fallo en la memoria caché se va a buscar ese bloque correspondiente a la MP, se trae a la MC y se sustituye por el bloque correspondiente (según el tipo de emplazamiento).

Lo complicado de este tipo de ejercicios es saber manejarse con direcciones de memoria. Por ejemplo, te darán la dirección de comienzo de un array y a partir de él tienes que saber calcular su rango de direcciones. Con esto lo que se consigue es conocer **qué bloques de memoria ocupa el array** y con ellos ya es fácil ir sustituyéndolos uno a uno en la caché y contar los fallos que ocasionan.

Ejemplo

El problema nos da:

- Memoria Principal: 64KB
- Memoria caché con emplazamiento directo: 64B (6 bits de dirección de caché)
- Tamaño de bloque: 16B (4 bits de desplazamiento y (6 – 4) = 2 bits de bloque)
- Código:

```
for(int j = 0; j < 512; j++) {
        for(int i = 0; i < 8; i++) {
                A[i][j] += A[i][j] * 2;
        }
}
```

- Un array A[8][256] de enteros (1 entero = 4 bytes) que comienza en la dirección de memoria 0x0300.
- La memoria guarda las matrices por filas.

Y nos pide hallar cuántos fallos de caché se producen y en qué bloques al ejecutar el código asumiendo que esta se encuentra vacía al inicio de la ejecución.

El código explora una matriz por filas, algo que es muy ineficiente por lo que veremos ahora.

La matriz tiene 8 filas y cada fila tiene 256 enteros. Para poder averiguar cuántos bloques ocupa en memoria la matriz debemos pasar los enteros a bytes:

$$8 \text{ filas} * 256 \text{ columnas} * 4 \text{ bytes} = 2^{11} \text{ enteros} * 2^2 \text{ bytes} = 2^{13}\ bytes = 0x200\ bytes$$

Un bloque en memoria ocupa $16B = 2^4 bytes$:

$$\frac{2^{13}\ bytes}{2^4\ bytes\ por\ bloque} = 2^9 = 512\ bloques\ de\ memoria$$

Cada bloque de A ocupa $16B = 0x10\ bytes$, por lo tanto, si sumamos este número a la dirección de comienzo 512 veces y le restamos 1 bloque (1 menos porque el primer bloque cuenta) hallaremos todas las direcciones de los bloques de A:

	Dirección Hexadecimal	Dirección Binario	Número de bloque en caché
Bloque 0	0x0300	0000 0011 0000 0000	00
Bloque 1	0x0310	0000 0011 0001 0000	01
Bloque 2	0x0320	0000 0011 0010 0000	10
Bloque 3	0x0330	0000 0011 0011 0000	11
. . .	. . .	. . .	. . .
Bloque 511	0x04F0	0000 0100 1111 0000	11

Ahora que tenemos todos los números de bloque de cada dirección de A, volvemos al código. El código explora la matriz por filas, es decir, para la primera columna explora las 8 primeras filas, para la segunda columna otras 8, y así…

¿Entonces cuál es el problema? Pues que entre una fila y otra hay

$$256 * 4 = 1024\ bytes = 64 = 0x40\ bloques\ de\ memoria$$

Veamos este caso para la primera iteración de "j":

i = 0 → se consulta 0x0300 = 0000 0011 0000 0000

i = 1 → se consulta 0x0340 = 0000 0011 0100 0000

i = 2 → se consulta 0x0380 = 0000 0011 1000 0000

i = 3 → se consulta 0x03C0 = 0000 0011 1100 0000

i = 4 → se consulta 0x0400 = 0000 0100 0000 0000

i = 5 → se consulta 0x0440 = 0000 0100 0100 0000

i = 6 → se consulta 0x0480 = 0000 0100 1000 0000

i = 7 → se consulta 0x04C0 = 0000 0100 1100 0000

Es decir, que a todos los elementos de la primera columna le corresponden el mismo bloque pero con etiqueta **DISTINTA**. Es decir, se van a estar aplastando unos a otros cada vez que se intenten leer. Por lo tanto, como en cada iteración de "i" se produce un fallo y se realizan 256 * 8 = 2048 iteraciones, se producen 2048 fallos.

Hay que tener en cuenta que en cada iteración se realiza una lectura y una escritura sobre la matriz, en ese orden. La lectura es la que causa error, pero la escritura no (independientemente de la forma de escritura de esa caché).

¿Cómo podemos mejorar este código?

Si se intercambia el bucle de la "i" por el bucle de la "j", ahora en vez de leerse por filas se leerá por columnas, es decir, se leerán posiciones seguidas de memoria que se encuentran en un mismo bloque.

```
for(int i = 0; i < 512; i++) {
        for(int j = 0; j < 8; j++) {
                A[i][j] += A[i][j] * 2;
        }
}
```

Con este nuevo código se producirá dos fallos iniciales en cada iteración de la "i" para traer los dos bloques correspondientes a los 8 enteros que se quieren leer (recordemos que un bloque son 4 enteros) y el resto de lecturas (6) serán aciertos.

Esto reduce los fallos totales a 2 por iteración de la "i", es decir, 512 * 2 = 1024 fallos.

¿Otras formas de optimizar el código?

Dicho esto, hay otras formas de optimizar el código para otros tipos de ejercicios (para este en concreto no) donde nos den dos o más arrays:

- **Alargar arrays**. Consiste en alargar uno de los arrays (se supone que están todos seguidos en memoria) para que de esa forma se desalineen los bloques de memoria respecto a la caché. Ejemplo:

A y B alineados

Bloques de MP	Bloques de MC
A[0, 3]	00
A[4, 7]	01
A[8, 11]	10
A[12, 15]	11
B[0, 3]	00
B[4, 7]	01
B[8, 11]	10
B[12, 15]	11

A y B desalineados

Bloques de MP	Bloques de MC
A[0, 3]	00
A[4, 7]	01
A[8, 11]	10
A[12, 15]	11
A[16, 19]	00
B[0, 3]	01
B[4, 7]	10
B[8, 11]	11
B[12, 15]	00

- **Intercalar/fursionar arrays**. Consiste en coger los bloques de memoria de ambos arrays e intercalarlos. De esta forma también se desalinean los bloques de memoria de ambos arrays.

A y B alineados

Bloques de MP	Bloques de MC
A[0, 3]	00
A[4, 7]	01
A[8, 11]	10
A[12, 15]	11
B[0, 3]	00
B[4, 7]	01
B[8, 11]	10
B[12, 15]	11

A y B intercalados

Bloques de MP	Bloques de MC
A,B[0, 1]	00
A,B[2, 3]	01
A,B[4, 5]	10
A,B[6, 7]	11
A,B[8, 9]	00
A,B[10, 11]	01
A,B[12, 13]	10
A,B[14, 15]	11

*A, B[0,1] significa que en ese bloque conviven los elementos 0 y 1 del array A y B. Lo mismo para el resto de bloques.

Calcular el rendimiento

Esto son puras fórmulas:

- $T_{cpu} = (Nº\ de\ ciclos\ CPU + Nº\ ciclos\ de\ espera\ memoria) * T_c$
    - $Nº\ de\ ciclos\ de\ espera\ por\ la\ memoria = Nº\ de\ fallos * MissPenalty$
    - $Nº\ de\ fallos = Nº\ de\ referencias\ a\ memoria * MissRate$
    - $Nº\ de\ referencias\ a\ memoria = N * AMPI$
- $TMAM = HitTime + \ MissRate * MissPenalty$
- $PMPI = AMPI * MissRate * MissPenalty$

De las primeras expresiones se concluye que:

- $Nº\ de\ ciclos\ de\ espera\ por\ la\ memoria = N * AMPI * MissRate * MissPenalty$
- $Tiempo\ de\ CPU = N * \big(CPI + (AMPI * MissRate * MissPenalty)\big) * T_c$

Cada cosa significa:

- T_c = tiempo de ciclo
- N = número de instrucciones
- AMPI = media de accesos a memoria por instrucción
- MissRate = tasa de fallos = $\frac{fallos\ totales}{nº\ de\ accesos\ a\ memoria}$
- MissPenalty = tiempo de penalización por fallo (lo da el problema)
- TMAM = tiempo medio de acceso a memoria
- PMPI = penalización media por instrucción

Cachés con Pre-Búsqueda

Las cachés con pre-búsqueda al producir un fallo, además de traer el bloque que lo ha causado, se trae también el siguiente ya que es probable que se acceda a él por el principio de localidad temporal.

Permiten reducir la tasa de fallos.

Cachés víctimas

Es un tipo de caché pequeña completamente asociativa que es complementaria a la caché principal donde van a parar todos los bloques que han sido reemplazados de la caché principal.

Permiten reducir la tasa de fallos.

Cachés multinivel

Son cachés de varios niveles (L1, L2, L3… Li).

Permiten reducir la penalización por fallo ya que no hay que ir a buscar el dato a la MP.

$$TMAM\ caché\ 1\ nivel = HitTime + MissRate * MissPenalty$$

$$TMAM\ caché\ 2\ niveles = HitTime_{L1} + MissRate_{L1} * MissPenalty_{L1}$$

$$MissPenalty_{L1} = HitTime_{L2} + MissRate_{L2} * MissPenalty_{L2}$$

$$Tasa\ de\ fallos\ local\ en\ L_i = \frac{fallos * L_i}{nº\ de\ accesos\ a\ L_i}$$

$$Tasa\ de\ fallos\ global\ L_i = \frac{fallos\ L_i}{nº\ total\ de\ accesos\ a\ memoria}$$

Cachés segmentadas

Nunca caen xd. De esto no voy a hacer apuntes, si eso consultad los de vuestro profesor.