Universidad
Europea de Madrid

LAUREATE INTERNATIONAL UNIVERSITIES

MEMORIA

MEMORIA CACHÉ II

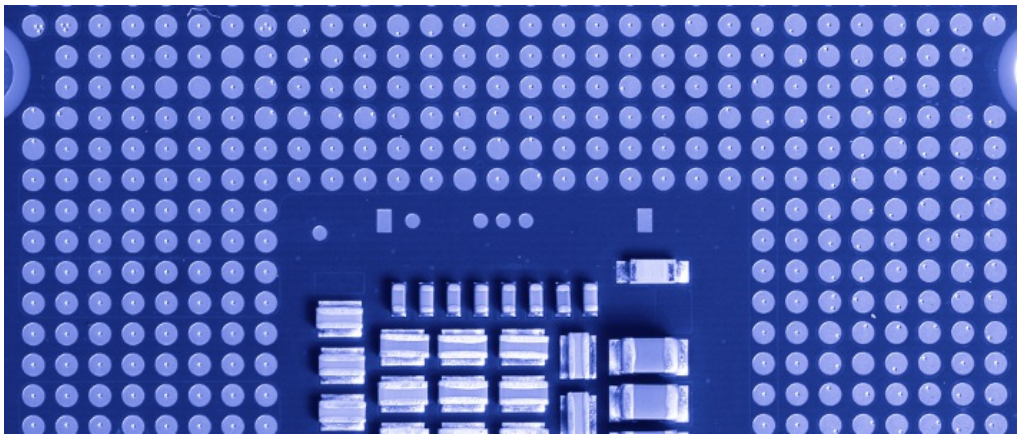
Índice

Presentación

En este tema terminamos con la caché. Veremos cómo mejorar la función de correspondencia directa y cuáles son los problemas que puede causar una memoria caché en un sistema informático.

- Ampliaremos las funciones de correspondencia con la correspondencia asociativa por conjuntos.
- Revisaremos la idea de jerarquía de caché y el porqué de su existencia.
- Descubriremos la necesidad de las políticas de reemplazo, que se encargan de determinar cuál y cuándo se elimina un dato de la caché.
- También veremos las políticas de escritura, que sirven para prevenir los problemas e incoherencia de caché.

Con este tema, damos por finalizada la memoria caché. Por lo que será necesario, al igual que en el tema anterior, hacer algún ejercicio para afianzar los conocimientos.



Problemas de la función de correspondencia directa

La implementación por hardware de la función de correspondencia directa es muy simple. Es más, no es un algoritmo en sí misma, sino una segmentación de las direcciones de memoria, lo que hace que la función sea **extremadamente rápida de implementar y tenga una efectividad muy alta**.

Sin embargo tiene un inconveniente muy grande: a cada dirección le corresponde una y solo una posición posible en la caché. No parece grave, pero piensa en tu agenda y considera que una de sus entradas solo pudiese contener un nombre. Los nombres *Andrés* y *Álvaro* no podrían estar simultáneamente en la agenda, ya que los dos consumen la *entrada A* de la agenda.

Este caso esta exagerado, ya que en la caché **no tenemos solo 28 entradas, sino que tenemos miles**. Es como si tuviésemos una agenda, en la que no ordenásemos por la primera letra, sino por las 2 o 3 primeras letras. Ahora si que podríamos tener a *Álvaro* y a *Andrés* simultáneamente en la agenda, ya que no solo se busca la *A*, sino que se ordena por *AL* y *AN*.

Pero si en vez de pensar en dichos nombres pensamos en *Fernando* y *Fernández* también tendríamos el mismo problema. Aunque tuviéramos una agenda de **2, 3 o incluso 7 caracteres como índice**, encontraríamos el mismo problema y, en tal caso, estaríamos hablando de una agenda de 28^7=13492928512 entradas. Si tuviésemos una agenda con tal cantidad de entradas, muchas de ellas (muchísimas de ellas) estarían sin utilizar, y aun así habría otras súper-utilizadas.

Aquí surge el segundo problema de una caché con demasiadas líneas: **pueden existir líneas de caché infra-utilizadas y también súper-utilizadas**. Con las súper-utilizadas no hay problema, simplemente producen fallos de caché, pero las infrautilizadas son un desperdicio, en coste y espacio, no admisible.

¿Queda claro el problema? En una caché con correspondencia directa, hay un grupo de direcciones de RAM a las que se les asigna una misma línea de caché. Puede ocurrir, y de hecho ocurre con mayor probabilidad de la deseada, que dos direcciones de caché luchen por la misma línea, lo que causa que estén provocando fallos de caché constantemente.

Así, aunque la función de correspondencia directa es una función muy utilizada, no es perfecta y puede ser mejorada, aunque para ello tendremos que buscar una nueva función de correspondencia.

Gran cantidad de entradas

¡¡Esto ya es una súper agenda!!

¿Cuántos de nosotros tienen una agenda con más de 13 millones de nombres?

Dos direcciones de caché luchen por la misma línea

Por el contrario, si la caché tiene muchas líneas, se reduce el número de direcciones de memoria que compiten por la misma línea de caché, pero entonces puede haber líneas de caché que no se utilizan o que se utilizan demasiado poco.

Función de correspondencia asociativa

Una caché que implementa una función de correspondencia asociativa lo que hace es, simplemente, **no tener una función que asigne una única posición de caché a cada dirección de memoria**. En cambio, a cada dirección de memoria le corresponde cualquier línea de caché.

Esta función elimina por completo el problema de la súper-utilización y de la infrautilización de líneas, aunque nos surge una pregunta: dado que al no haber un sistema de asignación de líneas no hay un sistema de búsqueda en caché:

¿Cómo hacemos para encontrar una dirección en la caché y determinar si hay acierto o fallo de caché?

La solución es **colocar un comparador en cada una de las líneas**, de manera que compruebe todas a la vez de manera simultánea. Con este método sí podríamos buscar el posible dato guardado de manera rápida determinando si hay acierto o fallo de caché y, en caso de acierto, leerlo para enviárselo al microprocesador.

> Esta es una caché muy eficiente, rápida y extremadamente cara. ¿Por qué? Por los comparadores que hay que poner en cada línea.

Supongamos una caché más o menos actual (la que podría llevar nuestro ordenador), de 4Mb, con un conjunto de vecindad de 32 vecinos. ¿Cuantas líneas tiene esta caché? Recordemos que el número de líneas viene determinado por **4Mb/32 = $2^{32}/2^5$ = 2^{27} ≈ 128** millones de comparadores.

Para colmo, al no haber línea, la etiqueta de la caché tendría que contener toda la información necesaria para determinar qué dato está guardado en la línea de caché, por lo que los bits de línea y etiqueta de la función de correspondencia directa se convierten íntegramente en etiqueta en la función de correspondencia asociativa.

Por tanto, esta caché tendría (32-5=27) bits en el campo etiqueta, con lo que tendría 128 millones de comparadores y de 27bits cada uno.

Esto no es mucho, es muchísimo. No se puede poner **128 millones de comparadores** en una caché o, al menos, en un espacio y precio aceptables, por lo que, a pesar de ser una caché con muchos beneficios, físicamente es muy difícil de implementar.

¿Acierto o fallo de caché?

Está claro que el sistema de la lista, o de ir comprobando cada línea, una por una, tardaría demasiado.

Además, el tiempo de respuesta dependería de dónde se ha guardado el dato, si en las primeras líneas de caché o en las últimas.

Función de correspondencia asociativa por conjuntos

Tenemos dos algoritmos para seleccionar una posición en caché del nuevo dato: la función de correspondencia directa y la función de correspondencia asociativa, cada una con sus ventajas e inconvenientes.

¿Qué tal si intentamos aprovechar lo bueno de cada una de ellas y al mismo tiempo minimizar las defectos?

Volvamos a nuestra archiconocida agenda para ver cómo solucionar el solapamiento de nombres y, al mismo tiempo, maximizar el uso de cada hoja.

La solución es simple: teniendo pocas entradas, 28, es suficiente utilizar una función de asignación directa como puede ser la primera letra del nombre, para que las búsquedas sean rápidas y, además, permitir que en una entrada de la agenda, se pueda guardar más de un dato.

Este sistema, conocido como **caché asociativa por conjuntos**, es el que se termina adoptando para la caché.

¿En qué consiste?

En tener varias cachés que trabajen en paralelo en vez de una directa. Por tanto, la función de asignación directa, determina la línea donde se guardara, pero no en qué replica de caché, lo que permite tener más de un valor por línea (un valor en cada copia de caché).

Al dividir la caché en varias iguales, reducimos el tamaño de cada una de ellas de manera individual, lo que reduce el número de líneas y provoca que sea más difícil que alguna línea de la caché se infrautilice.

¿Pero en cuantas ocurrencias dividimos nuestra caché?

De nuevo la respuesta no es simple, hay que llegar a un convenio entre número de líneas y número de entradas por línea. La respuesta la encontramos otra vez en la estadística.

En un sistema multiproceso (como son todos los sistemas operativos actuales), ¿Qué probabilidad hay de que dos procesos distintos luchen por la misma línea de caché? ¿Y de que sean tres, cuatro, etc.?

Cuando una caché se divide en 2, 4 o más cachés, a cada una de ellas se la conoce como *vía*. Ojo, no se aumenta el tamaño total de la caché según se aumenta el número de vías, sino que se divide el tamaño de cada caché entre el número de vías.

¿Y de que sean tres, cuatro, etc.?

En la siguiente tabla tenemos los resultados estadísticos que INTEL ha sacado de sus microprocesadores con un test Benchmark clásico:

Procesos que luchan por la misma línea	Probabilidad
Dos o más procesos	25%
Tres o más procesos	7%
Cuatro o más procesos	1%

Se pueda guardar más de un dato

Con este sistema, haremos agendas pequeñas y será raro encontrar alguna hoja vacía por no haber nombres para ella.

Jerarquía de caché

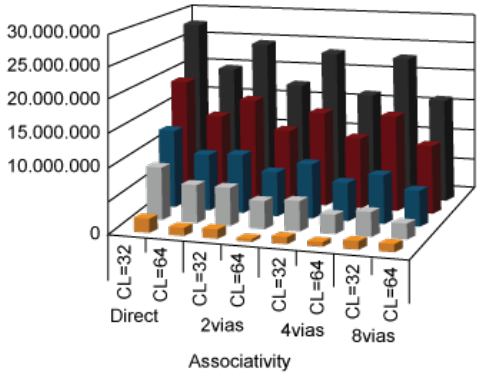
En la actualidad, la caché ha aumentado de manera exagerada el rendimiento de un procesador. Tanto que, respecto a ella se han hecho multitud de reformas. Lo más habitual en la actualidad, es disponer de una jerarquía de cachés.

La idea de jerarquía de cachés surge de los microprocesadores segmentados, que veremos en próximos temas. Entre las características de los mismos, se encuentra la de poder realizar muchas tareas a la vez, lo que origina que se realicen peticiones de datos e instrucciones de manera simultánea a la memoria.

La jerarquía de caché soluciona estos problemas, ya que:

- En su nivel más alto la caché está dividida en caché de datos y caché de instrucciones.
- En los niveles intermedios de la jerarquía se coloca una caché general para cada core de los procesadores multi-núcleo, lo que permite que cada uno de ellos tenga su propio banco de caché.
- En los niveles inferiores se coloca una caché compartida, lo que permite facilitar los datos compartidos y la división de trabajo con hilos (no procesos) de un mismo proceso (Intel tampoco comparte esta caché).

En términos generales, la caché de nivel superior, se puede ver como una caché de la caché de nivel inferior y así sucesivamente.

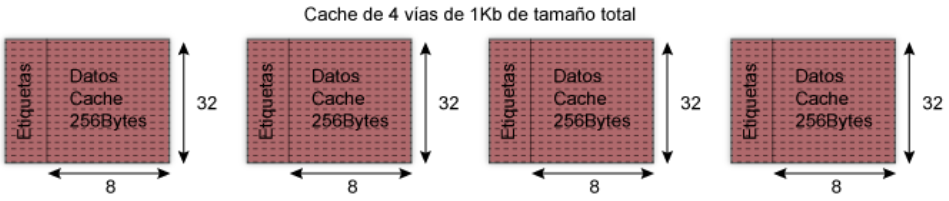


Durante este tema, y el anterior, estamos viendo las características generales de una caché, dejando de lado las peculiaridades de cada caché según su nivel. En la imagen adjunta, puedes comprobar el índice de fallos de caché originado por accesos a la RAM en diferentes configuraciones de INTEL.

Ejemplo de distribución de la caché

Ejemplo con la caché asociativa por conjuntos

Volvamos a nuestro ejemplo de caché de 1Kb con política de correspondencia directa y convirtámosla en una caché asociativa por conjunto de cuatro vías. Por lo tanto se convierte en una estructura como la de la siguiente imagen: cuatro cachés con correspondencia directa, donde cada una es 1/4 de la caché total.



Bits asignados
Memoria total = 20 bits
Tamaño de palabra = 3 bits
Conjunto = 5 bits
Etiqueta = 12 bits
Comprobación

Estudiemos los bits asignados a cada uno de los apartados: etiqueta, línea y palabra. El proceso de cálculo es el mismo que para la caché con correspondencia directa con una salvedad: la parte del MAR que se le llama *línea* ahora será llamada *conjunto*, ya que no representa una línea sino un conjunto de líneas (tantas como vías tenga la caché).

Si volvemos atrás veremos que, con respecto a la configuración de correspondencia directa, han cambiado los bits de conjunto (*línea*) y etiqueta.

MAR = 20 bits		
Etiqueta	Conjunto	Palabra
12	5	3

Comprobación

12+5+3=20, correcto.

Etiqueta

Memoria total entre tamaño de caché parcial: 1Mb/256b=$2^{20}/2^{8}=2^{12}$.

Por tanto, 12 bits.

Conjunto

Número de líneas de cada caché: la caché total es de 1Kb por lo que cada caché individual es de 1Kb/4= 256bytes. 256bytes=28 bytes.

Por tanto, el número de líneas es de $2^{8}/2^{3}=2^{5}$, es decir, 5 bits.

Memoria total

Recordemos que habíamos definido que se trataba de un microcontrolador con un tamaño máximo de memoria direccionable de 1Mb.

Por tanto el MAR es de (1Mb=2^{20}) 20 bits.

Tamaño de palabra

Representa el conjunto de vecindad. La caché define a 8 este conjunto, por tanto para palabra se necesitan (8=2^{3}) 3 bits.

Han cambiado los bits de conjunto y etiqueta

Es coherente, porque los cálculos para determinar el número de bits de cada división se realizan sobre el tamaño de una de las particiones de la caché. Y como esta es 1/4 de la original, el número de líneas disponibles es menor, necesitándose menos bits para direccionarla. Estos bits que perdemos del campo de línea, son ganados por el campo etiqueta y, por la misma razón, al comparar esta caché con el total de memoria del sistema, se necesitan menos bits para la etiqueta.

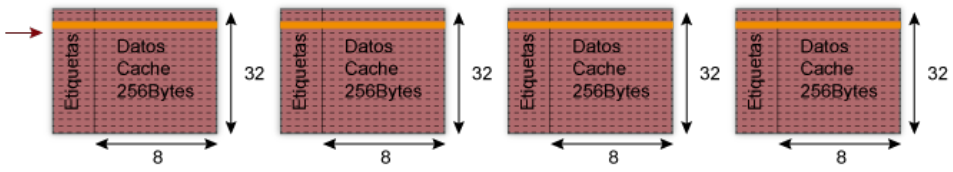
Problemas de la función de correspondencia asociativa por conjuntos

Si nos fijamos en el esquema de caché de la imagen anterior, y recordamos que las cachés funcionan en paralelo y que pueden seleccionar en que vía se guarda un dato, una vez seleccionada la línea correspondiente nos surge un problema.

¿Qué pasa cuando la caché está totalmente llena y hay que meter un nuevo elemento?

Para que el principio de vecindad funcione bien, es necesario que la caché siempre contenga los elementos más recientemente solicitados por el procesador, ya que estos tienen una mayor probabilidad de ser requeridos de nuevo. Por tanto, cuando la memoria caché está llena, cuando el procesador solicita un dato que no está en caché, se genera un fallo, por lo que el dato se trae desde memoria, pero también hay que guardarlo en la caché. Y aquí está el problema, para guardar el nuevo dato, hay que sacar de caché uno de los existentes.

¿Pero cuál? Para nuestro ejemplo podemos elegir entre cuatro (marcadas en color naranja), aunque de manera general podremos elegir entra tantos datos como vías tenga la caché.



Este es un problema grave, y para solucionarlo, se han desarrollado varios algoritmos que realizan la elección. A estos algoritmos se les conoce como *políticas de reemplazo* y, como en el caso de las funciones de correspondencia, tienen que ser rápidos, simples y exitosos.

Aunque no existe el algoritmo perfecto, hay otros algoritmos que, sin ser perfectos, obtienen unos buenos resultados, como son los siguientes:

Tantos datos como vías tenga la caché

Una mala elección puede provocar eliminar un dato que el procesador aún requiere o la permanencia en caché de un dato ya procesado que no volverá a ser requerido por el procesador, lo que en ambos casos generaría más fallos de caché.

- FIFO (First In, First Out).
- LFU (Least Frequency Used).
- LRU (Least Recently Used).

Algoritmos que realizan la elección

Son muchos los métodos que se desarrollan para este fin, y seguro que tú mismo puedes decidir alguno. Por ejemplo la política del aleatorio, es decir: "¿Cuál reemplazo? pues pito, pito, gorgorito,...", es fácil de implementar, el algoritmo resultante es rápido y, sorprendentemente, su eficiencia no es muy mala. Solo tiene un defecto: es poco científico y, sobre todo, su comportamiento es difícil de predecir.

El algoritmo perfecto

El algoritmo perfecto es fácil describir: aquel que consiga minimizar el número de fallos de caché, para lo que solo hay que reemplazar la línea de la vía que más tarde vaya a ser requerida por el procesador. Es decir, si, de los cuatro, eliminamos de la caché el elemento que más tarde va a ser requerido por el procesador, conseguimos retrasar lo más posible el fallo de caché, con lo que aumentamos la eficacia de la memoria.

Sin embargo, esta política tiene un defecto grave: que necesitamos conocer cuáles serán las futuras peticiones de datos por parte del procesador, y recordemos que la caché es autónoma, las decisiones las toma con la información de la que ella misma dispone.

Por tanto, esta política, a pesar de ser perfecta, no se puede resolver de manera computacional.

Políticas de reemplazo

Como antes hemos visto, las políticas de reemplazo se encargan de determinar qué elemento de la lista se eliminará para colocar un nuevo dato.

Estas solo tienen sentido cuando se tiene dónde elegir. Por ejemplo, para la política de correspondencia directa no se habla de políticas de reemplazo, ya que hay una, y solo una, posición valida, por lo que hay un, y solo un, dato a ser eliminado, poco podemos decidir.

Pero para la función de correspondencia asociativa por conjuntos, podemos elegir entre tantos datos como vías tenga nuestra caché. Así que recordemos que los métodos más comunes son los siguientes.

FIFO (First In, First Out)	Es un método muy rápido, ya que la caché solo tiene que guardar en qué orden fueron cargadas las líneas de modo que, al necesitar hacer espacio, pueda elegir fácilmente la primera línea cargada. Aunque las técnicas FIFO son simples e intuitivas, no se comportan de manera aceptable en la aplicación práctica, por lo que es raro su uso en su forma simple. Su principal problema consiste en que podemos quitar de caché una línea de memoria muy usada solo porque es la más antigua.

LFU (Least Frequently Used)	Se sustituye el bloque que ha experimentado menos referencias en un intervalo corto de tiempo. Es muy eficiente, pero necesitará un contador de referencias por cada línea. Si un comparador era muy caro para implementar la función de correspondencia Asociativa, imaginaros el coste de un contador. Además, el tiempo de reset del contador es crítico, ya que podemos dejar algunas referencias en caché, solo porque han sido muy utilizadas en el pasado, penalizando otras más nuevas y con mayor uso en el futuro. Por ejemplo el índice de un bucle *for.* Este índice (variable) tiene muchas referencias, pero terminado el *for*, no se vuelverá a necesitar.
LRU (Least Recently Used)	Solo determina cuáles son las referencias que llevan mas tiempo sin ser utilizadas. No tiene en cuenta ni el orden en ser insertadas en caché, ni las veces que se han usado, solo cuál o cuáles han sido los últimos datos de un conjunto en ser utilizados. A pesar de su poca fiabilidad, este es el método más fácil de implementar. Y tiene un índice de acierto muy bueno.

LRU (Least Recently Used)

Este es el método que utilizan casi todas las cachés actuales, ya que su implementación es simple y su efectividad muy alta.

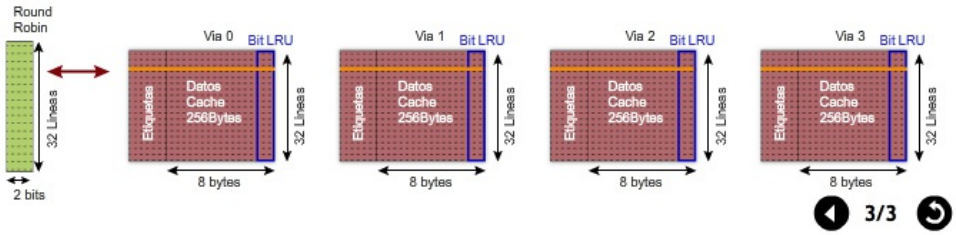
¿Cómo funciona? Hay dos variantes, para cachés de dos días, y para más de dos vías.

¿Cómo funciona?

Cada vez que se necesite reemplazar un elemento de una vía, se empieza a mirar en la siguiente vía que está indicada en la columna Round-Robin.

- Si esta tiene el bit LRU a 0, se reemplaza, se pone dicho bit a 1, se actualizan los bits del Round-Robin a la nueva vía chequeada y se termina el algoritmo.
- Si tiene el bit LRU a 1, entonces se pone a 0, y se comprueba la siguiente vía. Así hasta que se encuentre un LRU a 0.
- En caso de que todos estén a 1, como el chequeo de vías es cíclico, al llegar a la última empezamos en la primera. Después de dar una vuelta completa, encontraremos a 0 la primera vía que hemos chequeado.

En la columna Round-Robin queda anotado el número de vía que ha sido reemplazado, para que en la próxima iteración se empiece en la siguiente vía.



Problemas con el reemplazamiento

Ya todo parece que está perfecto. Gracias a las funciones de correspondencia sabemos dónde meter un nuevo elemento en la caché para que luego sea fácil de encontrar. Cuando hay varios elementos sobre los que podemos reemplazar, tenemos políticas de remplazo que determinan cual es el elemento más conveniente.

Pero se nos ha olvidado una cosa: el principio de funcionamiento de la caché, es que no tiene datos propios, sino copia de datos que existen en RAM. Por lo tanto:

¿Puede ocurrir alguna vez que el valor de una variable almacenada en RAM y su copia en caché no tengan el mismo valor?

La respuesta es sí, y además es muy sencillo. Imaginémonos un código en C tan simple como var=var+1. Esta instrucción en código maquina se traduce en: cargar el valor de var en el registro AX, incrementar AX en una unidad y guardar AX en la dirección de memoria var.

Como consecuencia, **se producen dos accesos a la variable var, uno en lectura y otro en escritura.**

La primera vez que se accede a var, tal vez haya fallo de caché, por lo que se va a memoria para traer su valor. Este se guarda en caché, junto con su conjunto de vecindad, pero luego se escribe el valor de var con un nuevo valor.

Ahora sí que la variable var esta en caché, ya que se ha cargado en el fallo anterior, por lo que hay un acierto de caché y el nuevo valor solo se actualiza en caché, quedando anticuado el valor de var en memoria.

A este efecto se le llama incoherencia de caché y es uno de los problemas más graves que puede tener la caché. En futuros temas, trataremos cómo prevenirlo.

Solucionado o no el problema, ahora mismo tenemos un error que hay que solucionar o todo nuestro sistema de caché fallará y se hará inservible.

1/3

Otra vez más, tenemos que dotar a la caché de algoritmos que eviten estos problemas, algoritmos que llamaremos políticas de escritura. Solo hay dos maneras de arreglar esto, con la política de write back y write thru.

Write through	Write back
Las operaciones de escritura en caché, siempre provocan fallo de caché, por lo que la orden de escritura siempre llega a la RAM, no pudiéndose producir nunca incoherencia de caché. Este es un método seguro pero poco eficiente, ya que cerca del 50% de los accesos a memoria para datos son en escritura, lo que reduce muchísimo la eficiencia de la caché.	La escritura es a demanda, justo antes de reemplazar una línea se comprueba que algún dato almacenado en la línea no está modificado, en caso de que este modificado se escribe todo el bloque en memoria de nuevo y luego se procede a eliminarlo. Es el método más utilizado, pues aunque retrasa las operaciones de reemplazo, optimiza mucho los accesos a RAM.

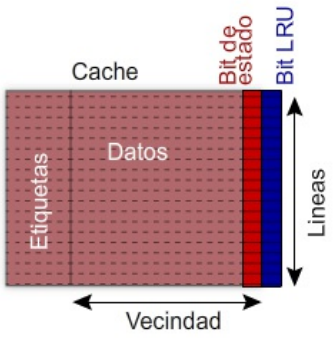
2/3

¿Pero cómo detecta la caché que un dato esta modificado?

No se lo va a preguntar a la RAM, pues para eso ya lo escribe, y se ahorra un paso. La idea es que se marquen de alguna manera las líneas modificadas, para que sean fácilmente comprobables.

La solución pasa por poner más información extra a la caché, tal como se aprecia en la imagen, marcados en rojo se encuentran los bits de estado. Aquí es donde se le pone información extra a cada una de las líneas, como el bit de modificado, bit de vacío, bit de LRU, etc...

En nuestro caso, para determinar si hay incoherencia de caché, se comprueba el bit de modificado, que se pondrá a 1 cada vez que se acceda a un dato de caché en modo de escritura.



3/3

Resumen

Con este tema hemos finalizado el tema de la memoria caché, pero no el de la jerarquía de memoria, como comprobaremos en el tema siguiente.

Para terminar con la caché, además de con la función de correspondencia, hemos ampliado nuestros conocimientos con los siguientes puntos:

- Problemas de la función de correspondencia directa.
- Función de correspondencia asociativa por conjuntos.
- Jerarquía de caché.
- Políticas de reemplazo.
  - LRU (Least Recently Used).
- Problemas con el reemplazamiento.

Ahora sí tenemos una visión de la caché mucho más exacta y en concordancia con cómo se implementa hoy en procesadores actuales, como por ejemplo en los de Intel, tal como se aprecia en una de las pantallas anteriores.