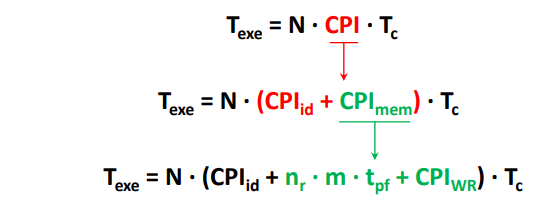
**Tema 5: conceptos avanzados Memoria Cache**

**Introducción**

Cualquier optimización a realizar tiene como objetivo final reducir el tiempo de ejecución:



Hay elementos que dependen del lenguaje máquina (N, nr) en los cuales no influye la jerarquía de memoria.

La jerarquía de memoria puede influir en:

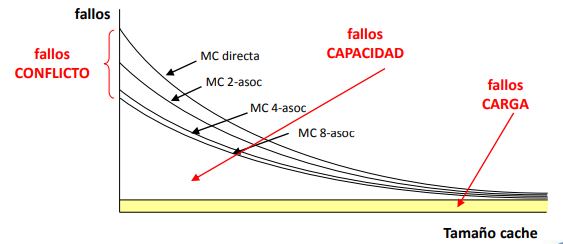
* Tasa de fallos, m (el objetivo es disminuirlo).
* Tiempo de penalización por fallos, tpf (el objetivo es disminuirlo).
* Cose de las escrituras, CPIWR (el objetivo es disminuirlo).
* Ancho de banda con memoria.

Clasificación de los fallos de cache (3C model):

Los fallos de cache pueden dividirse en tres categorías:

1. Carga (compulsory): se producen la primera vez que se accede a una posición de memoria.
2. Capacidad: todas las líneas que necesita un programa no caben en la memoria cache.
3. Conflicto: se producen cuando varios bloques se mapean en el mismo lugar de la MC (sólo en MC directas y asociativas por conjuntos).

(En los multiprocesadores aparecen los fallos de coherencia).



Técnicas básicas para mejorar el rendimiento de la cache:

* Aumentar el tamaño de bloque (m ↓): reduce los fallos de carga, pero puede ser contraproducente (m ↑).
* Aumentar el tamaño de la cache (m ↓): reduce los fallos de capacidad (y conflicto), pero aumenta el tiempo de acceso a la cache (tsa↑) y el consumo (W ↑).
* Aumentar el grado de asociatividad (m ↓): reduce los fallos de conflicto, pero aumenta el tiempo de acceso a la cache (tsa↑).
* Caches multinivel (tpf ↓): L1 pequeña (m ↑ y tsa ↓) y L2 grande (m ↓ y tsa ↑).
* Dar más prioridad a las lecturas que a las escrituras (CPIWR ↓).

El coste de las escrituras se puede reducir (ocultar) utilizando buffers de escrituras.

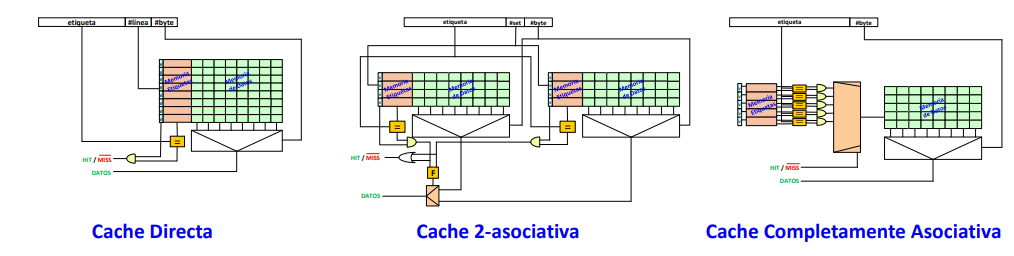
Técnicas avanzadas para mejorar el rendimiento de la cache:

* Reducir el coste de un acierto en cache (tsa↓): caches pequeñas y simples, predicción de vía y traces caches.
* Aumentar el ancho de banda de cache (W ↑): caches segmentadas, caches multi-banco y caches no bloqueantes.
* Reducir el coste de los fallos (tpf ↓): early restart y merging write buffers.
* Reducir la tasa de fallos (m ↓): optimizaciones del compilador.
* Reducir el coste de los fallos (tpf ↓) y la tasa de fallos (m ↓) vía paralelismos: pre-búsqueda hardware y pre-búsqueda software.

**Optimizaciones**

1.Caches pequeñas y simples para reducir el tiempo de acceso en acierto:

El tiempo de acceso a la cache depende del camino crítico:



Cache directa: acceso (memoria datos y etiquetas), comparar etiquetas y validar línea.

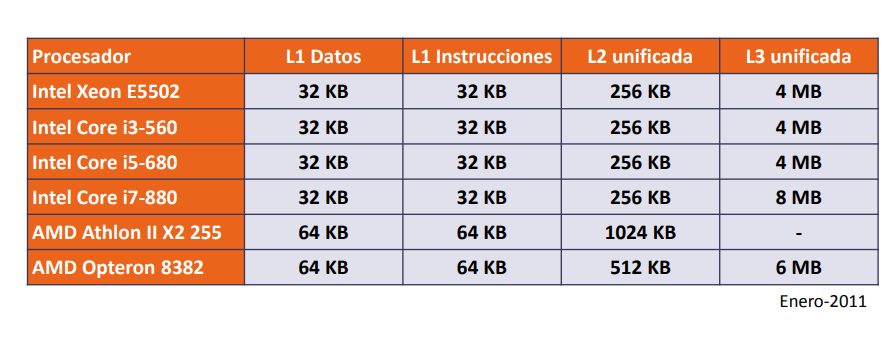
Cache asociativa por conjuntos: acceso (memoria datos y etiquetas), comparar etiqueta, validar línea y seleccionar vía.

Cache completamente asociativa: acceso (memoria etiquetas), comparar etiqueta, validar línea y acceso (memoria datos).

El tiempo de acceso a una cache directa es menor que el de una cache asociativa de capacidad equivalente.

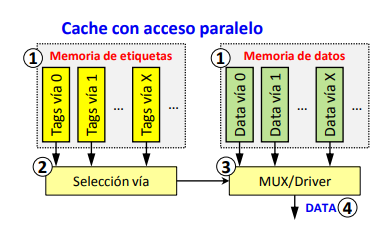
El tiempo de acceso a la cache depende del tamaño de la cache (a mayor capacidad aumenta el tamaño del array de memoria, de los decodificadores, …).

La mayoría de las caches de primer nivel de los procesadores actuales son pequeñas:

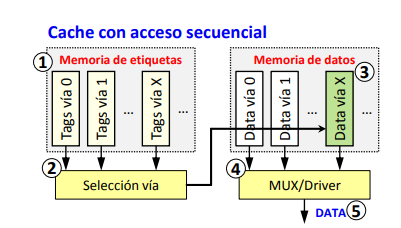


2.Predicción de vía para reducir el tiempo de acceso en acierto:

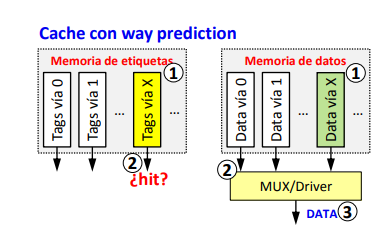
En una cache con acceso en paralelo, el consumo es elevado. El tiempo de acceso viene determinado por el acceso a memoria y la selección de la vía.



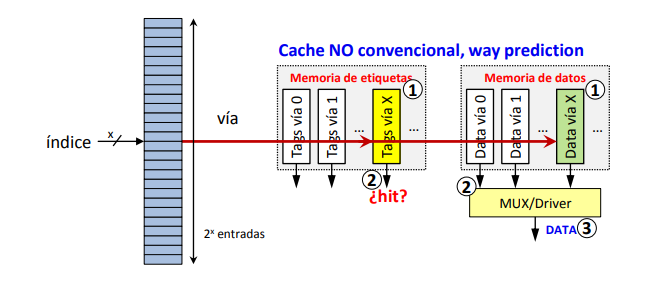
Una cache con acceso secuencial es lenta, pero reduce sustancialmente el consumo porque sólo accede a los datos de la vía seleccionada.



En una cache con way prediction, el consumo se reduce porque sólo accedemos a la vía indicada por la predicción y reduce el tiempo de acceso en caso de acierto, porque no necesita la selección de vía.

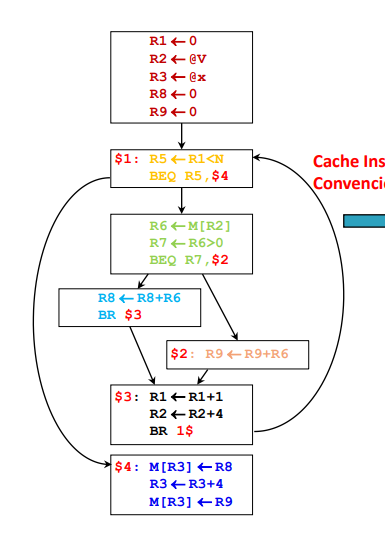


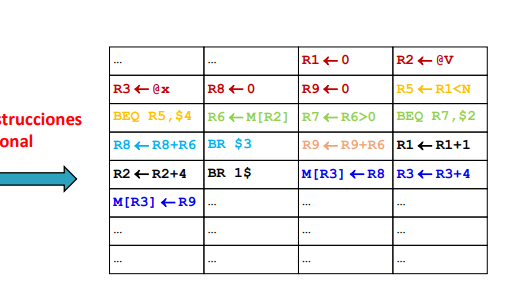
¿Cómo se realiza la predicción?

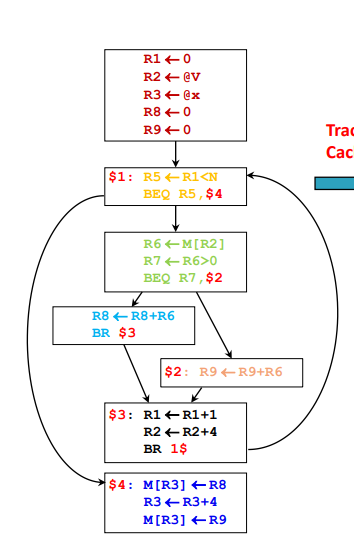


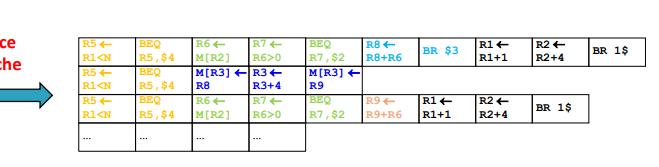
El índice podría ser el PC de la instrucción en ejecución, la tabla sería demasiado grande. Podemos utilizar unos pocos bits del PC (como en una cache). La tasa de aciertos en la predicción depende (entre otras cosas) del tamaño de la tabla (2x). Los programas tienen una << ejecución predecible >> (localidad). Y la predicción se actualiza con el comportamiento de los accesos previos.

3.Trace caches para reducir el tiempo de acceso en acierto:

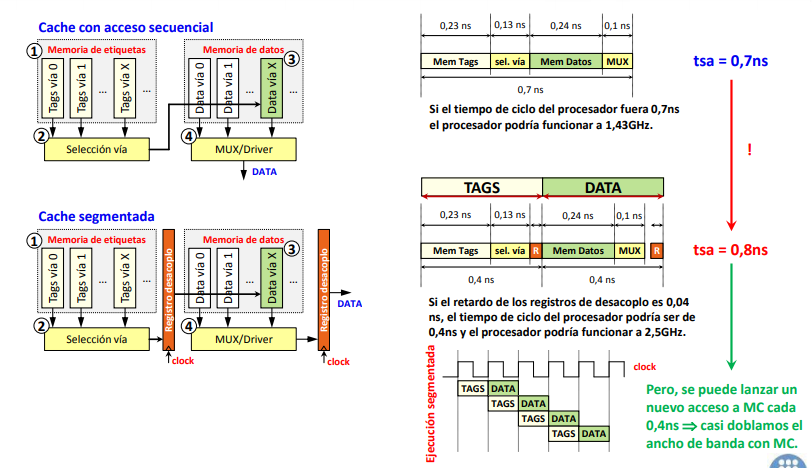
En una cache de instrucciones convencional, estas se almacenan en función de su dirección. La propia sintaxis del código hace que la localidad espacial no se aproveche.

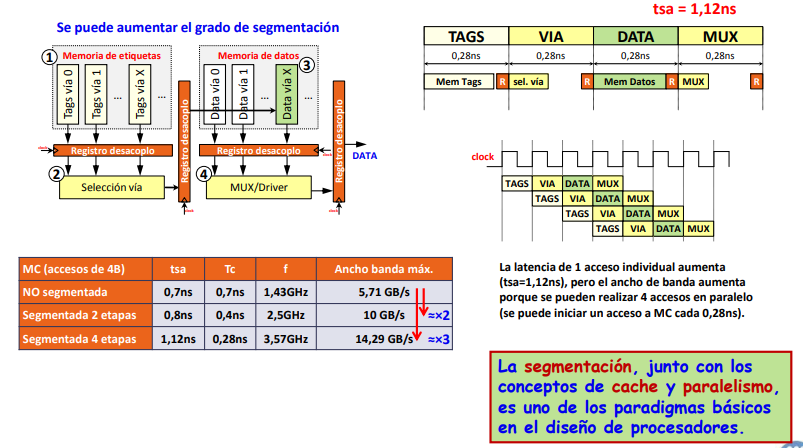


En una trace cache se almacenan secuencias dinámicas de ejecución de instrucciones. El predictor de trazas (saltos) es el encargado de seleccionar la traza a ejecutar (si acierta se aprovecha al máximo la localidad espacial). En la trace cache se puede guardar información adicional para acelerar la ejecución de instrucciones.

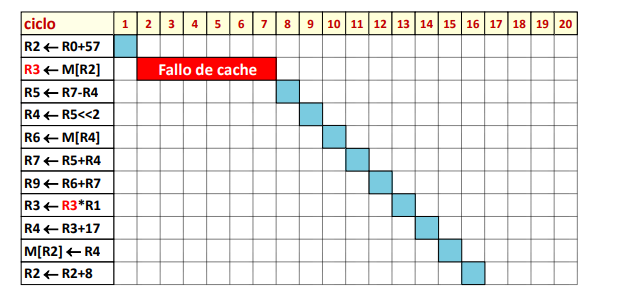


4.Caches segmentadas para aumentar el ancho de banda de la cache

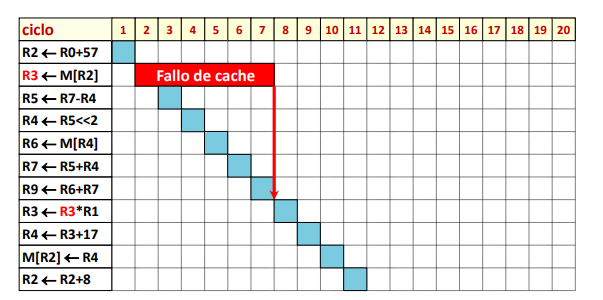


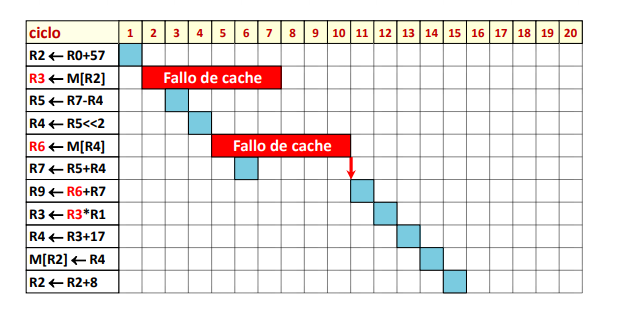


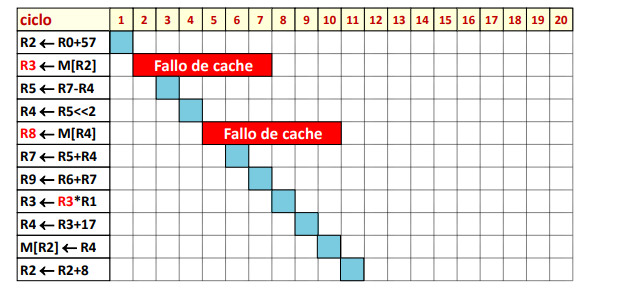
5.Non Blocking Caches:



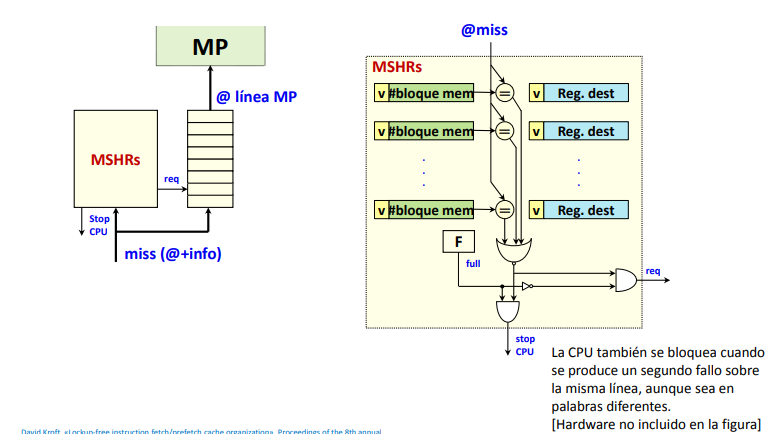
Hasta ahora, cuando se produce un fallo de cache, el procesador no inicia la ejecución de nuevas instrucciones hasta que se resuelve el fallo. En realidad, el procesador podría seguir ejecutando instrucciones mientras que no necesite R3.

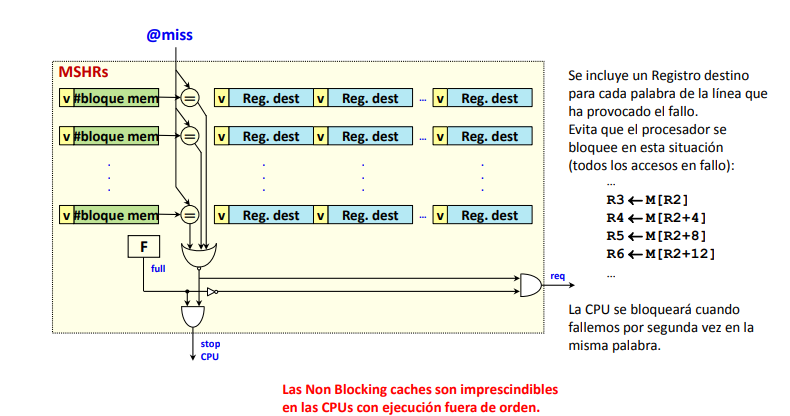


En una Non Blocking Cache, cuando se produce un fallo de cache el procesador continúa la ejecución de instrucciones y sólo se detiene cuando necesita el dato que ha provocado el fallo de cache.

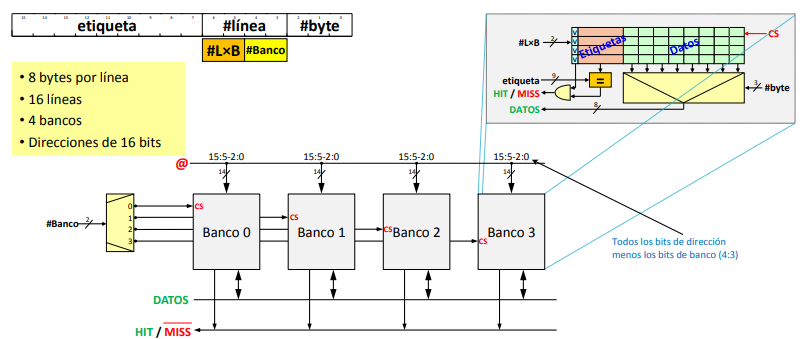


Una Non Blocking Cache usa MSHRs (Miss Status Handler Register) para gestionar los fallos pendientes. El número de MSHRs condiciona el número de fallos que puede soportar la MC sin detener el procesador. La idea original de los MSHRs es que el compilador / programador separe lo suficiente los accesos a memoria (R3 <= M[R2]) de su uso (R3 <= R3\*R1) para que, en caso de fallo de cache, el procesador no se detenga. El segundo fallo (R8 <= M[R4]) no da problemas porque R8 no se usa próximamente, pero si se usase, sería preciso mantener información del registro destino de 2 fallos de cache. Los MSHRs mantienen información de cual es el registro destino del load para controlar cuando se ha de bloquear el procesador.





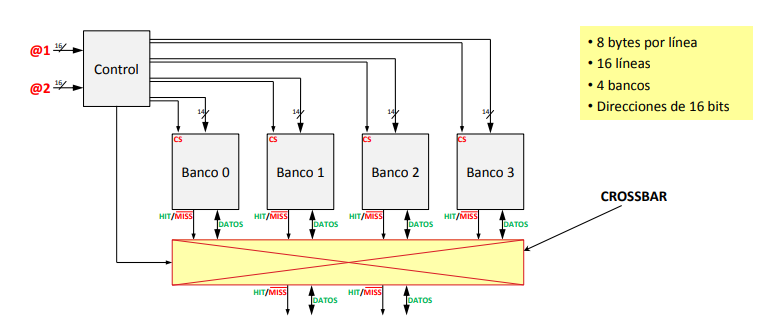
6.Caches multibanco:



Esta organización permite:

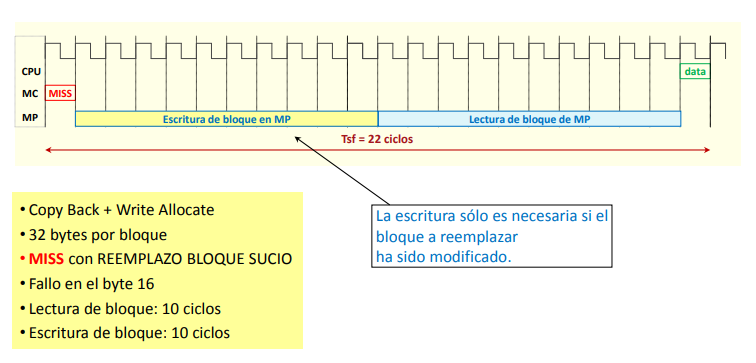
* Reducir consumo, sólo es necesario activar el banco al que se accede.
* Realizar accesos concurrentes (solo si van a bancos diferentes).

2 accesos simultáneos:

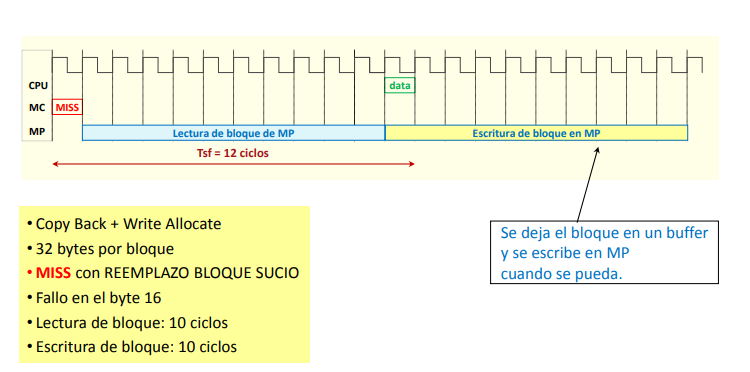


7.Reducir la penalización por fallo: early restart, transferencia en desorden:

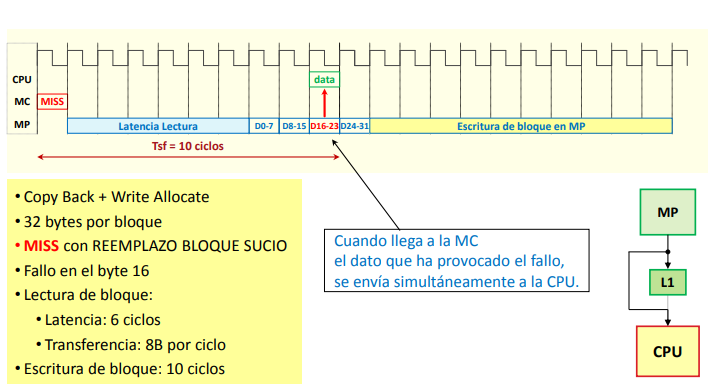
Punto de partida:



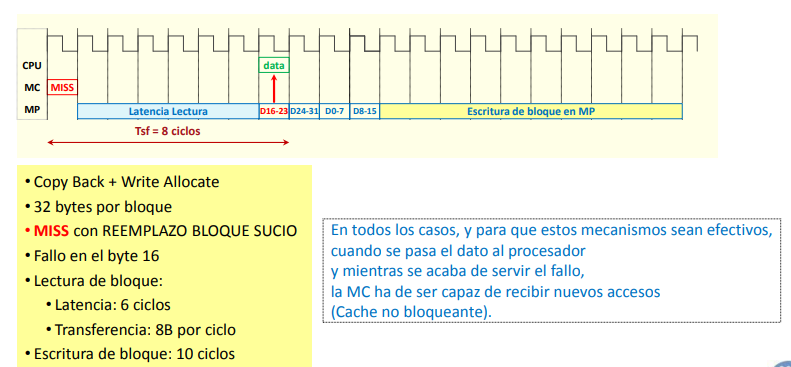
Actualizar MP después de leer línea:

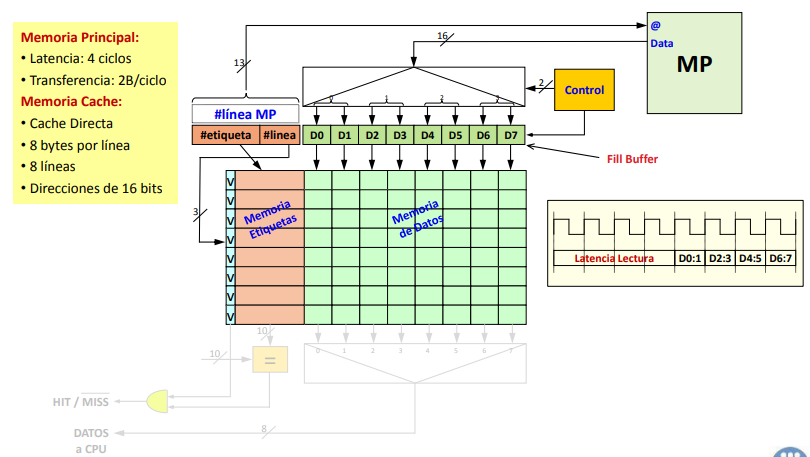


Continuación anticipada (Early restart): en cuanto se llega el dato que ha provocado el fallo, se envía al procesador:



Transferencia en desorden + continuación anticipada: se envía en primer lugar el dato que ha provocado el fallo:



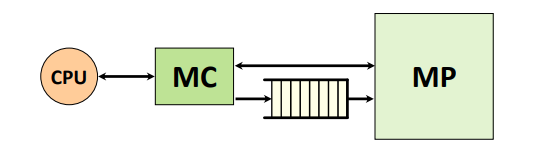


8.Buffers de escritura:

Si la cache es COPY BACK, para reducir la penalización en caso de fallo hay que dar prioridad a leer el bloque que contiene el dato que provoca el fallo a la escritura en MP del bloque reemplazado.

Solución: poner un buffer de n entradas entre la MC y MP:

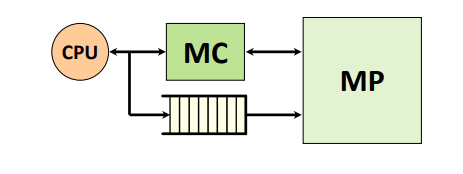
* Cuando se reemplaza una línea sucia, se deja el bloque en el buffer.
* El acceso al buffer es rápido (equivale a acceder a MC, 1 ciclo).
* Las escrituras se realizan cuando el bus entre MC y MP no está ocupado.
* Unas pocas entradas son suficientes.
* Cuando se realiza un acceso a MC, también hay que consultar en el buffer los bloques pendientes de escribir.



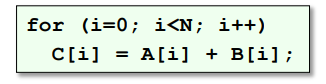
Si la cache es WRITE THORUGH, el coste de una escritura es el coste de escribir en Memoria Principal (no es aceptable).

Solución: poner un buffer de n entradas entre la CPU y MP:

* Las escrituras se almacenan en el buffer.
* El acceso a buffer es rápido (equivale a acceder a MC, 1 ciclo).
* Las escrituras se realizan cuando el bus entre MC y MP no está ocupado.
* Unas pocas entradas son suficientes.
* Cuando se realiza un acceso a MC, también hay que consultar en el buffer de datos pendientes de escribir.

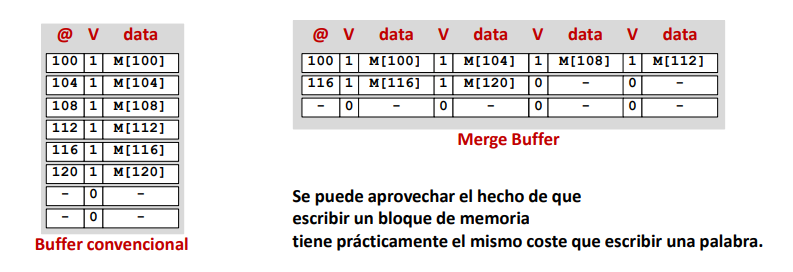


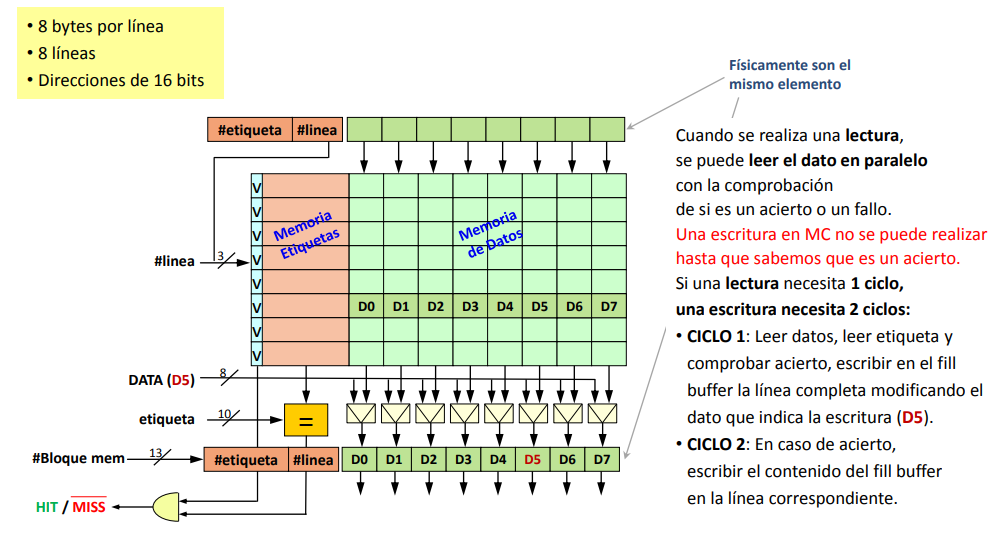
Situación común:



Cuando realizamos escrituras con localidad espacial, el buffer se llenará muy rápido y será poco eficiente. Además, las estructuras van a parar a la misma línea de cache.

Solución: Merge Buffers:



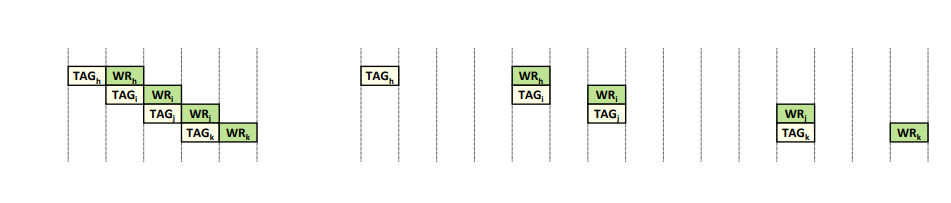


Solución: estructuras segmentadas:

En el primer ciclo se comprueba si es acierto o fallo y, en caso de acierto, se deja el dato a escribir en un registro intermedio.

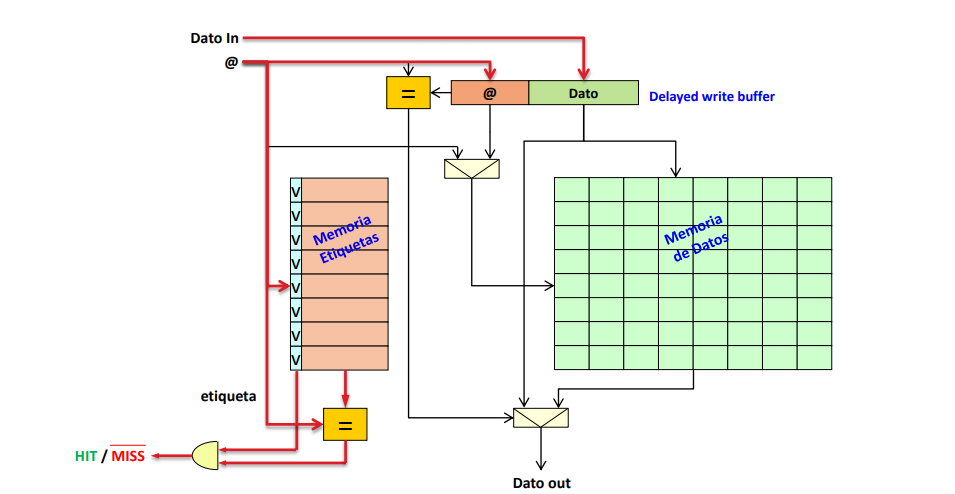
En la siguiente escritura (mientras se comprueba si es acierto o fallo) se realiza la escritura anterior.

Una escritura individual sigue tardando 2 ciclos, pero desde el punto de vista del procesador solo cuesta un ciclo.



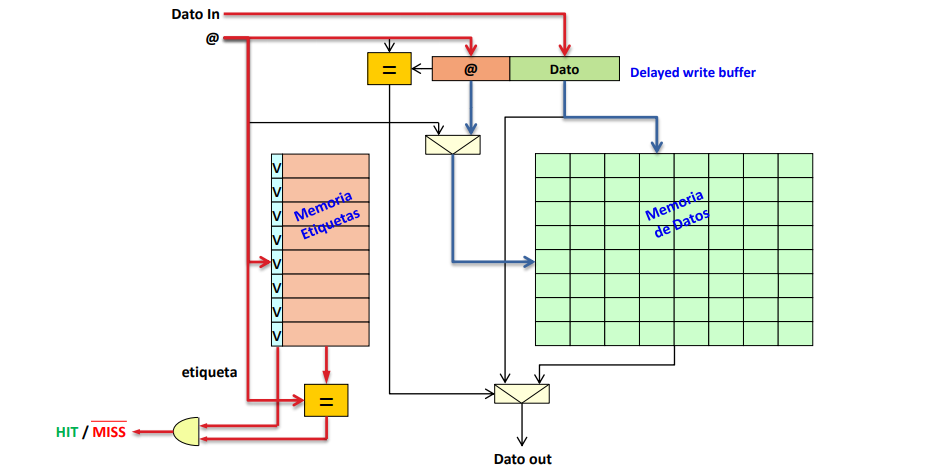
ESCRITURA DATO EN MC:

1er ciclo escritura: comprobación acierto:

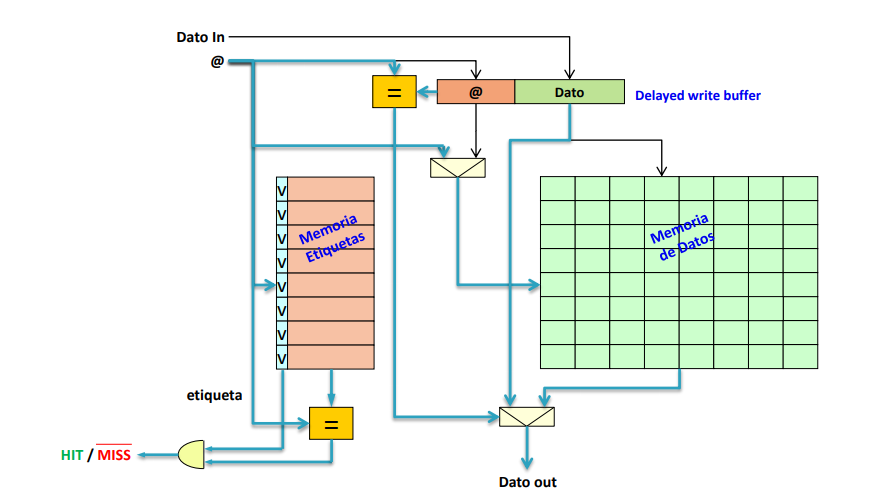


1er ciclo escritura siguiente: comprobación de acierto.

2do ciclo escritura anterior: escritura en memoria datos (si el ciclo 1 fue hit):



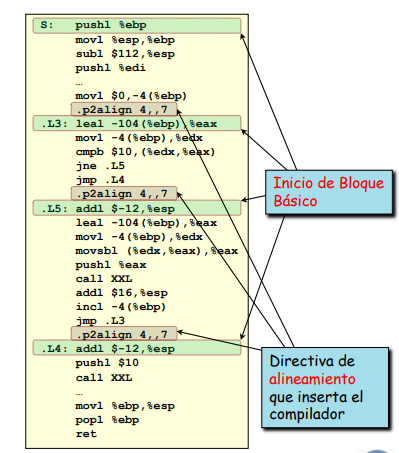
Lectura en paralelo (con comprobación del dato en el buffer):



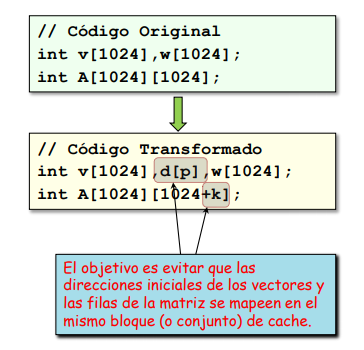
9.Optimizaciones de código para reducir tasa de fallos:

Reordenación de código:

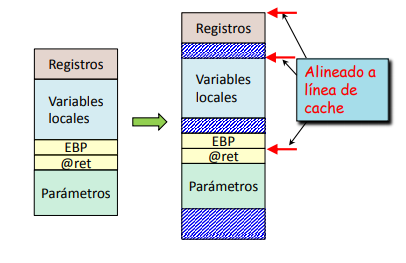
* Alineando los puntos de entrada de los bloques básicos con el inicio de la línea de cache, aumenta la probabilidad de acierto en cache para código secuencial.
* Si el compilador considera que un salto condicional se comportará la mayoría de las veces en el mismo sentido, puede organizar el código para que, en ese caso, el código se ejecute en secuencia.



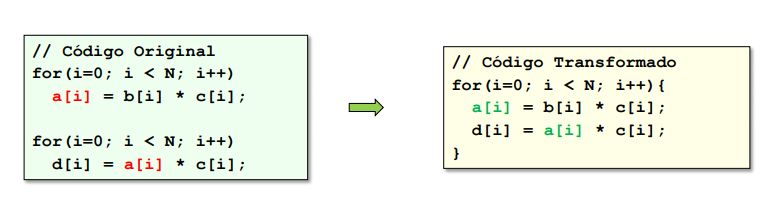
* Los datos pueden ubicarse en memoria para evitar conflictos.



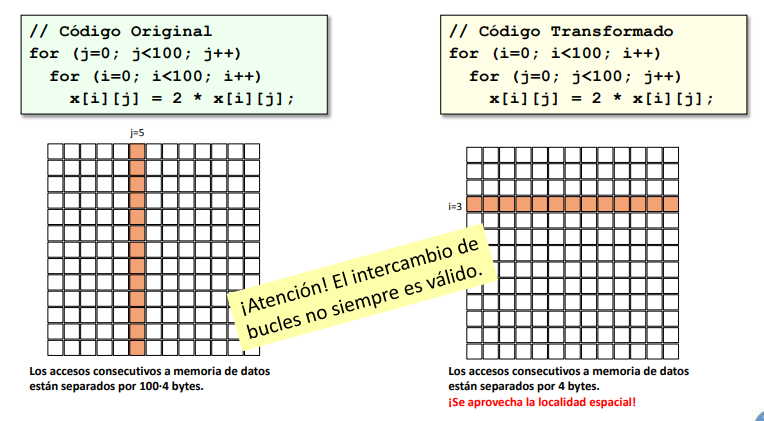
* Las diferentes partes del bloque de activación de una subrutina pueden alinearse con el inicio de la línea de cache para aprovechar la localidad espacial.



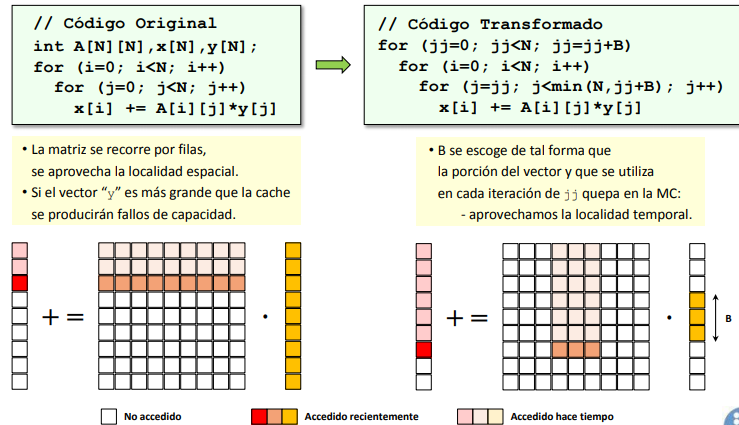
Loop Interchange (intercambio de bucles):

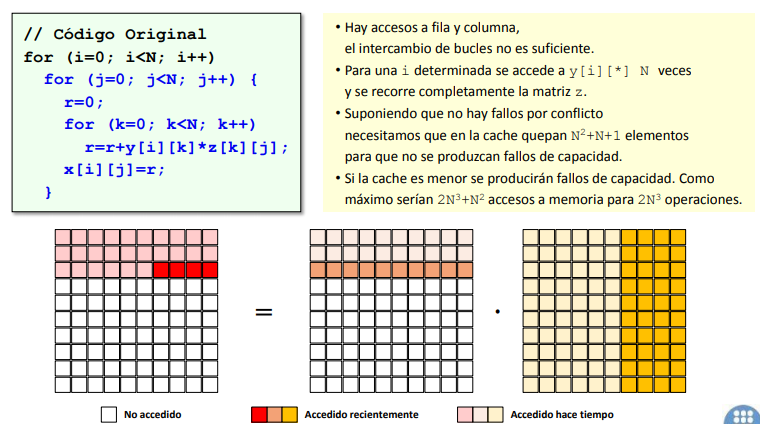


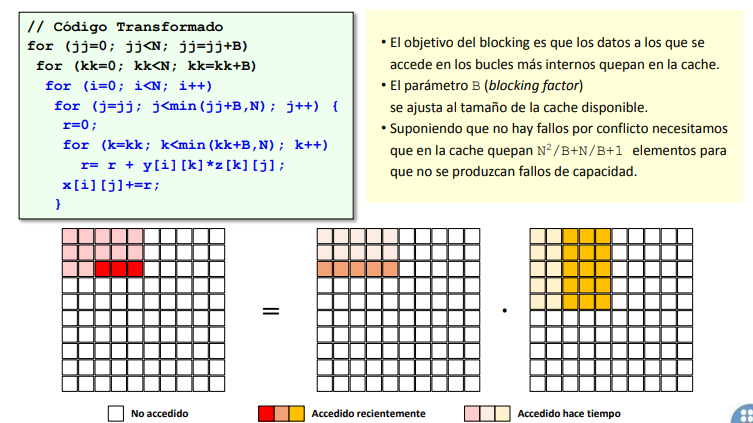
Aprovecha localidad temporal.



Blocking (MxV):







La memoria cache es transparente al programador. Sin embargo, tenerla en cuenta en la programación puede mejorar (a veces espectacularmente) el rendimiento de los programas.

Existen numerosas técnicas de programación / compilación para optimizar el rendimiento de la jerarquía de memoria.

Las optimizaciones se hacen pensando en todos los niveles de la jerarquía:

* Registros del procesador.
* Memoria Cache.
* TLB.

10.Prefetch de instrucciones y/datos para reducir tasa de fallos y/o la penalización por fallo:

Objetivo: reducir los fallos de CARGA.

Estrategia: especulamos con la localidad y traemos a MC aquella información que creemos que será utilizada en un futuro cercano, antes de que sea solicitada.

* Los accesos a instrucciones son más sencillos de predecir que los accesos a datos.
* La información ha de llegar a tiempo, ni muy pronto ni demasiado tarde.

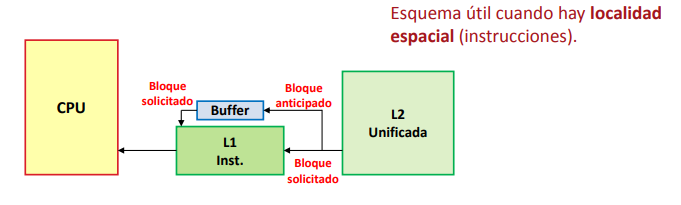
El problema del prefetch es que podemos traer información NO ÚTIL a la cache, ocupando espacio de MC y ancho de banda entre MC y MP.

Tipos de prefetch:

* Prefetch hardware (datos e instrucciones).
* Prefetch software (datos).

Prefetch hardware de instrucciones en el Alpha 21064:

* Cuando se produce un fallo de cache, se traen el bloque solicitado (bloque i) y el siguiente en secuencia (bloque i + 1), si no está ya en la cache.
* El bloque solicitado se deja en la MC y el siguiente en el buffer de prefetch.
* Si hay fallo en cache, pero acierto en el buffer, se sirve el fallo desde el buffer, se pasa el bloque i + 1 a la MC y se trae al buffer el siguiente (bloque i + 2).
* El buffer puede tener varias entradas:

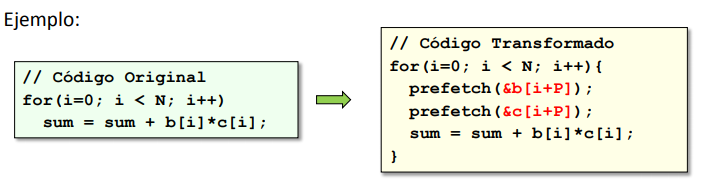


Prefetch hardware de datos:

* Prefetch en fallo:
  + Prefetch del bloque i + 1 cuando se falla en el bloque i.
* Esquema OBL (One Block Lookahead):
  + Prefetch del bloque i + 1 cuando se accede al bloque i.
* Prefetch con stride:
  + Si se observa una secuencia de accesos al bloque B, B + N, B + 2 \* N, entonces se hace prefetch de B + 3 \* N…

Prefetch software de datos:

* Se utilizan instrucciones especiales (las insertan el compilador o bien el programador de LM) para traer los datos de forma anticipada.
* En general, el prefetch puede introducir tráfico (entre MP y MC) innecesario.
* Se pierde tiempo ejecutando instrucciones de prefetch.



* El valor de P no es trivial de calcular:
  + Si hacemos prefetch muy cercano (P↓), es posible que el dato no llegue a tiempo.
  + Si hacemos prefetch muy lejano (P↑), tendremos polución en la cache.
* Normalmente, no ejecutaremos el prefetch en todas las iteraciones, lo haremos por ejemplo 1 de cada 4 (dependiendo del tamaño de línea de cache).
* Si suponemos que los vectores están alineados y que caben 4 elementos por línea, la transformación podría ser:

