Conceptos Avanzados de Memoria Caché

Queremos optimizar:

```
Ttotal = N * CPI * Tc
```

La optimización hay que hacerla en el CPI, tal que:

```
Ttotal = N * (CPI_ideal + n_ref * miss * t_pen + CPI_wr) * Tc
```

Para reducir los fallos hay que analizar los tipos de fallos.

- Fallos de carga: son fallos causados por la primera introducción en la caché. (no podemos evitar estos fallos).
- Fallos por conflicto: son fallos debidos a la asociatividad de la caché.
- Fallos de capacidad: Todas las líneas del programa no caben en la caché.

> Fallos de coherencia: en los sistemas multiprocesador.

Acciones posibles

- **Aumentar el tamaño de bloque (m↓)**. Reduce los fallos de carga, pero puede ser contraproducente (m↑).
- **Aumentar el tamaño de 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 (t_{pf}↓).** L1 pequeña (**m↑ y tsa↓**) y L2 grande (**m↓ y tsa↑**).
- **Dar más prioridad a las lecturas que a las escrituras (CPI_{WR}↓).** El coste de las escrituras se puede reducir (ocultar) utilizando buffers de escrituras.

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

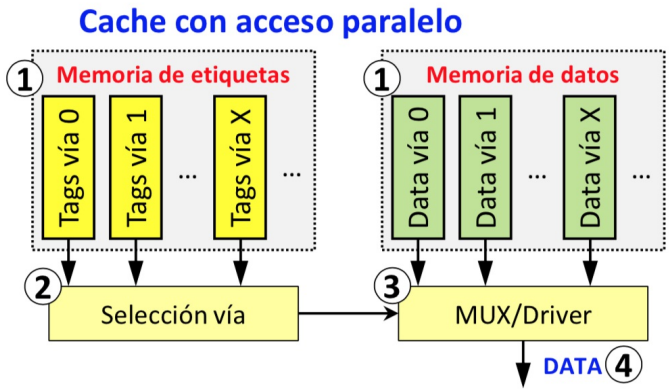
Cachés pequeñas y simples (- hit time)

Utilizar cachés pequeñas y simples para reducir el tiempo de acceso en caso de acierto.

Predicción de vía

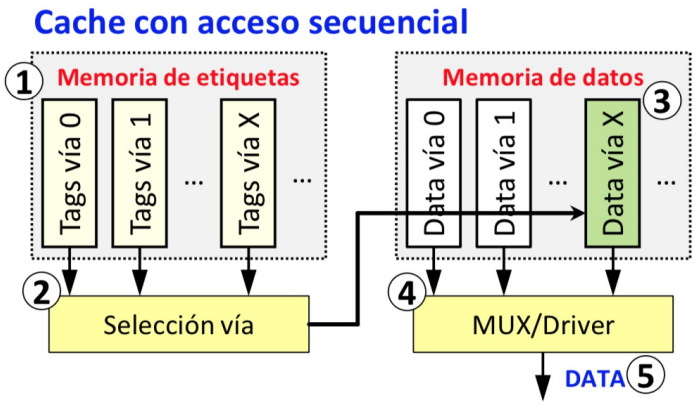
Para reducir el tiempo de acceso en cachés asociativas.

Paralelo



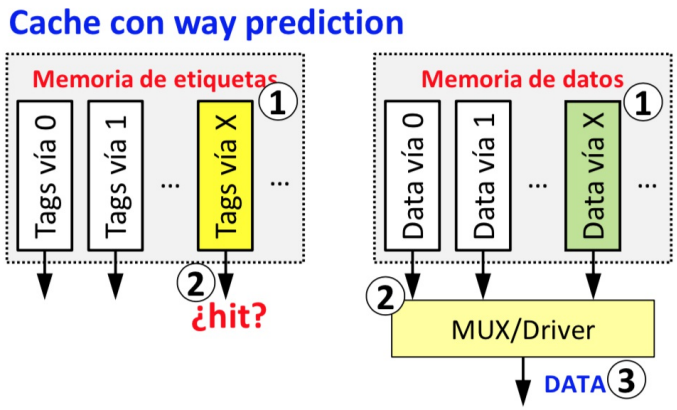
> Consumo alto. Tiempo de acceso más rápido.

Secuencial



> Consumo bajo. Tiempo de acceso más lento.

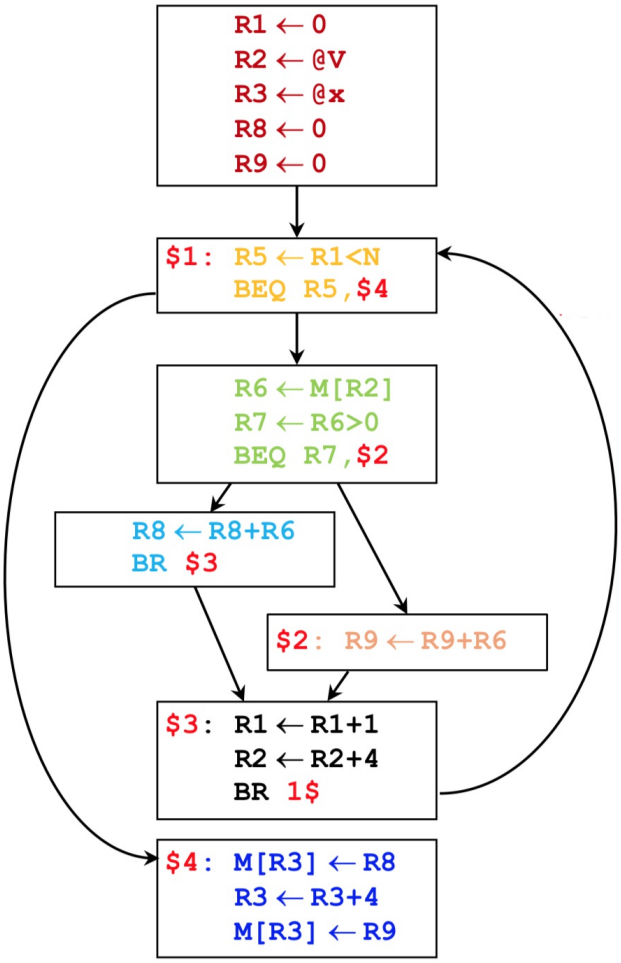
Way prediction



> Intento adivinar qué vía va a ser. Para ello tengo una tabla, en la que almaceno la vía en la que está guardada la etiqueta. Esta tabla es pequeña. Se direcciona con unos pocos bits.

Trace cachés

El compilador coge el código y lo parte en bloques básicos: mira todas las direcciones de salto y cada cosa que hay entre dos direcciones, hace un bloque.



Una caché normal hace esto:

...	...	R1 ← 0	R2 ← @V
R3 ← @x	R8 ← 0	R9 ← 0	R5 ← R1<N
BEQ R5,$4	R6 ← M[R2]	R7 ← R6>0	BEQ R7,$2
R8 ← R8+R6	BR $3	R9 ← R9+R6	R1 ← R1+1
R2 ← R2+4	BR 1$	M[R3] ← R8	R3 ← R3+4
M[R3] ← R9	...	...	...
...	...	...	...
...	...	...	...

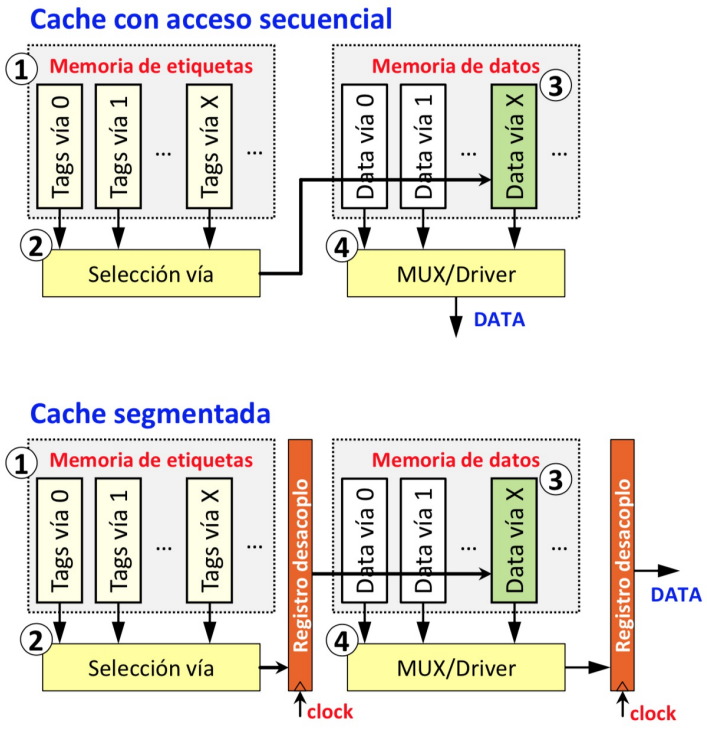
Una trace caché hace esto:

R5 ← R1<N	BEQ R5,$4	R6 ← M[R2]	R7 ← R6>0	BEQ R7,$2	R8 ← R8+R6	BR $3	R1 ← R1+1	R2 ← R2+4	BR 1$
R5 ← R1<N	BEQ R5,$4	M[R3] ← R8	R3 ← R3+4	M[R3] ← R9					
R5 ← R1<N	BEQ R5,$4	R6 ← M[R2]	R7 ← R6>0	BEQ R7,$2	R9 ← R9+R6	R1 ← R1+1	R2 ← R2+4	BR 1$	
...	...	...	...						

Tienes diversas rutas (trazas) de ejecución y lo que se tiene que hacer es intentar adivinar por qué traza de ejecución va a tirar el programa.

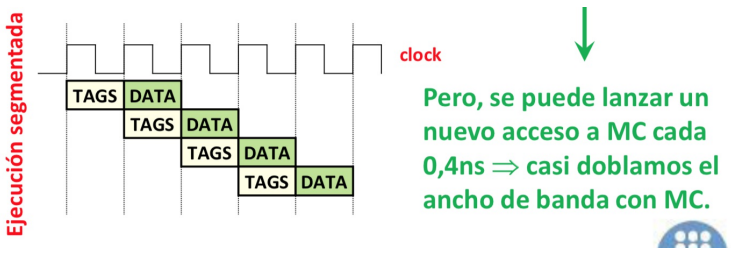
Cachés segmentadas (+ ancho de banda)

Pongo un registro en medio del proceso de la caché para guardar la información. Mientras busco el dato en una parte, puedo acceder a una memoria de etiquetas en la otra parte.

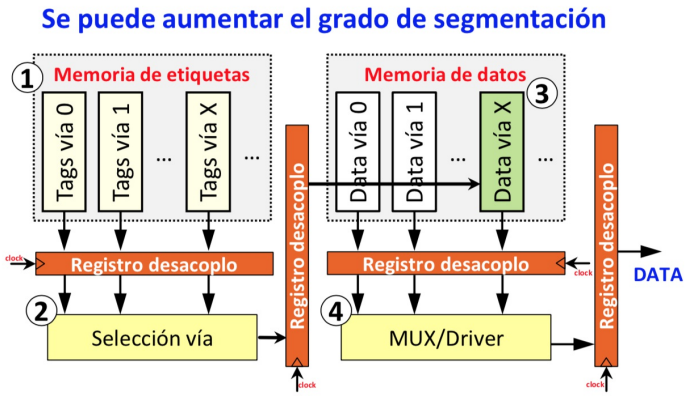


> El tiempo de reloj de la caché segmentada sería lo que tarde más, las etiquetas o los datos. (si una instrucción accede a memoria, duraría 2 ciclos)

Aunque el tiempo de acceso en caso de acierto sea más lento, transmito más bytes por ciclo ya que cada ciclo estaría dando una dirección de memoria:

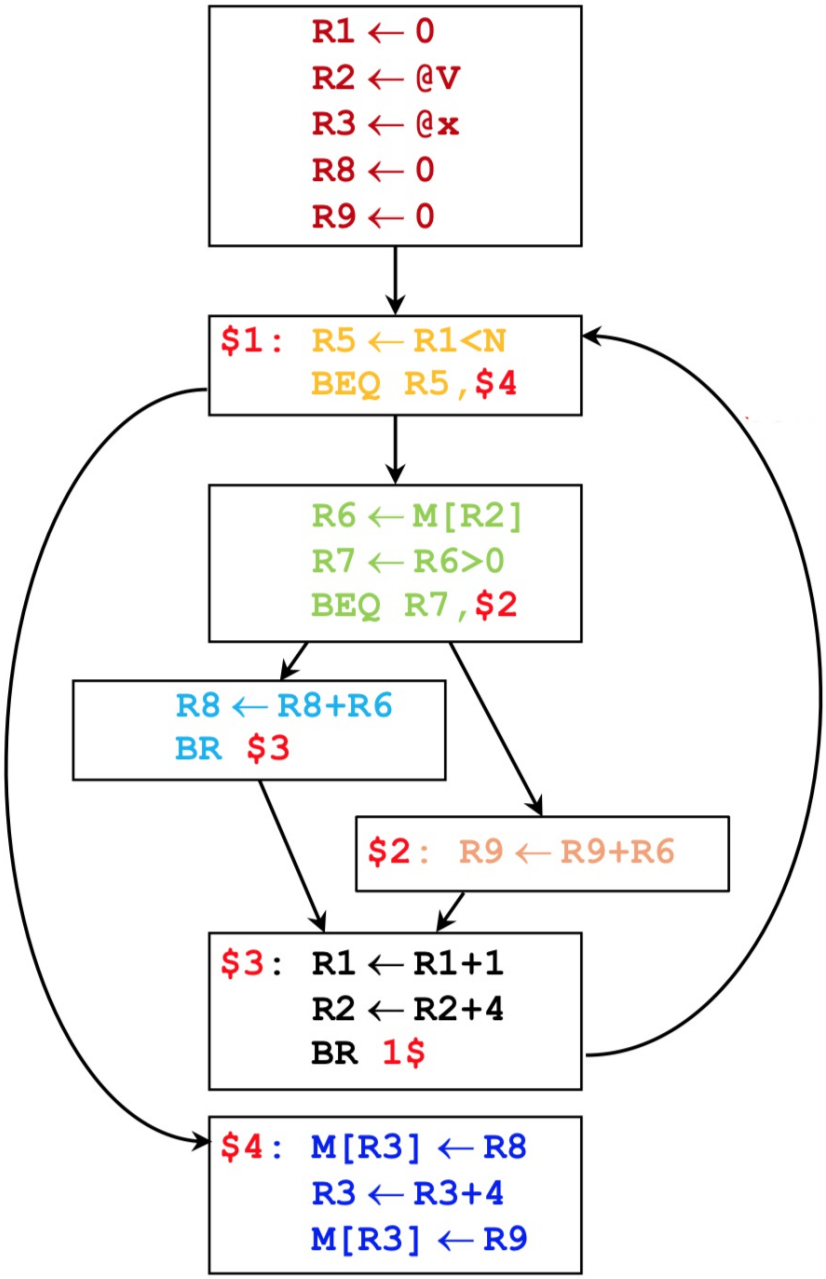


Puedo segmentar más poniendo más registros dentro de los procesos:



Non blocking caché

> Pasar de esto:


R1 ← 0
R2 ← @V
R3 ← @x
R8 ← 0
R9 ← 0
$1: R5 ← R1<N
BEQ R5,$4
R6 ← M[R2]
R7 ← R6>0
BEQ R7,$2
R8 ← R8+R6
BR $3
$2: R9 ← R9+R6
$3: R1 ← R1+1
R2 ← R2+4
BR 1$
$4: M[R3] ← R8
R3 ← R3+4
M[R3] ← R9

> A esto:

...	...	R1 ← 0	R2 ← @V
R3 ← @x	R8 ← 0	R9 ← 0	R5 ← R1<N
BEQ R5,$4	R6 ← M[R2]	R7 ← R6>0	BEQ R7,$2
R8 ← R8+R6	BR $3	R9 ← R9+R6	R1 ← R1+1
R2 ← R2+4	BR 1$	M[R3] ← R8	R3 ← R3+4
M[R3] ← R9	...	...	...
...	...	...	...
...	...	...	...

Para ello usa un **MSHRs (Miss Status Handler Register)** para gestionar los datos pendientes. Los MSHRs guardan el registro destino y cuando llega la información, se mete en el destino.

Para cada línea que falla, se pone un registro destino en el MSHRs.

Como pueden haber dos fallos de la misma línea en diferentes registros, asignamos varios registros por línea en el MSHR.

De este modo, el procesador tan solo se parará cuando necesite un dato que aún no ha llegado y no pueda ejecutar ninguna instrucción antes.