ARQUITECTURA E INGENIERÍA DE COMPUTADORES

Tema 3.2

dividas. Si es más rápido dan menor que 1 y ya.

Miquel Gómez Corral Ing. Informática ETSINF, UPV

Tema 3.2

Mejora de las prestaciones de las memorias cache

Mejorar las prestaciones de las memorias cache: Reduciendo cualquiera de los términos del $T_{acceso} = TA + TF \cdot PF$.

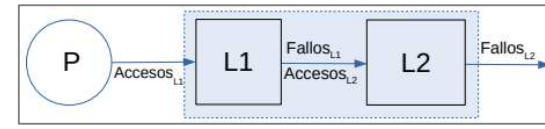
REDUCCION DE LA PENALIZACIÓN POR FALLO

Se puede de **3 maneras**: Caches multinivel "*Critical word first*" y "*Early restart*" o Buffers de escritura combinadas.

Caches multinivel

Reusar la idea de añadir una cache intermedia: Meter más caches. Ahora tenemos 2 por lo que cambia en T_{acceso} de la cache 1. Ahora depende del de la caché 2: $T_{acceso} = TA_{L1} + TF_{L1} \cdot (TA_{L2} + TF_{L2} \cdot PF_{L2})$

Tipos de tasas de fallos con varios niveles de cache



Tasa de fallos local de una cache: $\frac{Num\ fallos\ cache}{Num\ total\ acceso}$

- Para 2 niveles tendremos: $TF_{L1} = \frac{Fallos\ L1}{Accessos\ L1}$ y $TF_{L2} = \frac{Fallos\ L2}{Accesos\ L2}$

Tasa de fallos global (*llega memoria*): Producto de la tasa de fallos locales: $TF_{L1} \cdot TF_{L2}$. ***FRACCIÓN DE FALLOS QUE LLEGAN A MEMOIRA***.

La tasa de fallos global es casi la misma con 1 cache que con 2, lo que pasa es que con 2 tenemos una cache más rápida cerca del procesador, por lo que le pasa datos MUCHO más rápido.

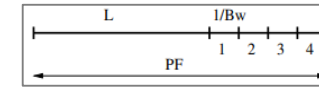
Inclusión/exclusión multinivel

Inclusión multinivel: **TODO lo que esté en la cache1 está en la cache2**. Aún que sea redundante, ayuda a mantener coherencia entre cachés (*solo necesitas comprobar el nivel 2*). Igualmente si invalidas a un nivel bajo invalidas a todos

Exclusión multinivel: **No hay bloques replicados en las caches**, están en L1 o en L2. Mejor aprovechamiento del espacio: SI hay un fallo en L1 y el bloque está en L2 intercambian un bloque. Si el bloque no está en ninguno SOLO se lo trae L1 de memoria. La coherencia es algo más complicada.

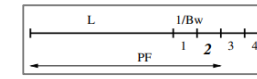
"Critical word first" y "Early restart"

El **tiempo que tardas en acceder** a memoria para conseguir un dato es: $PF = L + \frac{B}{Bw}$

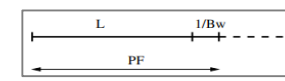


Latencia de acceder + el tiempo de pasar los datos: el Tamaño de bloque entre el ancho de banda: USAR UNIDADES CORRESPONDIENTES (*diferenciar entre frecuencia de la CPU y frecuencia de la memoria -> Pasar todo a segundos*): $PF = \left(L + \frac{B}{Bw}\right) \cdot \frac{f_{cpu}}{f_{mem}}$.

Early restart: El procesador cuando pide una palabra la cache le trae TODO el bloque, pero para que **NO se espere a que llegue TODO**, **solo se espera cuando le llega su palaba**: *Si había pedido la 2 pues cuando le llegue se desbloquea, sigue ejecutando lo suyo y de fondo sigue la cache trayendo.*



Critical word first: Se **trae primero la palabra solicitada** y se le entrega al procesador. Este NO se espera y mientras, la cache, se Trae el resto del bloque (*en orden circular*).



Las mejoras obtenidas por **Critical word first** y **Early restart** son mayores cuando se emplean tamaños de bloque grandes. Si El siguiente acceso a memoria no referencia el mismo bloque que se está cargando. *Si se accede al mismo bloque el segundo acceso debe esperarse a que el bloque entero este cargado.*

Buffers de escritura

Problema de las escrituras es que la **memoria es mucho más lenta que el procesador** por lo que las escrituras pueden parar al procesador. Se soluciona haciendo que el **procesador escribe sobre el buffer sin parar la ejecución**, ***desacoplando la ejecución de la escritura en memoria***, *que la lleva a cargo el controlador*.

Lo que pasa es que, si haces un store sobre una posición de memoria y luego se lee, como tarda en escribir en esa posición el load puede adelantarse y leer mal el dato. **Soluciones**:

1. **Espera**: Esperar a que el buffer de escritura se vacío antes de leer el dato. (*lento xq retrasas todas las load*) *Si hay estores pendientes te paras*.
2. **load-bypassing**: Comprobar si la **dirección referenciada está en el buffer de escritura** y, si no está, **dejar que la lectura** continue, sino se espera. *Si hay pendientes SOLO donde yo escribo me espero*.

- **store-to-load-forwarding**: Igual que antes, pero encima, NO se espera a que se escriba en memoria, **pilla el dato directamente del buffer**. *Si hay SOLO donde yo escribo LEO del buffer y sigo*.

Buffers de escrituras combinadas

Si el write buffer está lleno y se necesita una entrada habrá que poner stalls, tonces, lo que haces es, en vez escribir un bloque a la vez en vez de uno en uno, **los haces todos a la vez**.

Esto lo hace esperando un poco a empezar a escribir. Se espera a ver si hay más intentos de escribir en bloques consecutivos, y en tal caso se los lleva todos a la vez.

REDUCCIÓN DE LA TASA DE FALLOS

Los **fallos de la caché se producen por 3 causas**, las CCC: Compulsory, capacity, conflicto. **Formas de mejorar**: Ajustar la geometría (*Tamaño bloque*, *tamaño cache*, *numero de vías*) y optimizar el compilador.

Arranque (*Compulsory*): Originados la primera vez que se accede a un bloque xq al principio la caché está vacía. Es un **bajo porcentaje del total**.

Capacidad: Si la cache **no puede alojar todos los bloques** necesarios durante la **ejecución** de un programa, hay bloques activos que se reemplazan, por lo que se producirán fallos de capacidad: *Se reducen al incrementar el tamaño de la cache*.

Conflicto: Aparecen **cuando el conjunto destino está lleno,** pero hay espacio en otros conjuntos: No hay en una cache totalmente asociativa, pero necesita mucho hardware y puede reducir la frecuencia de reloj.

Tamaño de bloque

El tamaño típico es 64 Bytes. Algunos procesadores ponen un **tamaño más grande en el último nivel de cache**.

Ventajas: Contra **más grande sea el bloque**, menos fallos de arranque tienes xq cuando se lo traiga a cache tendrás más código/datos.

Desventajas

- No puedes hacerlos ultra grandes xq **sino tienes más fallos de capacidad** y de conflicto xq te caben menos bloques diferentes.
- Aumentar el tamaño de bloque aumenta la penalización en caso de fallo (↑ PF), ya que hay que traer más palabras. También depende de la latencia, xq si la latencia es grande pues un bloque más grande tiene menos importancia, pero si es pequeña cuidao: **compromiso entre TF y PF**.

Tamaño de la cache

- Reduce los **fallos por capacidad** (↓ TF)
- Aumenta el tiempo en caso de acierto (↑ TA), Aumenta el coste y consumo

Lo mejor es tener una L1 pequeña y rápida y otras L2 más grandes y menos costosas a cambio de ser más lentas.

Numero de vías (cuanto cabe en cada conjunto)

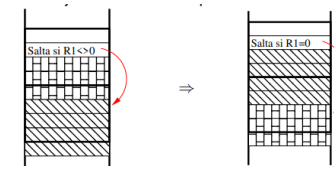
- Reduce los **fallos por conflicto** (↓ TF).
- Requiere más comparadores (más consumo de energía). El multiplexor de vía tiene más entradas por lo que aumentar el tiempo en caso de acierto (↑ TA)

Optimizaciones del compilador

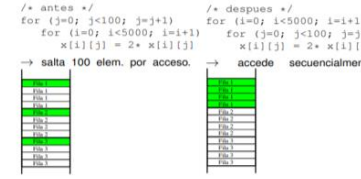
El compilador genera un **código optimizado que reduce la tasa de fallos**. Lo hace reordenando los accesos para que se hagan por bloques consecutivos de memoria y no de manera salteada.

También **Alinear el punto de entrada de los bloques** (*principio de donde está la información*) con el principio de un bloque de cache: **mejora la localidad espacial**. Si lo ajustas bien te pueden traer más instrucciones que luego leerás.

Branch Strainghtening (*Mejora localidad temporal*): Si el compilador cree que un salto sera´ "tomado", se modifica el codigo para: evaluar la condicion contraria, y úbicar a continuacion de la instrucci ´ on de salto el c ´ odigo antes ´ ubicado en el destino del salto. LO HACE PARA LA PROPIEDAD DE LA LOCACIDAD espacial.



Reducción de los fallos de datos: Ejemplo operaciones con matrices. Reorganizar el codigo para ´ operar sobre todos los datos de un bloque antes de pasar al siguiente



Blocking (*Mejorar la localidad temporal*): Dividir las matrices en bloquecitos más pequeños (submatrices) de forma que el procesamiento lo haces para cada subloquue de formaq que traigas menos información a la memoria y NO se me sobrescriban.

```
/* antes */
for (i=0; i<N; i=i+1)
  for (j=0; j<N; j=j+1) {
    r=0;
    for (k=0; k<N; k=k+1)
      r= r+y[i][k]*z[k][j];
    x[i][j]= r;
  }
```

```
/* despues */
for (jj=0; jj<N; jj=jj+B)
  for (kk=0; kk<N; kk=kk+B)
    for (i=0; i<N; i=i+1)
      for (j=jj; j< min(jj+B,N); j++) {
        r=0;
        for (k=kk; k<min(kk+B,N); k++)
          r= r+y[i][k]*z[k][j];
        x[i][j]= x[i][j]+r;
      }
```