Capítulo 5

La jerarquía de memoria y caches

A cualquier programador le agradaría tener una cantidad ilimitada de memoria. Además le agradaría que esta fuera muy rápida, en buena medida la rapidez en la ejecución de un programa está determinada por la velocidad de acceso a la memoria, varios ordenes de magnitud más lenta que el procesador.

Memorias de diferentes velocidades y precios.

Hay diferentes tipos de memoria, algunos muy rápidos, otros muy lentos. A mayor velocidad mayor es también el costo de la memoria. Existen memorias mucho más rápidas que nuestras RAM usuales, pero son mucho más caras, Si pretendiéramos que toda nuestra RAM fuera del tipo rápido el costo sería excesivamente alto.

Analicemos la situación. Hay dos extremos: (a) poner pura memoria lenta, el desempeño del sistema es el peor posible, el costo es el más barato posible. (b) poner pura memoria rápida, el desempeño del sistema es el mejor posible, el costo es el más alto posible. Es evidente el compromiso entre costo y velocidad, necesitamos lograr un equilibrio razonable, necesitamos encontrar un punto intermedio entre los extremos descritos.

De este problema se percataron los primeros en diseñar computadoras. Goldstine, Burks y Von Neumann escribieron: “Estamos forzados a reconocer la posibilidad de construir un jerarquía de memorias, cada una con una mejor capacidad que le precedente pero más lenta.

Jerarquía de memoria.

La solución fue, desde el principio, definir una jerarquía de memoria. Se colocan varios niveles de memoria. Unos muy rápidos, muy caros y muy cercanos al procesador, otros más lentos, más baratos y más lejanos. El tamaño de cada nivel es proporcional a su lejanía del procesador. Se pone poca memoria rápida y cara y mucha memoria lenta y barata. Cada nivel de memoria es, en términos de contenido, un subconjunto del nivel inmediato superior, más barato y grande. Es decir todos los datos contenidos en el nivel i están también en el nivel $i + 1$.

Hay diferentes tecnologías para construir dispositivos de memoria:

- SRAM[1]. Tiene tiempos de acceso de 5 a 25 ns. y un costo de 100 a 250 dolares el Mb.
- DRAM[2]. De 60 a 120 ns. de tiempo de acceso y de 5 a 10 dolares por Mb.
- Disco magnético. De 10 a 20 ms. de tiempo de acceso y de 0.1 a 0.2 dolares el Mb.

Cuando uno va a la biblioteca a hacer un trabajo de algún tema en particular, junta varios libros y los lleva a su mesa. Si se eligen bien los libros es muy probable que con esos baste, pero eventualmente ocurrirá que uno no encuentra algo en esos libros o ha cambiado de tema y tiene que regresar a la estantería a buscar nuevos libros. Algunos necesarios, otros porque se piensa que pueden ser útiles en algún momento. Usualmente los libros del mismo tema están cerca unos de otros en la estantería, así que por cada trabajo que hay que hacer, uno normalmente accede sólo a una pequeña sección de le estantería. Además suele suceder que, luego de que uno ha consultado un libro, lo continua utilizando durante un rato.

Principio de localidad.

A la situación ejemplificada en el párrafo anterior se le denomina *princi-*

[1]RAM estática, no requiere estarse refrescando periódicamente como la dinámica, el acceso a este tipo de memoria es más rápido que el acceso a DRAM porque no hay que esperar a que se termine el refresco.

[2]RAM dinámica, requiere de estarse rerfescando periodicamente a una cierta frecuencia propia de la memoria (tiene su propio reloj), esto limita la capacidad de respuesta de la memoria. Cuando el procesador solicíta un dato debe esperar a que termine el refresco y a que el dato sea extraido de la memoria. Para minimizar este periodo de latencia se inventó la memoria SDRAM (*Synchronized DRAM*) en la que la frecuencia de refresco es la misma que la frecuencia de operación del CPU y por tanto este no tiene que esperar tanto, ya que se puede sincronizar con la memoria.

pio de localidad. Como ya nos percatamos en el ejemplo, hay dos variantes: una vez que se ha accedido a un libro (dato) es muy probable que se accedan también los libros (datos) que se encuentran muy cerca de él (*localidad espacial*); una vez que se ha accedido un libro (dato), es muy probable que en un futuro cercano se vuelva a acceder nuevamente (*localidad temporal*). Al llevarnos los libros a nuestra mesa definimos una jerarquía de memoria. Nuestra mesa es una memoria de acceso rápido y es un subconjunto de la memoria principal, nuestra biblioteca.

La utilidad de la jerarquía de memoria radica en el principio de localidad. Si este no es cierto en ciertas circunstancias, la jerarquía de memoria tendrá un impacto menos favorable o desfavorable sobre el desempeño del sistema. Con una jerarquía de memoria es posible lograr ese buen compromiso que ya mencionamos en la relación costo-beneficio. En principio pueden existir muchos niveles de memoria. En general sólo ponemos 4 o cinco niveles: registros del procesador, cache L1, cache L2, RAM y disco (por cierto en Unix es muy evidente el uso de disco como miembro de la jerarquía, el área de *swap*).

En la jerarquía de memoria, tal como ocurre con la mesa de biblioteca, los niveles de memoria más rápidos, más caros y más cercanos al procesador son un subconjunto de (contienen información que también está contenida en) los niveles más lentos, más lejanos y baratos. Los datos son copiados únicamente entre niveles adyacentes. Las unidades de información mínima que se transfieren entre niveles se denominan *bloques*, el análogo de un libro en nuestro ejemplo.

Acierto (*hit*) y fallo (*miss*).

Cuando se solicita información a un nivel, si el bloque que la contiene está en dicho nivel se dice que ha tenido lugar un acierto (*hit*). Si la información no se encuentra en el nivel al que se solicito se dice qe ha ocurrido un fallo (*miss*). El administrador de este nivel debe solicitarla al nivel inmediato superior, más grande y más lento, hay que ir al estante por el libro solicitado.

Una medida del desempeño de una jerarquía es la tasa de acierto: la fracción de accesos a memoria que son satisfechos con éxito por el nivel más cercano al procesador.

Debemos definir ahora un par de intervalos promedio de tiempo:

1. Tiempo de acierto (*hit-time*). Tiempo de acceso al nivel de memoria más cercano, mas el tiempo necesario para determinar si es acierto o

fallo.

2. Penalización por fallo (*miss penalty*). Tiempo necesario para reemplazar un bloque de memoria de un nivel cercano al procesador por el que es requerido y que está en un nivel más lejano, mas el tiempo necesario para entregar el bloque al procesador.

Los caches de nuestras computadoras actuales están destinados a estar en los niveles más cercanos al CPU. Hay varias preguntas que debemos resolver para diseñar un cache: ¿donde debe colocarse un bloque del nivel de memoria superior en el cache? ¿como se encuentra un bloque de memoria en el cache? en caso de fallo, ¿cuál bloque del cache debe ser reemplazado por el bloque proveniente de memoria? ¿que se hace en caso de una escritura en memoria? ¿donde se escribe?

Hay varias opciones para responder cada pregunta, dependiendo de la respuesta, un cache pertenece a una categoría u otra.

5.1. Colocación de un bloque en el cache

Mapeo directo. Cada bloque de memoria principal puede ponerse en uno y sólo un sitio del cache. Generalmente el bloque de memoria en la dirección *dir* de memoria principal se coloca en el bloque de dirección:

$$dir \pmod{B_{cache}}$$

donde B_{cache} es el tamaño del cache en bloques.

Completamente asociativa. (*fully associative*) el bloque se puede poner en cualquier lugar del cache.

Asociativa de conjunto de n posibilidades. (*n-way set associative*) el bloque de memoria se mapea a un conjunto de bloques del cache, el bloque de memoria puede ponerse en cualquiera de los bloques del conjunto. Generalmente:

$$c = dir \pmod{\#C}$$

c es el número de conjunto del cache, *dir* la dirección del bloque en memoria principal, $\#C$ el número de conjuntos del cache.

Si hay n bloques en cada conjunto del cache entonces hay n maneras de elegir el bloque dentro del conjunto donde se habrá de colocar el bloque de memoria principal. Es por eso que se llama de n posibilidades: n es el número de bloques en cada conjunto.

Nótese que esta alternativa define una gama de posibilidades entre la primera y la segunda. El mapeo directo es asociativo de conjunto de una posibilidad, la asociatividad completa es asociativa de m posibilidades donde m es el número total de bloques en el cache.

Los caches actuales son generalmente de mapeo directo, asociativos de conjunto de dos y de cuatro posibilidades.

5.2. Localización de bloques en el cache

El tamaño de un cache es mucho menor que el tamaño de la memoria principal, ese es el objetivo, que sea pequeño porque es muy caro. Así que para direccionar un bloque en el cache se requieren de muchos menos bits que para direccionar uno en la memoria principal. Sin embargo el objetivo es que, tenga o no cache la máquina, el acceso a memoria sea transparente para el procesador. Este debe decir la dirección de memoria de la misma manera, independientemente de si tiene o no cache. Asi que se debe definir una manera de traducir direcciones de memoria principal a memoria cache. El encargado de hacer la traducción debe ser un dispositivo de interfaz del cache con el procesador.

Hay que hacer algunas otras cosas. Si un bloque en el cache nunca ha sido leído de memoria o es inválido hay que tener manera de determinarlo marcando el bloque.

Por cierto, dado que los caches son suficientemente pequeños es posible verificar en paralelo todos los bloques del cache para ver si un bloque dado está o no. Para comprender como podría hacerse esto y como funciona la búsqueda de bloques en general debemos comprender cabalmente la traducción de direcciones.

Imaginemos que tenemos un cache de mapeo directo de 4 Kbytes, un bus de direcciones del procesador de 32 bits y un tamaño de bloque de 4 bytes. El procesador solicita entonces acceso a un dato en memoria diciendo 32 bits de dirección, imaginemos que el dato solicitado es un byte. ¿Cuantos

bits necesitamos para...

- direccionar un byte dentro de un bloque? cada bloque tiene cuatro así que necesitamos dos bits.
- direccionar un bloque en el cache? el cache es de 4Kbytes, cada bloque es de 4 bytes, así que el cache tiene 1Kbloque (1024 bloques) y para direccionar un bloque se necesitan 10 bits.

Traducción de direcciones.

Así que de los 32 bits de dirección necesitamos 12 para direccionar en el cache. Los restantes 20 bits sobran... pero es necesario conservarlos de alguna manera, porque juntos los 32 bits direccionan un byte particular, si sólo nos quedamos con los 12 menos significativos hay un total de 2^{20} direcciones que coinciden en esos 12 bits, ¿como los distinguimos?. Lo que se hace es poner esos 20 bits más significativos de la dirección como etiqueta de cada bloque en el cache.

El proceso completo es entonces:

1. El procesador dice una dirección de 32 bits: $dir[0, \ldots, 31]$.
2. El cache usa los bits $dir[2, \ldots, 11]$ como índice del bloque en el cache. Ese bloque tiene asociada una etiqueta (tag) de 20 bits $tag[0, \ldots, 19]$
3. Se comparan $dir[12, \ldots, 31]$ y $tag[0, \ldots, 19]$ si coinciden y ese bloque está marcado como válido el cache entrega el byte indicado por los bits $dir[0, 1]$ al procesador.
4. Si no coinciden $dir[12, \ldots, 31]$ y $tag[0, \ldots, 19]$, o si coinciden pero el bloque está marcado como inválido o vacío, es un fallo. Se solicita a la memoria principal el bloque direccionado por $00 \uplus dir[2, \ldots, 31]$[3]. La memoria entrega el bloque al cache, este lo almacena y lo pasa al procesador.

5.3. Reemplazo de bloques en el cache

Imaginemos que se requiere un bloque de memoria y que no se encuentra en el cache y los lugares donde se puede poner están ocupados. Hay que traer

[3]Nótese que los bits de índices 0 y 1 han sido puestos a cero alineando por bloque. Se transfiere todo el bloque aunque sólo se haya solicitado un byte.

el bloque de memoria principal y ponerlo en uno de los lugares que le corresponde quitando el bloque que actualmente este allí. ¿cuál lugar elegimos para poner el que necesitamos eliminar a su ocupante actual (regresarlo a memoria principal)?

- Si es un cache de mapeo directo sólo hay una posibilidad, si el bloque está ocupado, ni modo, debe salir el bloque alojado allí actualmente para colocar el nuevo.
- Si es asociativa de varias posibilidades hay más de una opción:
  - Aleatorio. Elegimos al azar el bloque a reemplazar.
  - El menos recientemente usado (*least recently used* o LRU). Si usamos un bloque hace poco entonces es muy probable que se vuelva a usar pronto, asi que, haciendo uso del principio de localidad, quitamos el bloque del cache, de aquellos donde el bloque de memoria puede residir, que haya sido usado menos recientemente. Por supuesto esto implica que cada bloque en el cache tiene un sello de tiempo *timestamp* con la "hora" de entrada.
  - Usar una cola para sacar el bloque que haya entrado primero.

5.4. Escritura

En general es más común (en promedio tres veces más común) hacer lecturas de memoria que escrituras, así que con nuestros caches tal como los hemos pensado hasta ahora, para lecturas, estamos optimizando el desempeño en el caso más común. Eso es bueno, pero a pesar de eso sería bueno mejorar las escrituras involucrando al cache.

Manejo de escrituras.

Cuando el procesador ordena una escritura en memoria, es muy probable que en un futuro cercano requiera el mismo dato que escribió, así que, en principio, sería bueno tener el dato en el cache. Pero podemos pensar en dos opciones:

- Escribirlo en el cache y en memoria (*write through*) al mismo tiempo.
- Escribirlo sólo en el cache y regresarlo a memoria sólo cuando sea indispensable por reemplazo (*write back*).

La escritura simultánea en memoria y el cache tiene la ventaja de que la memoria principal siempre está actualizada, siempre es consistente con el cache. Esto tiene ventajas, por ejemplo, en sistemas paralelos con memoria compartida. Pero se incrementa notablemente el tráfico con la memoria principal. Generalmente cada vez que se escribe se debe esperar a que la memoria termine y eso genera retardos (*write stall*). A veces para evitar estos retardos se antecede la memoria de un buffer donde se almacena temporalmente el dato a escribir y de allí es tomado por la memoria principal sin generar retardos para el procesador, este hace de cuenta que ya escribió realmente.

Con el esquema de escritura *write back* se reduce el tráfico con la memoria, múltiples escrituras en el mismo bloque generan uns sola escritura final en memoria principal, mientras que con escritura simultánea cada escritura corresponde a una escritura en memoria principal. Generalmente el bloque modificado en el cache se marca para saber si hay que escribirlo o no cuando sale por reemplazo. Se le pone un bit "sucio" *dirty bit* para indicar que debe ser escrito.

Asi como hay fallos al momento de solicitar una lectura de memoria también podría haberlos al momento de solicitar una escritura. En esos casos hay nuevamente dos opciones:

- *write allocate* o *fetch on write*, se trae en bloque de memoria al cache y luego se escribe en el, ya sea con escritura simultánea o después, con *write back*.
- Se escribe directamente en la memoria, este esquema se llama simplemente *no-write allocate*.

Podría parecer que no es muy conveniente el primero de estos esquemas usando escritura simultánea, pero lo és gracias al principio de localidad, es muy probable que el bloque se se acaba de escribir sea utilizado en un futuro cercano, así que lo traemos al cache previniendo futuros accesos.

5.5. Desempeño de caches

Podemos preguntarnos ahora ¿como organizamos el cache de una máquina?. Hay dos cosas que la computadora debe acceder de la memoria, código y datos. Así que podemos pensar en dos alternativas diferentes: (1) poner un

solo cache en donde se almacenen tanto datos como código y (2) poner dos caches (o un cache dividido en dos secciones, formalmente hablando) uno para datos y otro para código. Como siempre, para saber que alternativa es mejor debemos hacer cuentas. En el libro de Hennessy (figura 5.7, página 384) se muestra una tabla con las tasas de fallo de diferentes tamaños de cache, tanto divididos como unificados. Lo que se observa es que la tasa de fallo para el cache de datos siempre es mayor que la tasa del unificado y la del cache de instrucciones siempre es menor. Por ejemplo para un cache de 64 Kb. la tasa de fallo para instrucciones es de 0.15 % de los accesos, la tasa de fallo para datos es de 3.77 % y la del unificado (de 128 Kb.) es de 0.95 %. Dado que son tasas de fallo podemos pensar en sacar el promedio de tasa de fallos para datos e instrucciones y comparar con la tasa del unificado, si hacemos esto obtenemos un promedio de tasa fallo de 1.96 % para el cache dividido (64Kb. datos + 64Kb. instrucciones), contra el 0.95 % de tasa de fallo del unificado (128Kb.), por lo que pensaríamos que es mejor tener un cache unificado.

Evaluación correcta del desempeño.

Pero en realidad sólo tenemos una visión parcial de las cosas, medir el desempeño de caches considerando únicamente la tasa de fallo es un error. Estamos perdiendo de vista las situaciones en las que el cache es realmente útil, cuando no hay fallo. Para evaluar correctamente el desempeño de los caches debemos abrir nuestro campo visual. Usaremos el tiempo promedio de acceso a memoria como medida del desempeño:

$$\overline{T}_{acc} = T_{acierto} + T_{fallo} R_{fallo}$$

En esta expresión $\overline{T}_{acc}$ es el tiempo promedio de acceso, $T_{acierto}$ es el tiempo de acceso a la memoria cuando lo que se solicita está en el cache, T_{fallo} es el tiempo de acceso a la memoria cuando lo que se solicita no está en el cache y R_{fallo} es el porcentaje, o proporción, o tasa, de fallo (número de veces que el acceso genera un fallo entre el número total de accesos).

Con esto podemos proceder a evaluar cual alternativa es mejor, si un cache unificado o uno de datos y otro de código. Supongamos que en caso de acierto el tiempo de acceso a memoria es de 1 ciclo de reloj y en caso de fallo es de 50 ciclos. En una máquina con pipelining, en un solo ciclo de reloj se pretendería acceder dos veces al cache unificado, porque la etapa de MEM de una instrucción ocurriría al mismo tiempo que el IF de otra. Esto, como habíamos dicho es un hazard estructural y hace que se tenga que atorar el pipeline un ciclo, así que, en caso de acierto en un cache unificado, hay un retardo extra de un ciclo de reloj en el acceso a datos, porque siempre

que se accede a un dato al mismo tiempo se pretende hacer el acceso por la siguiente instrucción.

También debemos tener en consideración, en al caso de un cache dividido, cuanto se usa cada sección, cuantos accesos a memoria son causados por acceso a instrucciones y cuantos por acceso a datos. Según medidas experimentales, un 75 % de los accesos a memoria son por instrucciones y el 25 % son por datos (SPEC). Así que nuestras tasas de fallo serán:

$$R_{fallo}^{dividido} = (0{,}75 \cdot 0{,}0015) + (0{,}25 \cdot 0{,}0377) = 0{,}01055 \ fallos/acceso \quad (5.5.1)$$

Esta es una medida más fina que el promedio que usamos arriba, se consideran los pesos reales de cada tipo de acceso (.75 para instrucciones y .25 para datos) que es más verosímil que usar el mismo peso para ambos tipos, como hicimos arriba.

por otra parte, según la tabla de Hennessy:

$$R_{fallo}^{unificado} = 0{,}0095 \ fallos/acceso \quad (5.5.2)$$

A pesar de que 5.5.1 es una medida más fina que el promedio burdo usado antes sigue ocurriendo que parece mejor un cache unificado. Pero las cosas cambian si usamos el tiempo promedio de acceso:

$$\overline{T}_{acc}^{unificado} = 0{,}75(1{+}0{,}0095{\cdot}50){+}0{,}25(1{+}1{+}0{,}0095{\cdot}50) = 1{,}725 \ ciclos/acceso \quad (5.5.3)$$

Por otra parte:

$$\overline{T}_{acc}^{dividido} = 0{,}75(1 + 0{,}0002 \cdot 50) + 0{,}25(1 + 0{,}0288 \cdot 50) = 1{,}3675 \ ciclos/acceso \quad (5.5.4)$$

Por supuesto, con la visión global que nos da el medir el tiempo esperado total de acceso a memoria, obtenemos el intuitivo resultado de que es mejor tener un cache dividido en vez de uno unificado que se vuelva un cuello de botella en una máquina con pipeline y, lo que es determinante, que tiene una tasa de fallo superior al promedio pesado de las tasas de cada división (sin pipeline, el tiempo promedio de acceso es 1.475 en el cache unificado, que sigue siendo mayor que el del dividido).

Así pues la medida correcta para evaluar el desempeño de un cache es el tiempo promedio de acceso a memoria. No sólo evaluamos lo que perdemos

si las cosas salen mal, sino que evaluamos también cuanto ganamos si las cosas salen bien.

5.6. Mejoras al desempeño

Por lo que hemos visto el desempeño de un cache puede ser mejorado por los siguientes medios:

1. Reduciendo el número (proporción) de fallos
2. Reduciendo la penalización por fallo
3. Reduciendo el tiempo de acceso en caso de acierto

Para reducir la penalización por fallo lo que se suele hacer desde hace unos pocos años (a partir de 1995 se ha ido haciendo más usual) es poner dos niveles de cache. En el esquema con un solo cache, la penalización por fallo es grande dado que hay que solicitar la información directamente a la memoria principal, mucho más lenta que la memoria cache, pero si se coloca otro cache, uno de segundo nivel (L2), es este el que, con muy alta probabilidad, debe responder en caso de que el dato solicitado no se encuentre en el cache de primer nivel (L1). El cache de segundo nivel no será tan rápido como el de primer nivel, pero si es mucho más rápido que la memoria principal. Actualmente los caches de primer nivel se colocan en la misma pastilla del procesador, en la misma estampa, por lo que se les llama internos. Los de segundo nivel son externos.

Otra estrategia para reducir la penalización por fallo consiste en hacer lo indispensable para responder y luego terminar. Cuando se requiere un dato de memoria, si es más barato traer sólo el dato requerido en vez de todo el bloque que lo contiene entonces se trae sólo el dato, se entrega y luego se termina la transferencia del bloque (*Critical Word First*).

Para reducir el tiempo de acceso en caso de un acierto se han hecho dos cosas: (1) hacer caches simples y pequeños y (2) evitar la traducción de direcciones. Esta segunda alternativa se denomina cache virtual y consiste en que el cache se hace pasar por la memoria principal sin traducir el espacio de direcciones de aquella al suyo propio. Básicamente el cache renombra sus direcciones para que correspondan a las de memoria y la desventaja de hacer

esto es que al cambiar de proceso, en un sistema capaz de ejecutar varios de ellos concurrentemente (*process switch*), hay que renombrar otra vez el espacio de direcciones completo y por tanto hay que vaciar todo el cache para asegurar consistencia con la memoria principal (*cache flush*).

Para atacar el primero de los rubros mencionados, reducir la tasa de fallos, hay que lograr un delicado equilibrio entre el tamaño de la memoria, el tamaño del cache y el grado de asociatividad de este. En general la memoria principal es mucho más grande que el cache, tres ordenes de magnitud mayor, aproximadamente. Si el cache tiene un grado alto de asociatividad (muchos bloques en cada conjunto), entonces tendrá menos conjuntos. Esto significa que cada conjunto debe dar servicio, en promedio, a grandes áreas de memoria principal, es decir, grandes áreas de memoria se mapean al mismo conjunto del cache. Esto hace que tienda a saturarse un bloque del cache mientras que muchos otros están desaprovechados, la saturación de un bloque hace que se incremente la probabilidad de tener que reemplazar bloques todavía en uso, por nuevos bloques que luego hay que reemplazar por los originales otra vez.

Regla 2:1 de caches.

Si el grado de asociatividad es muy bajo, entonces hay pocas opciones para colocar un bloque de memoria y es fácil que se llegue al conflicto de tener que reemplazar un bloque útil por uno nuevo y, al igual que en el caso anterior, tener que volverlo a reemplazar pronto. En general se ha observado que la tasa de fallo en un cache de mapeo directo de tamaño N es casi la misma que en un cache asociativo de conjunto de 2 opciones de tamaño $N/2$. Esto se conoce como *la regla 2:1 de caches*.

Para tratar de reducir estos problemas de reemplazos poco deseables pero necesarios, una alternativa es hacer un *cache de víctimas*, un área de memoria en la que se colocan los bloques que salen del cache por reemplazo, esta cantidad de memoria es de acceso rápido, no tanto como el cache, pero más rápida que la memoria principal, la idea es reducir la penalización por fallos debidos a reemplazos prematuros.

Cache exclusivo.

Hemos dicho que en la jerarquía de memoria cada nivel es un subconjunto del nivel inmediato superior, más grande, más barato y más lento. Así es porque la intención es que uno pueda acceder a los mismos datos que están en memoria pero más rápido. Sin embargo se esta ensayando la alternativa en la que en una máquina con dos niveles de cache, el cache L1 no es un subconjunto del L2. A este tipo de cache se le ha denominado exclusivo y es utilizado en el AMD *Thunderbird*, versión mejorada del *Athlon*. Aparen-

temente este cache exclusivo es punto clave (junto con el uso de cobre en vez de aluminio para las pistas) en la mejora de desempeño que ha puesto al Thunderbird ligeramente por encima de su rival, el *Copermine* de Intel, versión mejorada del Pentium III.

La ventaja evidente del cache exclusivo es que la totalidad del cache L2 es usada para almacenar datos que, por definición, no se encuentran en el de L1. En una situación normal, el cache L2 tiene una parte de su tamaño total ocupada por los mismos datos que contiene L1.