Práctica 2: Simulador de memoria caché

Luis Jafet Ramos Castillo y Walter Martínez Santana

1. Introducción

Las memorias caché son de tecnología de semiconductores de tipo estático, cuya velocidad de respuesta se ajusta de manera muy favorable a los tiempos del procesador. Lo que sustenta su implementación en el sistema es que su velocidad es compatible con las necesidades de obtención de la información por parte del procesador.

Para balancear costo, volumen de información y tiempo de acceso en la búsqueda de mejorar el rendimiento global, se utilizan memoria caché y memoria principal. La memoria caché se utiliza como memoria intermedia entre el procesador y memoria principal y almacena en forma temporal la información a la que se accede con mayor frecuencia en caché.

La caché es un subsistema constituido por una memoria de etiquetas, una memoria de datos y un controlador. El controlador se utiliza para gestionar su actividad. Una de sus funciones es seleccionar cuáles y cuántos bytes de memoria se copiarán en su matriz de datos. Este procedimiento no es arbitrario sino que sigue un criterio de actualización de datos o actualización de caché. Debemos indicar que el controlador observa el espacio de almacenamiento de la memoria RAM como un conjunto de bloques, cuyo tamaño es fijo.

La memoria de etiquetas es una matriz asociativa de $m * n$; cada fila está asociada a una línea de memoria de datos y se denomina etiqueta. Por lo tanto, hay tantas etiquetas como líneas. Se deduce que la cantidad de líneas de la caché depende de su capacidad expresada en bytes y que esta medida es el tamaño de la memoria de datos.

2. Implementación

EL simulador de memoria caché se implementó en C. A continuación se muestra parte del código acompañado de una breve explicación.

La estructura de linea de caché cuenta con un *tag* que determina en que linea del caché se encuentra, *dirty* indica si esa linea ha sido modificada, así como listas en caso de que exista una asociatividad mayor a 1.

```
/* structure definitions */
typedef struct cache_line_ {
  unsigned tag;
  int dirty;
  struct cache_line_ *LRU_next;     /* list for many ways */
  struct cache_line_ *LRU_prev;
} cache_line, *Pcache_line;
```

Código 1: Linea de caché

La estructura de caché cuenta un tamaño, asociatividad, número de lineas, mascaras que posteriormente se explicarán, listas de lineas de caché y el número de entradas validas en cada línea.

```
typedef struct cache_ {
  int size;   /* cache size */
  int associativity;  /* cache associativity */
  int n_sets;   /* number of cache sets */
  unsigned index_mask;  /* mask to find cache index */
  int index_mask_offset; /* number of zero bits in mask */
  Pcache_line *LRU_head; /* head of LRU list for each set */
  Pcache_line *LRU_tail; /* tail of LRU list for each set */
  int *set_contents;  /* number of valid entries in set */
} cache, *Pcache;
```

Código 2: Caché

La esctructura de estadísticas de caché cuenta con contadores para los accesos, fallos, reemplazos, *fetches* y *copies back*.

```
typedef struct cache_stat_ {
  int accesses;   /* number of memory references */
  int misses;   /* number of cache misses */
  int replacements; /*number of misses that cause replacements*/
  int demand_fetches;   /* number of fetches */
  int copies_back;  /* number of write backs */
} cache_stat, *Pcache_stat;
```

Código 3: Estadísticas de caché

Recordemos que una dirección de memoria se parte en 3: un conjunto de la parte menos significativa, denota el *offset*; luego, el número de línea; por último, el *tag*. Debido a lo anterior, las mascaras deben ser calculadas de tal manera que el número de linea y el *tag* puedan ser aislados con facilidad.

```
unsigned aux_mask1, tag;
unsigned int index;
aux_mask1 = c1.n_sets-1;
c1.index_mask = (aux_mask1) << LOG2(cache_block_size);
index = (addr & c1.index_mask) >> c1.index_mask_offset;
tag = addr >> (LOG2(c1.n_sets) + LOG2(cache_block_size));
```

Código 4: Máscaras

Lo anterior es útil, puesto que realizaremos diversas preguntas sobre lo que hay en el número de línea de la caché, ya que puede estar vacía y sólo la almacenamos e incrementamos el contador de entradas válidas en esa línea.

Si no esta vacía, entonces debemos de iterar sobre las listas de cada línea (debido a la asociatividad), sí encontramos que alguna línea ya tiene el *tag* de la dirección de memoria que queremos almacenar en caché, entonces la remplazamos, si no es el caso la almacenamos e incrementamos el contador de entradas válidas en esa línea sólo si este contador no es mayor que la asociatividad permitida, de lo contrario la remplazamos.

```
if(c1.LRU_head[index] != NULL){
 Pcache_line current_line;
 for(current_line=c1.LRU_head[index];current_line!=c1.LRU_tail[
    index]->LRU_next;current_line=current_line->LRU_next){
  if(current_line->tag == tag){
   // more things between
   Pcache_line line = current_line;
   delete(&c1.LRU_head[index], &c1.LRU_tail[index], current_line
   );
   insert(&c1.LRU_head[index], &c1.LRU_tail[index], line);
   return;
  }
 }
 Pcache_line line = malloc(sizeof(cache_line));
 if(c1.set_contents[index] < cache_assoc){
  // more things between
  insert(&c1.LRU_head[index], &c1.LRU_tail[index], line);
  c1.set_contents[index]++;
 }else{
  // more thing between
  delete(&c1.LRU_head[index], &c1.LRU_tail[index], c1.LRU_tail[
    index]);
  insert(&c1.LRU_head[index], &c1.LRU_tail[index], line);
 }
}
```

Código 5: Caso no vacío

3. Evaluación de desempeño

Se hicieron distintos experimentos con ciertas configuraciones para caracterizar el tamaño de la memoria, evaluar el impacto del tamaño del bloque, de la asociatividad y comparar el ancho de banda de la memoria.

Los experimentos se realizaron con distintas trazas proveídas: *spice*, *gcc* y TEX

3.1. Caracterización del tamaño de la memoria

Se hicieron distintas simulaciones utilizando una caché dividida para instrucciones y datos, bloque de 4 bytes, totalmente asociativa con políticas *write-back* y *write-allocate*. De esta manera, lo que vario fue el tamaño de la caché iniciando en 4 bytes e incrementando en factores de 2.

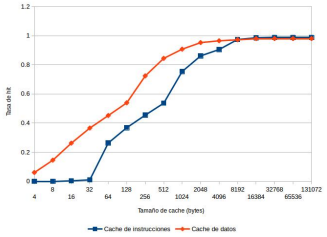

Figura 1: Traza spice

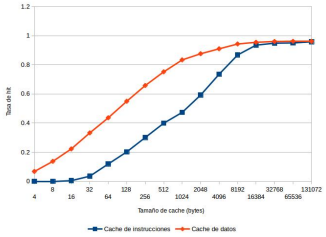

Figura 2: Traza gcc

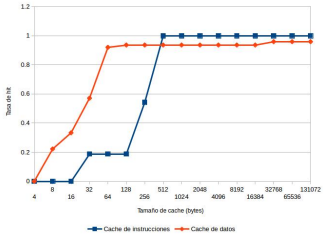

Figura 3: Traza tex

1. Explique que hace este experimento y cómo funciona. Además explique la significancia de los atributos en las gráficas.

   Este experimento muestra como incrementa la tasa de hit al incrementar el tamaño de la caché, lo anterior se debe al principio de localidad temporal, ya que podemos referencias varias veces una dirección almacenada en la caché y al aumentar el tamaño de esta podemos guardar varias direcciones. Asimismo, se puede observar en las gráficas que después de cierto tamaño de caché la tasa de hit permanece igual.

2. ¿Para cada traza, cuál es el tamaño de la memoria indicado para la caché de instrucciones y datos?

Traza	Caché de instrucciones (bytes)	Caché de datos (bytes)
spice	32, 768	16, 384
gcc	131, 072	32, 768
tex	512	32, 768

Tabla 1: Tamaño de la memoria

3.2. Impacto del tamaño del bloque

Se hicieron distintas simulaciones utilizando una caché dividida para instrucciones y datos de 8KB con políticas *write-back* y *write-allocate*. De esta manera, lo que vario fue el tamaño del bloque iniciando en 4 bytes e incrementando en factores de 2 hasta llegar a 4KB.

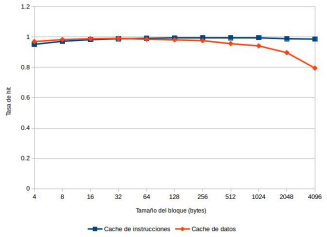

Figura 4: Traza spice

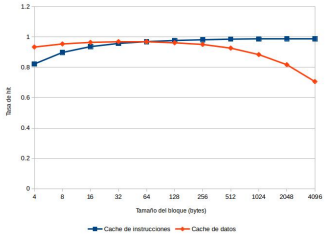

Figura 5: Traza gcc

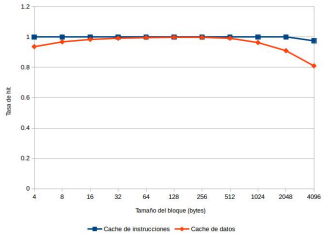

Figura 6: Traza tex

1. Explique porqué las gráficas tienen esa figura, en particular explique la relevancia del principio de localidad espacial en la figura de las curvas.

   En la primera parte de las gráficas, la tasa de hit aumenta un poco debido al principio de localidad espacial, ya que como los bloques son más grandes, se pueden almacenar más referencias; sin embargo, después de cierto tamaño de bloque se puede observar que la tasa de hit disminuye, ya que cada referencia contigua es cada vez menos local (espacialmente) entre ellas.

2. ¿Cuál es el tamaño de bloque óptimo (considere las caché de instrucciones y datos por separado)?

Traza	Caché de instrucciones (bytes)	Caché de datos (bytes)
spice	1024	32
gcc	2048	32
tex	2048	128

Tabla 2: Tamaño del bloque

3 ¿El tamaño óptimo para las referencias de instrucciones y datos es diferente? ¿Qué dice esto sobre la naturaleza de las referencias de instrucciones y datos?

Si son diferentes y esto indica que las referencias de instrucciones muestran un mayor principio de localidad espacial que de datos, posiblemente debido a que la probabilidad de un salto en las instrucciones es bajo y se ejecutan secuencialmente por lo general.

3.3. Impacto de la asociatividad

Se hicieron distintas simulaciones utilizando una caché dividida para instrucciones y datos de 8KB, bloque de 128 bytes con políticas *write-back* y *write-allocate*. De esta manera, lo que vario fue la asociatividad iniciando en 1 e incrementando en factores de 2 hasta llegar a 64.

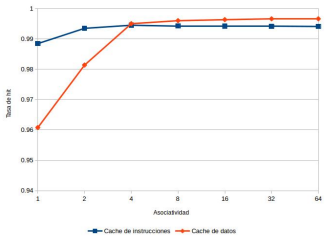

Figura 7: Traza spice

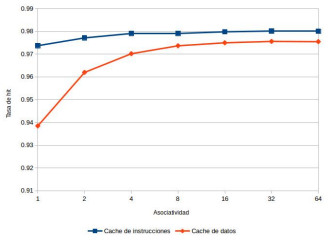

Figura 8: Traza gcc

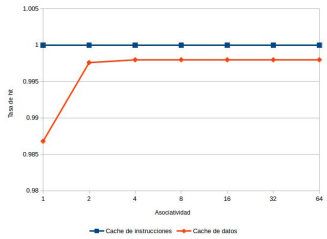

Figura 9: Traza tex

1. Explique porqué las gráficas tienen esa figura.

   En la primera parte de la gráfica la tasa de hit aumenta (más para la de datos) debido a que cada vez se van reduciendo más y más los fallos causados por conflictos y sólo quedan los obligatorios (la primera vez que se referencia una dirección); sin embargo, se llega un punto en la que la asociatividad no ayuda.

2. ¿Existe una diferencia entre las curvas para la referencias de instrucciones y datos? ¿Qué dice esto sobre la diferencia de impacto para cada una?

   Si son diferentes, ya que las referencias de datos son más beneficiadas que las de instrucciones, puesto que estas casi no muestran fallos por conflicto. Lo anterior indica que las referencias de datos tienen mas

conflictos, puesto que pueden estar almacenados en distintos lugares de memoria que pertenecen a la misma línea en la memoria caché.

3.4. Ancho de banda de la memoria

Se hicieron distintas simulaciones utilizando una caché dividida para instrucciones y datos de 8KB y 16KB, tamaño de bloque de 64 bytes y 128 bytes, asociatividad de 2 y 4, por ahora fije la política *write-no-allocate* variando la política *write-through* y *write-back*.

1. ¿Cuál caché tiene menos tráfico y por qué?

   Los fetches demandados permaneces iguales, pero se hacen más copias con la política *write-through*, ya que con la política *write-back* sólo se escribe en memoria principal cuando se elimina una línea de la caché en la que se escribió, mientras que con la política *write-through* siempre se escribe cuando hay un hit.

2. ¿Existe algún escenario en el qué la respuesta anterior cambiaría? Explique.

   No, porque la política *write-through* siempre escribe cuando hay un *hit*, mientras que la otra sólo cuando se tiene que eliminar una línea en la que se escribió.

Ahora, use los mismos parámetros, pero fije la política *write-back* variando la política *write-allocate* y *write-no-allocate*.

1. ¿Cuál caché tiene menos tráfico y por qué?

   No existe alguna política que sea mejor que la otra, ya que depende de la traza que se suministré.

2. ¿Existe algún escenario en el qué la respuesta anterior cambiaría? Explique.

   No se puede concluir algo, puesto que los resultados varian, ya que las trazas que tienen más escrituras en la caché se desempeñan mejor con la política write-allocate,

4. Conclusión

En conclusión, la teoría detrás de la memoria caché y su importancia para un sistema de cómputo veloz se pudo constatar por medio de las simulaciones y experimentos realizados.

Referencias

[1] M Morris Mano. *Arquitectura de computadoras*. Pearson Educación, 1994.

[2] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface*. Newnes, 2013.