FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2: Memorias Caché

6620 - Organización de Computadoras
Primer cuatrimestre de 2020

Alumno	Padrón	Email
Maximiliano Yacobucci	93321	myacobucci@fi.uba.ar
Federico del Mazo	100029	fdelmazo@fi.uba.ar
Ariel Vergara	97010	avergara@fi.uba.ar

Resumen

En este trabajo práctico se realizará una implementación de una memoria caché para familiarizarse con los distintos conceptos alrededor de esta.

1. Introducción

En el siguiente trabajo práctico se realizará una implementación de una memoria caché asociativa con las siguientes características:

- 4KB de capacidad
- bloques de 128 bytes
- política de reemplazo LRU
- poltica de escritura WT/NWA

Teniendo en cuenta el espacio de direcciones de 16 bits, la memoria principal a simular es de 64KB.

Cada bloque de la memoria deberá contener su metadata:

- tag
- bit V
- información necesaria para la política de reemplazo

2. Diseño, implementación y documentación

2.1. Primitivas

Para simular el accionar de la memoria caché se implementaron una serie de primitivas. A continuación se describe el funcionamiento de cada una:

- `void init()`

  Esta función inicializa la memoria principal en cero, como así también la memoria caché poniendo como inválidos sus bloques y en cero al contador de hits y misses.

- `unsigned int get_offset(unsigned int address)`

  Permite obtener el offset de una dirección, teniendo en cuenta el tamaño del bloque.

- `unsigned int find_set(unsigned int address)`

  Obtiene el set correspondiente a una dirección teniendo el cuenta el tamaño del bloque y la cantidad de sets de la caché.

- `unsigned int select_oldest(unsigned int setnum)`

  Obtiene el bloque menos usado recientemente del set comparando la variable counter del bloque y quedándose con el de mayor valor. Esto tiene sentido ya que siempre que se ingresa un nuevo bloque o que se lee un bloque su atributo counter se setea en cero y al de los otros bloques se le suma uno.

- `int compare_tag(unsigned int tag, unsigned int set)`

  Dentro del set indicado va comparando el tag de los bloques. En caso de encontrar el tag especificado por parámetro, devuelve su índice dentro del set, sino retorna -1.

- `void read_tocache(unsigned int blocknum, unsigned int way, unsigned int set)`

  Lee el número de bloque especificado de la memoria principal y lo guarda en la memoria caché en el set y vía indicados. Al guardar setea los atributos de bloque valid en true, el tag correspondiente y el counter en cero.

- `void write_tocache(unsigned int address, unsigned char value)`

  Escribe en caché el valor indicado en el bloque correspondiente a la dirección que se pasa por parámetro.

- `unsigned char read_byte(unsigned int address)`

  Lee el byte que se encuentra en la dirección indicada y lo devuelve. Primero lo busca en caché, si no está va a buscarlo a memoria principal y lo inserta en la caché.

- `void write_byte(unsigned int address, unsigned char value)`

  Escribe el valor en la dirección indicada, si está en caché también lo escribe allí (al ser WT). Si no está en caché no (al ser NWA).

- `float get_miss_rate()`

  Devuelve el porcentaje de misses de la caché. Para esto se contabiliza el total de misses y de hits, y se calcula el porcentaje de misses sobre dicho total.

2.2. Funciones auxiliares

Se han definido diferentes funciones auxiliares que se describen a continuación:

- `unsigned char _read_from_cache(unsigned int set, unsigned int way, unsigned int offset)`

  Lee valor de memoria caché correspondiente al set, vía y offset indicados. A sus vez setea el counter del bloque en cero.

- `void _update_blocks_counter_from_set(unsigned int set)`

  Adiciona uno al atributo counter de todos los bloques de memoria caché del set indicado.

- `unsigned int _get_block(unsigned int address)`

  Obtiene el bloque de memoria caché correspondiente a la dirección indicada.

- `unsigned int _get_tag(unsigned int address)`

  Obtiene el tag del bloque de la memoria caché correspondiente a la dirección indicada.

- `void _update_cache_count(bool data_is_in_cache)`

  Actualiza el valor de hit o miss de la caché dependiendo del valor true o false indicado (si el dato está en caché o no).

2.3. Estructuras de datos

Se definieron tres estructuras de datos para simular el funcionamiento de la memoria caché: la memoria principal, la memoria caché y el bloque de memoria.

La estructura de la memoria caché se compone solo con los datos que posee, asignándole el tamaño correspondiente al total de la memoria (en este caso 64 KB).

La memoria caché está compuesta por los bloques que la conforman, organizados en sets y vías. A su vez posee un contador de misses y uno de hits.

La estructura de cada bloque está conformada por los datos (del tamaño del bloque, en este caso 128 bytes), el tag, el contador counter (utilizado para determinar el bloque más viejo) y una variable de validez.

2.4. Comandos

Los comandos que leerá el programa son los siguientes:

- `FLUSH`

  Vacía la caché llamando a la función init().

- `R ddddd`

  Lee el data de la dirección ddddd llamando a la función read_byte(ddddd) e imprime el resultado.

- `W ddddd, vvvv`

  Escribe el valor vvvv en la dirección ddddd llamando a la función write_byte(ddddd, vvvv) e imprime el resultado.

- `MR`

  Imprime el miss rate de la caché al llamar a la función get_miss_rate().

2.5. Validaciones

El programa leerá de un archivo los comandos a ser ejecutados y llamará a las funciones tal como se indica en la sección anterior. Se harán diversas verificaciones para verificar la validez del archivo a leer. Se verifica que el archivo exista y que los comandos a ejecutar sean válidos, es decir, que respeten el formato indicado. Por otro lado, se chequea que la dirección no sea mayor al tamaño de la memoria principal ya que no tendría sentido.

2.6. Técnicas y procesos de medición

Para medir el miss rate de la memoria caché se poseen los atributos ya descriptos de hits y misses que se van actualizando a medida que se van leyendo y escribiendo datos.

Para establecer si una lectura de una dirección resulta en un miss se calcula el tag y el set correspondiente a la memoria caché. Luego se compara el tag con el de los bloques del set, si se encuentra un match querrá decir que el bloque se encuentra en caché, sino no. En cualquier caso se suma el atributo counter de cada bloque en uno. Si el bloque no está caché se lee de memoria reemplazando el bloque menos usado recientemente (el de mayor valor de counter) y se suma un miss al caché. Si está se lee el dato de la caché y se setea en cero el counter, así como se suma en uno a la cantidad de hits de la caché.

En el caso de la escritura se procede similarmente para verificar si la dirección se encuentra en memoria o no, actualizando el valor de miss/hit. El valor a escribir se escribirá siempre en memoria y en caso de estar en caché se actualizará su valor allí también (y se setea el counter del bloque en cero).

3. Compilación

Para compilar el programa hay que ejecutar el siguiente comando:

```
gcc -g -std=c99 -Wall -Werror main.c cache.h cache.c -o cache
```

Para ejecutar el programa para un cierto archivo con comandos luego de compilarlo:

```
./cache archivo
```

4. Mediciones

Se han realizado diferentes mediciones de la caché, particularmente con el set de pruebas otorgado por la cátedra. Para simplicar la ejecución de estos archivos se creó un archivo Makefile que compilar el programa y lo ejecuta para cada archivo.

De esta manera ejectando make obtenemos las siguientes mediciones:

```
./cache pruebas/prueba1.mem
INIT
WRITE: 0 <- 255
WRITE: 1024 <- 254
WRITE: 2048 <- 248
WRITE: 4096 <- 96
WRITE: 8192 <- 192
READ: 0 -> 255
READ: 1024 -> 254
READ: 2048 -> 248
READ: 8192 -> 192
MISS_RATE: 100.00%
```

En este primer archivo de prueba se puede ver que todos los accesos fallan dado que cada una de las direcciones accedidas pertenecen a bloques diferentes.

```
./cache pruebas/prueba2.mem
INIT
READ: 0 -> 0
READ: 127 -> 0
WRITE: 128 <- 10
READ: 128 -> 10
```

```
WRITE: 128 <- 20
READ: 128 -> 20
MISS_RATE: 50.00%
```

En este caso el segundo read a la posicion 128 es un miss devido a la política WT/NWA dado que no se trae el bloque a memoria al hacer la operación a escritura. Los demás accesos a 128 son hits.

```
./cache pruebas/prueba3.mem
INIT
WRITE: 128 <- 1
WRITE: 129 <- 2
WRITE: 130 <- 3
WRITE: 131 <- 4
READ: 1152 -> 0
READ: 2176 -> 0
READ: 3200 -> 0
READ: 4224 -> 0
READ: 128 -> 1
READ: 129 -> 2
READ: 130 -> 3
READ: 131 -> 4
MISS_RATE: 75.00%
```

En esta prueba, los primeros 4 accesos son al mismo bloque, pero todos generan misses por la política de escritura. Esto no ocurre en los últimos 4 reads dado que el primero trae el bloque a memoria.

```
./cache pruebas/prueba4.mem
INIT
WRITE: 0 <- 255
WRITE: 1 <- 2
WRITE: 2 <- 3
WRITE: 3 <- 4
WRITE: 4 <- 5
READ: 0 -> 255
READ: 1 -> 2
READ: 2 -> 3
READ: 3 -> 4
READ: 4 -> 5
READ: 4096 -> 0
READ: 8192 -> 0
READ: 2048 -> 0
READ: 1024 -> 0
READ: 0 -> 255
READ: 1 -> 2
READ: 2 -> 3
READ: 3 -> 4
READ: 4 -> 5
MISS_RATE: 57.89%
```

En esta ejecución, se da el caso particular que el segundo read a la dirección 0 provoca un miss, ya que la política de reemplazo LRU entró en accion y libero a ese bloque previamente.

```
./cache pruebas/prueba5.mem
INIT
ERR: ADDR_TOO_BIG [131072]
READ: 4096 -> 0
READ: 8192 -> 0
READ: 4096 -> 0
READ: 0 -> 0
READ: 4096 -> 0
MISS_RATE: 60.00%
```

En la prueba 5 se puede ver la reacción del programa al intentar escribir en una dirección de memoria más grande que el tamaño de esta.

```
./cache pruebas/prueba6.mem
INIT
WRITE: 0 <- 1
WRITE: 1024 <- 2
WRITE: 2048 <- 3
WRITE: 4096 <- 4
WRITE: 8192 <- 0
READ: 0 -> 1
READ: 1024 -> 2
READ: 2048 -> 3
READ: 4096 -> 4
READ: 8192 -> 0
MISS_RATE: 100.00%
```

Esta prueba es muy similar a los ejemplos anteriores.

5. Conclusión

Este trabajo práctico nos permitió comprender de manera extensiva el funcionamiento interno de una memoria caché con las características ya especificadas. Para qué sirve el tag, cómo se divide la caché en bloques, sets y vías, lo que se debe hacer y lo que no ante una escritura al ser WT/NWA, y cómo seleccionar el bloque al reemplazar al ser LRU son conceptos que se pusieron en práctica al tener que codificar el funcionamiento de la caché.

66:20 Organización de Computadoras
Trabajo práctico 2: Memorias caché

1. Objetivos

Familiarizarse con el funcionamiento de la memoria caché implementando una simulación de una caché dada.

2. Alcance

Este trabajo práctico es de elaboración grupal, evaluación individual, y de carácter obligatorio para todos alumnos del curso.

3. Requisitos

El trabajo deberá ser entregado en la fecha estipulada, con una carátula que contenga los datos completos de todos los integrantes.

Además, es necesario que el trabajo práctico incluya (entre otras cosas, ver sección 8), la presentación de los resultados obtenidos, explicando, cuando corresponda, con fundamentos reales, las causas o razones de cada resultado obtenido. Por este motivo, el día de la devolución deben concurrir todos los integrantes del grupo.

El informe deberá respetar el modelo de referencia que se encuentra en el grupo, y se valorarán aquellos escritos usando la herramienta TEX / LATEX.

4. Recursos

Este trabajo práctico debe ser implementado en C[2], y correr al menos en Linux.

5. Introducción

La memoria a simular es una caché[1] asociativa por conjuntos de cuatro vías, de 4KB de capacidad, bloques de 128 bytes, política de reemplazo LRU y política de escritura WT/¬WA. Se asume que el espacio de direcciones es

de 16 bits, y hay entonces una memoria principal a simular con un tamaño de 64KB. Estas memorias pueden ser implementadas como variables globales. Cada bloque de la memoria caché deberá contar con su metadata, incluyendo el tag, el bit `V` y la información necesaria para implementar la política de reemplazo LRU.

6. Programa

Se deben implementar las siguientes primitivas:

```
void init()
unsigned int get_offset (unsigned int address)
unsigned int find_set(unsigned int address)
unsigned int select_oldest(unsigned int setnum)
int compare_tag (unsigned int tag, unsigned int set)
void read_tocache(unsigned int blocknum, unsigned int way, unsigned int set)
void write_tocache(unsigned int address, unsigned char value)
unsigned char read_byte(unsigned int address)
void write_byte(unsigned int address, unsigned char value)
float get_miss_rate()
```

- La función `init()` debe inicializar la memoria principal simulada en 0, los bloques de la caché como inválidos y la tasa de misses a 0.
- La función `get_offset(unsigned int address)` debe devolver el *offset* del byte del bloque de memoria al que mapea la dirección `address`.
- La función `find_set(unsigned int address)` debe devolver el conjunto de caché al que mapea la dirección `address`.
- La función `select_oldest()` debe devolver la vía en la que está el bloque más "viejo" dentro de un conjunto, utilizando el campo correspondiente de los metadatos de los bloques del conjunto.
- La función `compare_tag(unsigned int tag, unsigned int set)` debe devolver la vía en la que se encuentra almacenado el bloque correspondiente a `tag` en el conjunto `index`, o -1 si ninguno de los *tags* coincide.
- La función `read_tocache(unsigned int blocknum, unsigned int way, unsigned int set)` debe leer el bloque `blocknum` de memoria y guardarlo en el conjunto y vía indicados en la memoria caché.
- La función `read_byte(unsigned int address)` debe buscar el valor del byte correspondiente a la posición `address` en la caché; si éste no se encuentra en la caché debe cargar ese bloque. El valor de retorno siempre debe ser el valor del byte almacenado en la dirección indicada.

- La función `write_byte(unsigned int address, unsigned char value)` debe escribir el valor `value` en la posición `address` de memoria, y en la posición correcta del bloque que corresponde a `address`, si el bloque se encuentra en la caché. Si no se encuentra, debe escribir el valor solamente en la memoria.
- La función `get_miss_rate()` debe devolver el porcentaje de misses desde que se inicializó la caché.

Con estas primitivas, hacer un programa que llame a `init()` y luego lea de un archivo una serie de comandos y los ejecute. Los comandos tendrán la siguiente forma:

```
FLUSH
R ddddd
W ddddd, vvv
MR
```

- El comando "`FLUSH`" se ejecuta llamando a la función `init()`. Representa el vaciado del caché.
- Los comandos de la forma "`R ddddd`" se ejecutan llamando a la función `read_byte(ddddd)` e imprimiendo el resultado.
- Los comandos de la forma "`W ddddd, vvv`" se ejecutan llamando a la función `write_byte(unsigned int ddddd, char vvv)` e imprimiendo el resultado.
- El comando "`MR`" se ejecuta llamando a la función `get_miss_rate()` e imprimiendo el resultado.

El programa deberá chequear que las líneas del archivo correspondan a un comando con argumentos dentro del rango estipulado, o de lo contrario estar vacías. En caso de que una línea tenga otra cosa que espacios blancos y no tenga un comando válido, se deberá imprimir un mensaje de error informativo.

7. Mediciones

Se deberá incluir la salida que produzca el programa con los siguientes archivos de prueba:

- prueba1.mem
- prueba2.mem
- prueba3.mem

- prueba4.mem
- prueba5.mem
- prueba6.mem

7.1. Documentación

Es necesario que el informe incluya una descripción detallada de las técnicas y procesos de medición empleados, y de todos los pasos involucrados en el mismo, ya que forman parte de los objetivos principales del trabajo.

8. Informe

El informe deberá incluir:

- Este enunciado;
- Documentación relevante al diseño e implementación del programa, incluyendo las estructuras de datos;
- Instrucciones de compilación;
- Resultados de las corridas de prueba;
- El código fuente completo del programa, en formato digital.

9. Fecha de entrega

La fecha de entrega es el jueves 25 de junio de 2020.

Referencias

[1] Hennessy, John L. and Patterson, David A., Computer Architecture: A Quantitative Approach, Third Edition, 2002.

[2] Kernighan, Brian, and Ritchie, Dennis, The C Programming Language.