Trabajo Práctico 2
Memorias caché

Lucas Medrano, *Padrón Nro. 99247*
lucasmedrano97@gmail.com
Federico Álvarez, *Padrón Nro. 99266*
fede.alvarez1997@gmail.com

Grupo Nro. - 2do. Cuatrimestre de 2018
66.20 Organización de Computadoras
Facultad de Ingeniería, Universidad de Buenos Aires

Resumen

En este trabajo se busca implementar en el lenguaje de programación C un tipo de dato que pueda simular el funcionamiento de una memoria caché de 4KB de capacidad, asociativa por conjuntos de cuatro vías, con política de reemplazo LRU y política de escritura Write Back/Write Allocate, junto con la memoria principal asociada. El objetivo de esta simulación será observar y documentar su comportamiento, para una mejor comprensión de dicho funcionamiento.

Índice

1. Desarrollo

Para interactuar con la memoria caché se desarrollará un programa capaz de leer de un archivo que contendrá instrucciones de lectura y escritura en una posición de memoria, y un comando para obtener el *miss rate*. Éste recibirá como parámetro el nombre del archivo que contiene las instrucciones a ser ejecutadas.

1.1. Implementación

1.1.1. Programa

El programa tomará como único parámetro del nombre del archivo y, en caso de que no exista, imprimirá por pantalla un mensaje de error acorde.

Ante cada escritura se mostrará un mensaje indicando si se pudo realizar, y ante cada lectura se mostrará el valor leído. En el caso de que una línea del archivo contenga texto que no se corresponde con ninguno de los comandos especificados, simplemente se la ignorará y pasará a la siguiente. Sin embargo, en caso de que la instrucción sea válida pero la dirección o el valor a escribir no, se imprimirá un mensaje por pantalla.

1.1.2. Memoria caché

La estructura de la memoria caché constará de dos subpartes principales, los *sets* y los *bloques*. Cada set contendrá cuatro bloques, y los bloques, ademas de los datos a almacenar en memoria, contendrán toda la metadata necesaria para realizar las tareas de lectura, escritura, reemplazo y *write back*. Todas las posiciones de memoria se inicializarán a `0`, y se asume que tanto las direcciones como los valores contenidos en ellas no pueden ser negativos.

Por otra parte, habrá también una memoria principal del tamaño especificado que interactuará con el caché cuando sea necesario. Cabe aclarar que será la estructura del caché la que decidirá qué bloque reemplazar cuando un set esté lleno, mediante un seguimiento del bloque menos recientemente utilizado, y cuándo escribir los contenidos de dicho bloque en memoria, mediante un chequeo al bit *dirty*.

Llevará cuenta, además, de la cantidad de misses ocurridos, y de accesos a memoria totales, para presentar así el miss rate.

2. Mediciones

A continuación se muestran las salidas y tiempos de corrida de los cinco archivos de prueba. Cabe aclarar que las operaciones de escritura consumen más tiempo ya que requieren del procesamiento de un parámetro extra.

Al correr el programa con el archivo `prueba.mem` se observa la siguiente salida.

```
$./cache prueba1.mem
El valor 255 se escribio correctamente
El valor 254 se escribio correctamente
El valor 248 se escribio correctamente
El valor 96 se escribio correctamente
El valor 192 se escribio correctamente
Read 255
Read 254
Read 248
Read 192
Miss rate: 88%
```

El miss rate obtenido es del 88 %, ya que en ocho de los nueve accesos a memoria se produce un *miss*. Esto se debe a que todas las direcciones ingresadas mapean al mismo set, y con la política de reemplazo LRU, se van reemplazando una a una.

A su vez, utilizando el comando `time` de la consola, se observa el siguiente tiempo.

```
real 0m0,003s
```

Cabe destacar que las instrucciones de este archivo implican que además de cargar en el caché el bloque correspondiente, se debe escribir el contenido del bloque reemplazado en memoria, lo que consume más tiempo.

Al ingresar ahora el archivo `prueba.mem`, se observa el siguiente miss rate, junto con el siguiente tiempo.

```
$ time ./cache prueba2.mem
Read 0
Read 0
El valor 10 se escribio correctamente
Read 10
El valor 20 se escribio correctamente
Read 20
Miss rate: 33%

real 0m0,001s
```

Lo cual es consisente, ya que no sólo se registran menos misses, sino que también se leen menos instrucciones y no se escribe ningún bloque en la memoria principal.

En el caso de `prueba3.mem`:

```
$ time ./cache prueba3.mem
El valor 1 se escribio correctamente
El valor 2 se escribio correctamente
El valor 3 se escribio correctamente
El valor 4 se escribio correctamente
Read 0
Read 0
Read 0
Read 0
Read 1
Read 2
Read 3
Read 4
Miss rate: 50%

real 0m0,002s
```

Se registra un tiempo mayor al de la prueba anterior, que se condice tanto con el mayor porcentaje de misses registrado, como con la mayor cantidad de instrucciones presentes.

Para la prueba 4:

```
$ time ./cache prueba4.mem
El valor 256 no es valido
El valor 2 se escribio correctamente
El valor 3 se escribio correctamente
El valor 4 se escribio correctamente
El valor 5 se escribio correctamente
Read 0
Read 2
Read 3
Read 4
Read 5
Read 0
Read 0
Read 0
Read 2
Read 3
Read 4
Read 5
Miss rate: 18%

real 0m0,002s
```

En este caso se obtiene un tiempo similar al anterior, lo que se explica

observando la gran cantidad de instruccuiones ejecutadas con tan bajo porcentaje de misses, y teniendo en cuenta que los valores inválidos no se procesan y tampoco se deben realizar ni reemplazos ni escrituras de bloques en memoria.

Por último, al correr `prueba5.mem` se registra:

```
$ time ./cache prueba5.mem
La direccion 131072 no es valida
Read 0
Read 0
Read 0
Read 0
Read 0
Miss rate: 60%

real 0m0,001s
```

El tiempo es bajo porque las intrucciones son pocas y la primera instrucción no se procesa.

3. Conclusiones

La realización de este trabajo resultó importante para comprender el funcionamiento de la memoria caché y todas las operaciones involucradas en el proceso de almacenar o leer un dato de memoria. Permitió principalmente visualizar la utilidad de este tipo de estructuras y realizar un profundo seguimiento de las acciones que realiza cada parte por separado. Además fue útil para entender la función de cada uno de los elementos de la metadata contenida en cada bloque, así como del *tag*, *index* y *offset* que componen a la dirección de memoria.

Por último, el desarrollo sirvió para observar en la práctica el funcionamiento de esto, al medir el tiempo de ejecución en cada prueba, observar las diferencias entre estas y relacionarlas con factores como el *miss rate*, la cantidad de reemplazos de bloque realizados, o la cantidad de escritura de bloques en memoria al ser reemplazados.

Se entregan junto con este informe todos los archivos de código fuente y headers que componen el programa.

4. Apéndice

4.1. Enunciado

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 personalmente, 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 entrega 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, 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 64 bytes, política de reemplazo LRU y política de escritura WB/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 bit D, el *tag*, y un campo que permita implementar la política de LRU.

6. Programa

Se deben implementar las siguientes primitivas:

```
void init()
int find_set(int address)
int find_lru(int setnum)
int is_dirty(int way, int setnum)
void read_block(int blocknum)
void write_block(int way, int setnum)
int read_byte(int address)
int write_byte(int address, char value)
int get_miss_rate()
```

- La función `init()` debe inicializar los bloques de la caché como inválidos y la tasa de misses a 0.
- La función `find_set(int address)` debe devolver el conjunto de caché al que mapea la dirección `address`.
- La función `find_lru(int setnum)` debe devolver el bloque menos recientemente usado dentro de un conjunto (o alguno de ellos si hay más de uno), utilizando el campo correspondiente de los metadatos de los bloques del conjunto.
- La función `is_dirty(int way, int blocknum)` debe devolver el estado del bit D del bloque correspondiente.
- La función `read_block(int blocknum)` debe leer el bloque `blocknum` de memoria y guardarlo en el lugar que le corresponda en la memoria caché.
- La función `write_block(int way, int setnum)` debe escribir los datos contenidos en el bloque `setnum` de la vía `way`.
- La función `read_byte(address)` debe retornar el valor correspondiente a la posición `address`.
- La función `write_byte(int address, char value)` debe escribir el valor `value` en la posición correcta del bloque que corresponde a `address`.

- La función `get_miss_rate()` debe devolver el porcentaje de misses desde que se inicializó el cache.
- `read_byte()` y `write_byte()` sólo deben interactuar con la memoria a través de las otras primitivas.

Con estas primitivas (más las que hagan falta para manejar LRU y WB/WA), hacer un programa que lea de un archivo una serie de comandos, que tendrán la siguiente forma:

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

- 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(int ddddd, char vvv)` e imprimiendo el resultado.
- Los comandos de la forma "`MR`" se ejecutan llamando a la función `get_miss_rate()` e imprimiendo el resultado.

El programa deberá chequear que los valores de los argumentos a los comandos estén dentro del rango de direcciones y valores antes de llamar a las funciones, e imprimir un mensaje de error informativo cuando corresponda.

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

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;
- Análisis de la performance del programa tal como está presentado, para el caso de prueba.
- El código fuente completo del programa modificado, en dos formatos: digital e impreso en papel.

9. Fecha de entrega

La última fecha de entrega y presentación es el jueves 1 de Noviembre de 2018.

Referencias

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