Asignatura:
66.20 Organización de computadoras

Profesor: Hamkalo, José Luis

Trabajo Práctico:
Conjunto de instrucciones MIPS

Nombre y apellido	Padrón	Correo electrónico	Slack
Julian Mejliker	100866	jmejliker@gmail.com	Julian Mejliker
Tomas Nocceti	100853	nocetti.tomas@gmail.com	Tomas Nocetti
Netanel Jamilis	99093	njamilis@fi.uba.ar	Netanel Jamilis

Primer cuatrimestre 2019

Índice

1. Introducción

El objetivo del presente trabajo práctico es desarrollar en el lenguaje de programación C, un simulador de memoria caché con politica de reemplazo LRU y política de escritura WT/ WA.

2. Modelo utilizado

2.1. Block

La estructura más *atómica* de nuestro modelo. Define una bloque de memoria de 128 Bytes. La definición de su estructura es la siguiente

```
struct block {
    char *bytes;
    uint8_t valid;
    uint8_t tag;
    uint8_t last_used;
}
```

Donde se define una referencia a los datos que contiene la misma, un campo *valid* utilizado por el cache para saber si el bloque de memoria actual que guarda el caché representa un bloque válido traido desde la memoria, un campo *tag* donde se guarda el tag de la direccion de memoria del bloque, y un iterador que se acuaiza a 0 cada vez que se accede al bloque en cuestion o se suma uno, cada vez que se accede al set donde recide pero a otro bloque.

2.2. Memory

Define la estructura que representa la memoria principal de la máquina. Esta estructura a su vez, está compuesta por bloques *(block_t)*. Para el objetivo de este trabajo práctico, ocupa un espacio de 64 KB de memoria. La definición de la estructura es la siguiente:

```
struct memory {
    block_t **memory;
};
```

Dónde estamos definiendo que una memoria es un puntero a direcciones de memoria *(block_t)*

2.3. Cache

Define la estructura de nuestro caché. Este está compuesto por conjuntos *(set_t)*. Para el objetivo planteado para este tp, el caché puede almacenar un espacio total de 4 KB de memoria. La definición de la estructura es la siguiente:

```
struct cache {
    memory_t *pri_mem;
    set_t **sets;
```

```
    size_t hits;
    size_t miss;
} cache_t;
```

Donde tenemos una referencia a la memoria principal a la cuál estamos *cacheando*, una referencia a direcciones de memoria de conjuntos, y el indice de hits y miss de la propia cache.

2.4. Set

Define la estructura que representa un conjunto de nuestro cache. Esta estructura a su vez está compuesta por bloques, al igual que la memoria principal *(block_t)*. El conjunto tiene 4 vias. Es decir, que puede almacenar 4 bloques de memoria de forma simultánea.
Cuando se inserta un nuevo bloque de memoria, se ingresa el bloque en la primer via disponible.

Cuando todas las vias están ocupadas, se quita el conjunto que hace mas tiempo fue utilizado, en esta implementacion seria el que posee el atributo last_used mayor.

La definición de la estructura utilizada es la siguiente:

```
struct set {
    block_t **blocks;
    int blocks_inserted;
};
```

Posee un contador de la cantidad de bloques insertados en el conjunto actual, para saber cuando desencolar un elemento de la cola.

3. Estructura de memoria utilizada

Nos estamos manejando con una memoria que tiene direcciones 16 bits. La cual para utilizarla con nuestro cache la dividimos de la siguiente manera:

3.1. Offset

En nuestro cache, los bloques de memoria ocupan un espacio de 128 bits. Es decir 2^7 bits. Por lo tanto, los 7 bits menos significativos de la dirección de memoria corresponden al offset.

3.2. Index

Nuestro caché tiene la capacidad de guardar 4 Kb de memoria. En cada bloque estamos guardando 128 bytes y tenemos 4 vias disponibles por conjunto. Por lo tanto 4 kb /(128 bytes $*8$) $= 8 = 2^3$ es la cantidad de indices disponibles dentro de nuestro caché. Es decir, necesitamos 3 bits para direccionar a los conjuntos del caché.

3.3. Tag

El tag corresponde a los 7 bits restantes. Se utiliza para verificar si la dirección de memoria dentro de nuestro cache corresponde a la que estamos buscando.

4. Casos de prueba

Corriendo todos los test obtuvimos lo siguiente. En cada caso se indica la prueba corrida, y el missrate:

```
ejercicio 1
write value 255 to addr 0
write value 254 to addr 1024
write value 248 to addr 2048
write value 96 to addr 4096
write value 192 to addr 8192
read addr 0
read 255
read addr 1024
read 254
read addr 2048
read 248
read addr 8192
read 192
miss rate: 1.00

real 0m0.004s
user 0m0.004s
sys 0m0.000s

ejercicio 2
read addr 0
read 0
read addr 127
read 0
write value 10 to addr 128
read addr 128
read 10
write value 20 to addr 128
read addr 128
read 20
miss rate: 0.67

real 0m0.005s
user 0m0.005s
sys 0m0.000s
```

```
ejercicio 3
write value 1 to addr 128
write value 2 to addr 129
write value 3 to addr 130
write value 4 to addr 131
read addr 1152
read 0
read addr 2176
read 0
read addr 3200
read 0
read addr 4224
read 0
read addr 128
read 1
read addr 129
read 2
read addr 130
read 3
read addr 131
read 4
miss rate: 0.83

real 0m0.005s
user 0m0.000s
sys 0m0.005s

ejercicio 4
write value 255 to addr 0
write value 2 to addr 1
write value 3 to addr 2
write value 4 to addr 3
write value 5 to addr 4
read addr 0
read 255
read addr 1
read 2
read addr 2
read 3
read addr 3
read 4
read addr 4
read 5
read addr 4096
read 0
read addr 8192
read 0
```

```
read addr 2048
read 0
read addr 1024
read 0
read addr 0
read 255
read addr 1
read 2
read addr 2
read 3
read addr 3
read 4
read addr 4
read 5
miss rate: 0.68

real 0m0.004s
user 0m0.000s
sys 0m0.004s

ejercicio 5
read addr 131072
read 0
read addr 4096
read 0
read addr 8192
read 0
read addr 4096
read 0
read addr 0
read 0
read addr 4096
read 0
miss rate: 0.50

real 0m0.004s
user 0m0.000s
sys 0m0.005s

ejercicio 6
write value 1 to addr 0
write value 2 to addr 1024
write value 3 to addr 2048
write value 4 to addr 4096
write value 0 to addr 8192
read addr 0
read 1
```

```
read addr 1024
read 2
read addr 2048
read 3
read addr 4096
read 4
read addr 8192
read 0
miss rate: 1.00

real 0m0.004s
user 0m0.000s
sys 0m0.004s
```

5. Política de reemplazo

Con respecto a la política de escritura, se dan dos casos:

- Si hay un hit, se escribe en la cache y luego en la memoria.
- Si hay un miss, se escribe solo en la memoria principal y el bloque no es traido a memoria ya que la cache es "NoWriteAllocate".

Con respecto a la política de lectura, se da don casos tambien:

- Si hay un hit, se lee el dato de la cache.
- Si hay un miss, se reemplaza si es que es necesario y se trae el bloque de memoria donde se encuentra el dato en la MP y se procede con la lectura desde la Cache.

6. Conclusión

A través de este trabajo práctico, comprendimos como funciona la implementación de una memoria cache a través de la simulación mediante software. Como conclusión, decimos que este trabajo práctico nos ayudó a construir la idea de que la caché es una abstracción de un sistema de manejo de memoria, y que no es algo atado a la implementación de un hardware, sino que es una herramienta aplicable a varios esquemas de distintas maneras, a todos los niveles de memoria existentes. Es decir, la memoria caché no es simplemente un hardware al lado del CPU de la computadora, sino que es una idea abstracta aplicable a múltiples escenarios.

Con respecto al algoritmo de reemplazo, si se da como en la teoria de cercania de espacio, este es un muy buen algoritmo ya que los que hace menos tiempo se utilizan se mantienen en la cache y asi seran accedidos con un tiempo menor de acceso.