Trabajo Práctico 2
Memorias Caché

Tomás Lubertino, *Padrón 95140*
tomas.lubertino@gmail.com

Maximiliano Suppes, *Padrón 96938*
maxisuppes@gmail.com

Zuretti Agustin, *Padrón 95605*
zurettiagustin@gmail.com

1er. Cuatrimestre de 2020
86.37 / 66.20 Organización de Computadoras – Práctica Jueves
Facultad de Ingeniería, Universidad de Buenos Aires

1. Introducción

Este trabajo práctico tiene como objetivo familiarizarse con el funcionamiento de la memoria caché a partir de la simulación de una cache dada:

- Asociativa por conjuntos de 4 vías;
- 4KB de capacidad;
- Bloques de 128 bytes;
- Política de reemplazo LRU;
- Política de escritura WT/¬WA.

2. Desarrollo

2.1. Primitivas

Para simular el funcionamiento de la memoria caché se implementaron una serie de primitivas de acuerdo al enunciado del trabajo práctico:

- `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()`

2.2. Auxiliares

A continuación una lista de funciones auxiliares que se utilizaron:

- `unsigned char read_cache(unsigned int set, unsigned int way, unsigned int offset)`
- `void inc_block_counter(unsigned int set)`
- `unsigned int get_block(unsigned int address)`
- `unsigned int get_tag(unsigned int address)`
- `void inc_rates(bool isHit)`

2.3. Estructuras de datos

Se definieron 3 estructuras de datos para simular el funcionamiento de la memoria caché:

- La memoria principal (64KB);
- La memoria cache (4WSA) con contador de miss y hits;
- El bloque de memoria (128B) con su meta-data, tag, variable de validez y un contador para llevar a cabo la política LRU;

2.4. Comandos

El programa leerá los siguientes comandos:

- FLUSH

  Vacía la memoria cache llamando a init().

- R ddddd

  Lee el dato de la dirección ddddd e imprime el resultado (`read_byte(ddddd)`).

- W ddddd, vvvv

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

- MR

  Imprime el miss rate de la memoria cache (`get_miss_rate()`).

2.5. Validaciones y proceso de medición

El programa leerá de un archivo los comandos descriptos en la sección anterior y llamara a las funciones correspondientes. Se aplicaran las siguientes validaciones:

- Existencia del archivo;
- Formato de los comandos a ejecutar;
- Direcciones que no superen el tamaño de la memoria principal;

Para definir si una lectura resulta en miss se procede a calcular el set y tag. Luego se compara el tag con el de los bloques del set y en caso de que se encuentre una coincidencia, significara que el bloque se encuentra en la cache. En caso contrario, se procede a leer el bloque de la memoria y se reemplaza el bloque correspondiente aplicando la política LRU (bloque menos usado recientemente) y se suma un miss al cache. Para decidir esto, utilizamos un contador que determina el uso del bloque a partir de su valor. Es decir, el bloque con el contador de mayor valor sera el bloque a reemplazar. Una vez obtenido el bloque, se resetea el contador en cero. En todos los casos se incrementa el contador de todos los bloques en uno.

Para definir si una escritura resulta en miss, se procede de igual manera que en el caso de la lectura. Por la política de escritura (WT), el valor se escribirá siempre en memoria principal y en caso de que se encuentre también en cache, se actualizara el valor (se resetea el contador del bloque en cero).

3. Proceso de compilación y ejecución

La compilación del archivo fuente se hace con el siguiente comando:

```
$ gcc TP2.c -o tp2
```

Para correr el programa junto con un archivo de prueba, se debe utilizar el siguiente comando:

```
$ ./tp2 pruebaX.mem
```

4. Mediciones y análisis

A continuación se detallan las salidas del programa para cada archivo de pruebas.

4.1. Prueba 1

```
./tp2 prueba1.mem

------------------------------------------------------------
Se escribió en la dirección 0, el dato => 255
Se escribió en la dirección 1024, el dato => 254
Se escribió en la dirección 2048, el dato => 248
Se escribió en la dirección 4096, el dato => 96
Se escribió en la dirección 8192, el dato => 192
Se leyó en la dirección 0, el dato => 255
Se leyó en la dirección 1024, el dato => 254
Se leyó en la dirección 2048, el dato => 248
Se leyó en la dirección 8192, el dato => 192
MISS RATE: 100.00%
------------------------------------------------------------

OBS: Todos los accesos fallan ya que las direcciones accedidas pertenecen a bloques diferentes.
```

4.2. Prueba 2

```
./tp2 prueba2.mem

------------------------------------------------------------
Se leyó en la dirección 0, el dato => 0
Se leyó en la dirección 127, el dato => 0
Se escribió en la dirección 128, el dato => 10
Se leyó en la dirección 128, el dato => 10
Se escribió en la dirección 128, el dato => 20
Se leyó en la dirección 128, el dato => 20
MISS RATE: 50.00%
------------------------------------------------------------

OBS: El segundo read a la direccion 128 es MISS por ser WT/¬WA. Los demas accesos a 128 son hits.
```

4.3. Prueba 3

```
./tp2 prueba3.mem

------------------------------------------------------------
Se escribió en la dirección 128, el dato => 1
Se escribió en la dirección 129, el dato => 2
Se escribió en la dirección 130, el dato => 3
Se escribió en la dirección 131, el dato => 4
Se leyó en la dirección 1152, el dato => 0
Se leyó en la dirección 2176, el dato => 0
Se leyó en la dirección 3200, el dato => 0
Se leyó en la dirección 4224, el dato => 0
Se leyó en la dirección 128, el dato => 1
Se leyó en la dirección 129, el dato => 2
Se leyó en la dirección 130, el dato => 3
Se leyó en la dirección 131, el dato => 4
MISS RATE: 75.00%
------------------------------------------------------------
```

OBS: Las escrituras se hacen al mismo bloque y todas resultan en miss por ser WT/¬WA.

4.4. Prueba 4

```
./tp2 prueba4.mem

------------------------------------------------------------
Se escribió en la dirección 0, el dato => 255
Se escribió en la dirección 1, el dato => 2
Se escribió en la dirección 2, el dato => 3
Se escribió en la dirección 3, el dato => 4
Se escribió en la dirección 4, el dato => 5
Se leyó en la dirección 0, el dato => 255
Se leyó en la dirección 1, el dato => 2
Se leyó en la dirección 2, el dato => 3
Se leyó en la dirección 3, el dato => 4
Se leyó en la dirección 4, el dato => 5
Se leyó en la dirección 4096, el dato => 0
Se leyó en la dirección 8192, el dato => 0
Se leyó en la dirección 2048, el dato => 0
Se leyó en la dirección 1024, el dato => 0
Se leyó en la dirección 0, el dato => 255
Se leyó en la dirección 1, el dato => 2
Se leyó en la dirección 2, el dato => 3
Se leyó en la dirección 3, el dato => 4
Se leyó en la dirección 4, el dato => 5
MISS RATE: 57.89%
------------------------------------------------------------
```

OBS: En la segunda lectura a la dirección 0 se genera un miss por la política de reemplazo L

4.5. Prueba 5

```
./tp2 prueba5.mem

------------------------------------------------------------
Error: La dirección 131072 es muy grande.
Se leyó en la dirección 4096, el dato => 0
Se leyó en la dirección 8192, el dato => 0
Se leyó en la dirección 4096, el dato => 0
Se leyó en la dirección 0, el dato => 0
Se leyó en la dirección 4096, el dato => 0
MISS RATE: 60.00%
------------------------------------------------------------
```

OBS: Se ve como el programa muestra un error ya que se intento escribir una dirección mas gr

4.6. Prueba 6

```
./tp2 prueba6.mem

------------------------------------------------------------
Se escribió en la dirección 0, el dato => 1
Se escribió en la dirección 1024, el dato => 2
Se escribió en la dirección 2048, el dato => 3
Se escribió en la dirección 4096, el dato => 4
Se escribió en la dirección 8192, el dato => 0
Se leyó en la dirección 0, el dato => 1
Se leyó en la dirección 1024, el dato => 2
Se leyó en la dirección 2048, el dato => 3
Se leyó en la dirección 4096, el dato => 4
Se leyó en la dirección 8192, el dato => 0
MISS RATE: 100.00%
------------------------------------------------------------
```

5. Código fuente

TP2.c

```c
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>

#define MAIN_MEMORY_SIZE (64 * 1024) // 64 KB
#define CACHE_SIZE (4 * 1024)
#define BLOCK_SIZE 128
#define SET_AMOUNT 8
#define WAYS_AMOUNT 4
#define INVALID 0
#define VALID 1

#define BLOCKS_COUNT (CACHE_SIZE / WAYS_AMOUNT / BLOCK_SIZE)
#define ADDRESS 16
#define OFFSET (log(BLOCK_SIZE) / log(2))
#define INDEX (log(SET_AMOUNT) / log(2))
#define TAG (ADDRESS - INDEX - OFFSET)

typedef struct {
    unsigned char data[MAIN_MEMORY_SIZE];
} main_memory_t;

typedef struct {
    unsigned int tag;
    int counter;
    bool valid;
    unsigned char data[BLOCK_SIZE];
} cache_block_t;

typedef struct {
    cache_block_t blocks[SET_AMOUNT][WAYS_AMOUNT];
    int miss_count;
    int hit_count;
} cache_t;

main_memory_t main_memory;
cache_t cache;
```

```
//-------------------------------------AUX-----------------------------------------//

unsigned char read_cache(unsigned int set, unsigned int way, unsigned int offset) {
    cache_block_t block = cache.blocks[set][way];
    block.counter = 0;

    return block.data[offset];
}

void inc_block_counter(unsigned int set) {
    cache_block_t *blocks_in_set = cache.blocks[set];
    for (size_t i = 0; i < WAYS_AMOUNT; i++) {
        blocks_in_set[i].counter += 1;
    }
}

unsigned int get_block(unsigned int address) {
   return address / BLOCK_SIZE;
}

unsigned int get_tag(unsigned int address) {
    unsigned int int_for_set_and_offset = pow(2, INDEX + OFFSET);
    return address / int_for_set_and_offset;
}

void inc_rates(bool isHit) {
    if (isHit) {
      cache.hit_count += 1;
    } else {
        cache.miss_count += 1;
    }
}

//-------------------------------------MAIN----------------------------------------//

void init() {

   printf("------------------------------------------------------------- \n");
    memset(main_memory.data, 0, MAIN_MEMORY_SIZE);

    memset(cache.blocks, INVALID, sizeof(cache.blocks));
    cache.miss_count = 0;
    cache.hit_count = 0;
}

unsigned int find_set(unsigned int address) {
   //Devuelve el conjunto de cache al que mapea la direccion address.

    return address / BLOCK_SIZE % SET_AMOUNT;
}

unsigned int get_offset(unsigned int address) {
   //Devuelve el o[U+FFFD]set del byte del bloque de memoria al que mapea la direccion
       address.

   return address % BLOCK_SIZE;
}

unsigned int select_oldest(unsigned int set) {
   //Devuelve la v[U+FFFD]a en la que esta el bloque mas [U+FFFD]viejo[U+FFFD] dentro de
       un conjunto,
   //utilizando el campo correspondiente de los metadatos de los bloques del conjunto.

   cache_block_t *blocks_in_set = cache.blocks[set];
    unsigned int oldest = 0;
```

```
    for (size_t i = 1; i < WAYS_AMOUNT; i++) {
        if (blocks_in_set[oldest].counter < blocks_in_set[i].counter) {
            oldest = i;
        }
    }
    return oldest;
}

void read_tocache(unsigned int blocknum, unsigned int way, unsigned int set) {
   //lee el bloque blocknum de memoria y guardarlo en el conjunto y
   //via indicados en la memoria cache.

   int block_starting_address = blocknum * BLOCK_SIZE;
    cache_block_t *block = &cache.blocks[set][way];
    for (int i = 0; i < BLOCK_SIZE; i++) {
        block->data[i] = main_memory.data[block_starting_address + i];
    }
    block->valid = VALID;
    block->tag = get_tag(block_starting_address);
    block->counter = 0;
}

signed int compare_tag(unsigned int tag, unsigned int set) {
   //devuelve la v[U+FFFD]a en la que se encuentra almacenado el bloque correspondiente
   //a tag en el conjunto index, o -1 si ninguno de los tags coincide.

   cache_block_t *blocks_in_set = cache.blocks[set];
    cache_block_t block;
    for (int i = 0; i < WAYS_AMOUNT; i++) {
        block = blocks_in_set[i];
        if (block.valid && block.tag == tag) {
            return i;
        }
    }
    return -1;
}

void write_tocache(unsigned int address, unsigned char value) {
   //Como la cache es WT/[U+FFFD]WA, si escribimos en la cache, esto nos garantiza que el
       bloque correspondiente a la
   //direccion se encuentra en la misma.

    unsigned int tag = get_tag(address);
    unsigned int set = find_set(address);
    unsigned int offset = get_offset(address);
    int way = compare_tag(tag, set);
    cache_block_t *block = &cache.blocks[set][way];
    block->data[offset] = value;
    block->counter = 0;
}

unsigned char read_byte(unsigned int address) {
   /*busca el valor del byte correspondiente a la posicion
   address en la cache; si este no se encuentra en la cache debe cargar ese bloque.
   El valor de retorno siempre debe ser el valor del byte almacenado en la direccion
   indicada.*/

   unsigned int tag = get_tag(address);
    unsigned int set = find_set(address);
    int way = compare_tag(tag, set);
    unsigned char byte_read;
    bool is_in_cache = way >= 0;
    inc_block_counter(set);
    if (!is_in_cache) {
        unsigned int block_num = get_block(address);
        way = select_oldest(set);
```

```
        read_tocache(block_num, way, set);
    }

    unsigned int offset = get_offset(address);
    byte_read = read_cache(set, way, offset);
    inc_rates(is_in_cache);

    printf("Se ley[U+FFFD] en la direcci[U+FFFD]n %d, el dato => %u \n", address,
        byte_read);
    return byte_read;
}

void write_byte(unsigned int address, unsigned char value) {
   /*Escribe el valor value en la posicion address de memoria, y en la posicion
   correcta del bloque que corresponde a address, si el bloque se encuentra en la cache.
   Si no se encuentra, debe escribir el valor solamente en la memoria*/

   unsigned int tag = get_tag(address);
    unsigned int set = find_set(address);
    int way = compare_tag(tag, set);
    bool is_in_cache = way >= 0;

    inc_block_counter(set);
    main_memory.data[address] = value;
    if (is_in_cache) {
        write_tocache(address, value);
    }
    inc_rates(is_in_cache);
    printf("Se escribi[U+FFFD] en la direcci[U+FFFD]n %d, el dato => %u \n", address,
        value);
}

float get_miss_rate() {
   //Devuelve el porcentaje de misses desde que se inicializo la cache.

   if (cache.miss_count == 0) return 0;
    int total_count = cache.miss_count + cache.hit_count;
    float mr = cache.miss_count / (float)total_count * 100;
    printf("MISS RATE: %0.2f%% \n", mr);
    printf("------------------------------------------------------------ \n");
    return mr;
}

//-----------------------------------------------------------------------------//

static int read_address(char *line) {
    unsigned int address;
    int result = sscanf(line, "R %u", &address);
    if (result != 1) {
        fprintf(stderr, "Error al leer la direcci[U+FFFD]n.\n");
        return 1;
    } else if (address >= MAIN_MEMORY_SIZE) {
        fprintf(stderr, "Error: La direcci[U+FFFD]n %d es muy grande. \n", address);
    } else {
        read_byte(address);
    }
    return 0;
}

static int write_address(char *line) {
    unsigned int address;
    unsigned int value;
    int result = sscanf(line, "W %u, %u", &address, &value);
    if (result != 2) {
        fprintf(stderr, "Error al leer la direcci[U+FFFD]n y el valor.\n");
        return 1;
```

```
    } else if (address >= MAIN_MEMORY_SIZE) {
        fprintf(stderr, "Error: La direcci[U+FFFD]n %d es muy grande. \n", address);
    } else {
        write_byte(address, value);
    }
    return 0;
}

int parser(const char *filename) {
   FILE *commands_file;
    if (!(commands_file = fopen(filename, "r"))) {
        fprintf(stderr, "Error al abrir el archivo.\n");
        return 1;
    }

    char *line = NULL;
    size_t len = 0;
    int result = 0;

    while (getline(&line, &len, commands_file) != -1) {
        if (strncmp(line, "FLUSH", 5) == 0) {
            init();
        } else if (strncmp(line, "R", 1) == 0) {
            result = read_address(line);
        } else if (strncmp(line, "W", 1) == 0) {
            result = write_address(line);
        } else if (strncmp(line, "MR", 2) == 0) {
            get_miss_rate();
        } else {
            fprintf(stderr, "Error: l[U+FFFD]nea inv[U+FFFD]lida.\n");
            result = 1;
        }

        if (result == 1) return result;
    }
    fclose(commands_file);
    return result;
}

int main(int argc, char* argv[]) {
   if (argc != 2) {
        fprintf(stderr, "Argunmentos invalidos.\n");
        return 1;
    }

    init();

    int result = parser(argv[1]);

    return result;
}
```

6. Conclusión

Este trabajo práctico nos ayudó a familiarizarnos con el funcionamiento básico de una memoria caché y a comprender la estructura de datos de la misma. Por otro lado, a través de las pruebas pudimos ver qué tan importantes son las políticas de reemplazo y de escritura y cuán influyentes son a la hora del armado de una memoria caché.

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.