.UBAfiuba
FACULTAD DE INGENIERÍA

Trabajo Práctico 2

[66.20/86.37] Organización de Computadoras
Curso 2
Segundo cuatrimestre de 2020

Alumnos	Padrón	Correo electrónico	Slack
Gómez, Joaquín	103735	joagomez@fi.uba.ar	Joaquín Gomez
Grassano, Bruno	103855	bgrassano@fi.uba.ar	Bruno Grassano
Romero, Adrián	103371	adromero@fi.uba.ar	Adrián Romero

Índice

1. Introducción

El objetivo del presente trabajo es simular, en un programa escrito en C, el funcionamiento de una memoria caché asociativa por conjuntos con las siguientes características:

- Se deberá poder pasar por parámetro el número de vías, la capacidad y el tamaño de bloque.
- La política de reemplazo de la memoria sera LRU y la política de escritura será WB/WA.
- El espacio de direcciones será de 16 bits, y por lo tanto se simulará también una memoria principal con un tamaño de 64KB.
- Cada bloque de la memoria caché deberá contar con su metadata, incluyendo el bit V, el bit D, el tag, y un campo que permita implementar la política de LRU.
- Se deberá leer de un archivo cuales son las lecturas y escrituras que se hacen sobre la memoria.

2. Diseño e implementación

Para el desarrollo del programa hemos considerado los siguientes aspectos:

2.1. Estructuras creadas

Para la realización del trabajo se crearon tres diferentes estructuras para representar el caché y la memoria principal. Estas son: memoria principal, bloque y caché.

La estructura de memoria principal contendrá simplemente un arreglo de 64K caracteres de 1B. Esto nos permite no solo simular una memoria de tamaño 64KB sino también nos facilita el indexado, pues por ejemplo se puede acceder a la dirección 345 con: main_memory.data[345].

Mostramos la estructura que simula la memoria principal:

```
#define TAMANIO_MEMORIA_PRINCIPAL (64*1024)

typedef struct main_memory{
  unsigned char data[TAMANIO_MEMORIA_PRINCIPAL];
}main_memory_t;
```

Por otro lado, la estructura de memoria caché contiene los siguientes atributos:

- **was_hit**, indica si el último acceso fue hit o fue miss, indica si hay que buscar el dato pedido en memoria principal o si no es necesario, pues esta en caché
- **misses**, cuenta la cantidad de fallos de caché, permite calcular el miss rate
- **hits**, cuenta la cantidad de aciertos de caché, permite calcular el miss rate
- **cachesize**, indica el tamaño de la caché, es un atributo cuyo valor se puede establecer desde línea de comando
- **blocksize**, indica el tamaño de los bloques de caché, es un atributo cuyo valor se puede establecer desde línea de comando
- **ways**, indica la cantidad de vías de la cache, es un atributo cuyo valor se puede establecer desde linea de comando
- **sets**, indica la cantidad de conjuntos de la caché, queda determinado por los anteriores tres atributos
- **blocks**, son referencias a estructuras de bloques

Como se puede ver, la memoria caché tiene estructuras blocks, estas estructuras contienen a su vez los siguientes atributos:

- **lru**, indica que tan recientemente usado fue el bloque en comparación con los otros del conjunto, permite implementar la política de reemplazo homónima pues reemplazaremos el bloque del conjunto cuyo lru sea mayor.
- **dirty**, indica si el bloque de cache fue escrito o no, permite implementar la politica de escritura WB, pues se llevan los bloques dirty de cache a memoria principal si estos fueran a ser reemplazados.
- **valid**, indica si el bloque de cache contiene información valida
- **tag**, es el campo tag de la dirección del primer byte del bloque, en un acceso a memoria se compara el tag de la dirección a acceder con los tags de los bloques del conjunto al que cachea esa dirección
- **data**, contiene los datos que fueron cacheados desde la memoria principal

Mostramos las estructuras block y cache anteriormente mencionadas:

```
typedef struct block{
  unsigned int lru;
  bool dirty;
  bool valid;
  unsigned int tag;
  unsigned char* data;
}block_t;

typedef struct cache{
  bool was_hit;
  unsigned int misses;
  unsigned int hits;
  unsigned int cachesize;
  unsigned int blocksize;
  unsigned int ways;
  unsigned int sets;
  block_t** blocks;
}cache_t;
```

2.2. Cantidad de vías, tamaño de caché y tamaño de bloque

Dado que el espacio de direcciones es de 16 bits sabemos que las direcciones serán de 16 bits. Estas direcciones estarán dividas en tres campos: offset, index y tag.

El tamaño de bloque es el que nos determinará cuantos bits se utilizarán para el campo offset. Lo obtuvimos de la siguiente manera:

$$bitsOffset = log_2(tamañoBloque) \tag{1}$$

La cantidad de conjuntos es la que nos determinará cuantos bits se utilizarán para el campo index. Lo obtuvimos de la siguiente manera:

$$bitsIndex = log_2(cantidadConjuntos) \tag{2}$$

Donde:

$$cantidadConjuntos = \frac{tamañocache}{tamañoBloque * cantidadVías} \tag{3}$$

La cantidad de bits de tag será entonces: 16 - bitsOffset - bitsIndex.

Por ejemplo, supongamos queremos saber como se divide la dirección 2048 en una caché de 4KB, 4 vías y tamaño de bloque de 32 bytes. Entonces:

$$bitsOffset = log_2(tamañoBloque) = log_2(32) = 5 \tag{4}$$

$$bitsIndex = log_2(\frac{tamañocache}{tamañoBloque * cantidadVias}) = log_2((4K/32)/4) = 5 \tag{5}$$

$$bitsTag = 16 - bitsOffset - bitsIndex = 16 - 5 - 5 = 6 \tag{6}$$

La dirección resultante es:

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0

Y por lo tanto: Tag = 1 Index = 0 Offset = 0

Dado que para realizar estas operaciones necesitamos calcular logaritmos, incluimos la biblioteca $< math.h >$ y utilizamos la función log() junto con propiedades del logaritmo para obtener el resultado en potencias de 2. Esto se puede ver en las siguientes funciones, que dada una dirección llamada address, obtienen el tag, index y offset respectivamente. Para ello, realizan desplazamientos de forma tal de quedarnos con los bits correspondientes.

```
unsigned short getTag(unsigned int address){
  unsigned short offset_bits = ceil(log(cache.blocksize)/log(2));
  unsigned short index_bits = ceil(log(cache.sets)/log(2));
  unsigned short shift_size  = (offset_bits + index_bits);
  return address >> shift_size;
}

unsigned int find_set(unsigned int address){
  unsigned short offset_bits = ceil(log(cache.blocksize)/log(2));
  unsigned short index_bits = ceil(log(cache.sets)/log(2));
  unsigned short tag_bits = (BITS_DIRECCION - offset_bits - index_bits);
  unsigned short index = address << tag_bits;
  index = index >> (offset_bits + tag_bits) ;
  return index;
}

unsigned short getOffset(unsigned int address){
  unsigned short offset_bits = ceil(log(cache.blocksize)/log(2));
  unsigned short offset = address << (BITS_DIRECCION - offset_bits);
  offset = offset >> (BITS_DIRECCION - offset_bits);
  return offset;
}
```

2.3. Política de reemplazo LRU

La política de reemplazo LRU en una caché asociativa por conjuntos reemplaza el bloque más antiguamente utilizado del conjunto, en caso de un miss de caché.

Para implementar esto, hicimos que los bloques dentro de un conjunto tengan un número "lru", el cual representa hace cuanto se accedió al bloque en cuestión.

Cuando ocurre un hit de caché actualizamos los valores de lru: aquel bloque que fue accedido con un hit pasa a tener valor lru = 0 y el resto de los bloques aumentan en 1 su valor lru.

Cuando ocurre un miss de caché también actualizamos los valores de lru: desplazamos cualquier bloque inválido o, si no hay inválidos, el bloque de mayor valor lru, luego el bloque traído de

memoria principal a memoria caché tiene valor lru = 0 y aumentamos el valor lru del resto de los bloques en 1.

Mostramos un ejemplo de como se sucedería la secuencia de accesos a los bloques: A - B - A - D - C - E suponiendo que todos estos mapean al conjunto 0 de una caché asociativa por conjuntos de 4 vías:

	W0	W1	W2	W3
Set0	~V	~V	~V	~V
LRU	0	0	0	0

-> A	W0	W1	W2	W3
Set0	A	~V	~V	~V
LRU	0	1	1	1

-> B	W0	W1	W2	W3
Set0	A	B	~V	~V
LRU	1	0	2	2

-> A	W0	W1	W2	W3
Set0	A	B	~V	~V
LRU	0	1	3	3

-> D	W0	W1	W2	W3
Set0	A	B	D	~V
LRU	1	2	0	4

-> C	W0	W1	W2	W3
Set0	A	B	D	C
LRU	2	3	1	0

-> E	W0	W1	W2	W3
Set0	A	E	D	C
LRU	3	0	2	1

La actualización de los valores de lru la hemos realizado en la siguiente función:

```
void update_lru(unsigned int index,unsigned int way){
    for(int i = 0; i < cache.ways; i++){
        if(i!=way){
            cache.blocks[index][i].lru++;
        }
    }
    cache.blocks[index][way].lru = 0;
}
```

2.4. Política de escritura WB/WA

La política de escritura WB indica que en las escrituras escribiremos únicamente en caché e indicaremos que el bloque fue escrito marcándolo como dirty.

Para implementar esto le agregamos a la estructura de bloque de caché una variable booleana llamada dirty.

La política WA indica que en el caso de un miss de escritura debemos alocar el bloque pedido de memoria principal a caché. Sin embargo debemos tener el cuidado de que si estamos por reemplazar un bloque que esta dirty debemos mover antes este bloque a memoria principal.

Estas funcionalidades se pueden ver en las funciones que nos fueron pedidas:

```
void read_block(unsigned int blocknum){
  int first_address = blocknum*cache.blocksize;
  int set = find_set(first_address);
```

```
  int way = find_lru(set);

  if(is_dirty(way,set)){
    write_block(way,set);
  }

  for(int i = 0; i<cache.blocksize ;i++){
    cache.blocks[set][way].data[i] = main_memory.data[first_address + i];
  }

  cache.blocks[set][way].dirty = false;
  cache.blocks[set][way].valid = true;
  cache.blocks[set][way].lru = 0;
}


void write_block(unsigned int way,unsigned int setnum){
  if(!is_dirty(way,setnum)){
    return;
  }
  unsigned int address = getAddress(way,setnum);

  for(int i=0; i<cache.blocksize; i++){
    main_memory.data[address+i] = cache.blocks[setnum][way].data[i];
  }
}
```

2.5. Supuestos

Hemos supuesto que el tamaño de caché, los tamaños de los bloques y la cantidad de vías es potencia de 2. Esto nos permite no tener problemas al calcular log_2. También suponemos que las direcciones de memoria que nos dan están en el rango [0, 2^{16} - 1]. Para el caso de las potencias, realizamos la siguiente verificación, la cual consiste en restar 1 al numero. Si era potencia de 2, la operación dará 0. Esto es valido cuando el numero no es 0.

```
bool esPotenciaDeDos(unsigned int numero){
  return numero && (!(numero & (numero-1)));
}
```

Además llamamos a la función init() antes de comenzar los accesos a memoria a pesar de que no se indique en el archivo .mem. Esto es para asegurarnos que los bloques de la caché están inválidos, la memoria simulada en 0 y la tasa de misses también en 0.

3. Portabilidad

El programa fue hecho en su totalidad en el lenguaje C. El mismo fue probado en sistemas operativos Ubuntu, por lo que no debería de haber problemas para portabilidad con respecto a sistemas operativos de tipo UNIX.

4. Casos de prueba

Para la evaluación del correcto funcionamiento del programa hemos creado un archivo bash que contiene pruebas. Junto a este, vienen también mas archivos con extensión .mem que contienen

las secuencias de lecturas y de escrituras que se realizarán sobre la memoria. El mismo archivo se puede observar en el apéndice.

Los archivos de prueba que creamos son (los primeros dos fueron otorgados):

- pruebas1.mem ->Se escribe en las direcciones 0, 16384, 32768, 49152, luego se lee en esas direcciones en el mismo orden.
- pruebas2.mem ->Se escribe en las direcciones 0, 1024, 2048, 3072, 4096 luego se lee en esas direcciones en el mismo orden.
- pruebas3.mem ->Se leen consecutivamente las direcciones 0 - 15
- pruebas4.mem ->Se escribe en las direcciones 28 - 35, se leen las direcciones 28 - 35
- pruebas5.mem ->Se escribe en las direcciones 0, 55, 100, 400, luego se leen las direcciones 0, 55, 100, 400.

Cada uno de los archivos se ejecutará en las siguientes configuraciones:

- Configuración 1: 4 KB de tamaño, 4 vías, bloques de 32 bytes
- Configuración 2: 16 KB de tamaño, 1 vıa, bloques de 128 bytes

Explicamos brevemente las pruebas hechas. Las pruebas 1 y 2 se explican en detalle en la siguiente sección.

- En la prueba 3 para la configuración 1 tendremos un miss que nos trae el bloque de 32 bytes a alguna de las 4 vías. Luego accedemos a direcciones 1-15 que ya se encuentran en este bloque traído. Por lo tanto el miss rate que esperamos ver es: $1/16 = 6\,\%$.
- En la prueba 3 para la configuración 2 ocurre algo similar. Ocurre un primer miss debido a que la caché esta vacía y nos trae el bloque de 128B al conjunto 0 (que solo tiene capacidad de 1 bloque). Los siguientes accesos 1-15 serán luego hits: miss rate $1/16 = 6\,\%$
- En la prueba 4 para la configuración 1 tendremos 1 miss que trae un bloque con las direcciones 0-31 por lo que los siguientes 3 accesos seran hit (acceden del 29-31). Al acceder al byte 32 tenemos un miss, traemos el siguiente bloque (de direcciones 32-63) y por lo tanto los siguientes accesos serán hits, entonces miss rate $= 2/16 = 12\,\%$
- En la prueba 4 para la configuración 2 tendremos 1 miss que trae un bloque con las direcciones 0-128 y por lo tanto todos los siguientes accesos seran hits, entonces miss rate $= 1/16 = 6\,\%$
- En la prueba 5 para la configuración 1 los bloques cachean a conjuntos distintos entonces tendremos 4 misses de las escrituras seguidos de 4 hits de las escrituras, entonces el miss rate sera del $50\,\%$.
- En la prueba 5 para la configuración 2 el primer acceso trae un bloque de tamaño 128 a la caché (1 miss). Luego las siguientes 2 escrituras a los byets 55 y 100 se realizan sobre este bloque de caché (2 hits). La escritura sobre el byte 400 producirá un miss y las siguientes 4 lecturas deberán ser hits. Por lo tanto el miss rate es: $(1+1/8) = 25\,\%$

Por último creamos una prueba, llamada prueba6.mem en la que comparamos dos nuevas configuraciones:

- Configuración 3: tamaño cache 4KB, 1 vía, tamaño bloque 16B
- Configuración 4: tamaño cache 4KB, 4 vías, tamaño bloque 16B

El objetivo de esta prueba es comparar la influencia de la cantidad de vías en accesos a posiciones de memoria que cachean al mismo conjunto. Con este objetivo escribimos en las posiciones de memoria: 0, 4096, 8192 (en cada configuración, estas direcciones tienen el mismo offset e index) y luego leemos en ellas en ese orden.

Esperamos ver los siguientes resultados:

- Para la configuración 3 la caché es de mapeo directo, por lo que si todos los bloques cachean al mismo conjunto, cada nuevo acceso desplazará al bloque anterior: MR = 100 %
- Para la configuración 4 la caché tiene 4 vías y se suceden 3 escrituras inicialmente que cachean al mismo conjunto, c/u será un miss pero ocupará una vía distinta y por lo tanto las siguientes 3 lecturas serán hit: MR = 50 %

Para correr estas pruebas ejecutar:

```
bash pruebas.sh
```

También realizamos un archivo de pruebas tests.c, el cual se puede compilar con *make test* y ejecutar con ./tests. El objetivo de este archivo es poder evaluar de mejor forma y en un ambiente mas controlado las diversas funciones planteadas por el enunciado, ya que con el archivo bash se evalúa el programa de forma general. El archivo mencionado se puede ver también en el apéndice.

5. Presentación de resultados obtenidos

En esta sección procederemos a fundamentar los resultados obtenidos en los archivos de prueba: prueba1.mem y prueba2.mem para las siguientes configuraciones de caché:

- Configuración 1: 4 KB de tamaño, 4 vías, bloques de 32 bytes
- Configuración 2: 16 KB de tamaño, 1 vıa, bloques de 128 bytes

Los resultados de las pruebas 1 y 2 en la configuración 1 fueron las siguientes:

```
$ ./tp -c 4 -w 4 -b 32 prueba1.mem
Se escribe el caracter 255 en 0
* El resultado de la operacion fue: MISS
Se escribe el caracter 254 en 16384
* El resultado de la operacion fue: MISS
Se escribe el caracter 248 en 32768
* El resultado de la operacion fue: MISS
Se escribe el caracter 96 en 49152
* El resultado de la operacion fue: MISS
Se lee el byte en la direccion: 0
* El valor obtenido fue:  (255)
* El resultado de la operacion fue: HIT
Se lee el byte en la direccion: 16384
* El valor obtenido fue:  (254)
* El resultado de la operacion fue: HIT
Se lee el byte en la direccion: 32768
* El valor obtenido fue:  (248)
* El resultado de la operacion fue: HIT
Se lee el byte en la direccion: 49152
* El valor obtenido fue: ` (96)
* El resultado de la operacion fue: HIT
El miss rate es de: 50 %
```

```
$ ./tp -c 4 -w 4 -b 32 prueba2.mem
Se escribe el caracter 123 en 0
* El resultado de la operacion fue: MISS
Se escribe el caracter 234 en 1024
* El resultado de la operacion fue: MISS
Se escribe el caracter 33 en 2048
* El resultado de la operacion fue: MISS
Se escribe el caracter 44 en 3072
* El resultado de la operacion fue: MISS
Se escribe el caracter 55 en 4096
* El resultado de la operacion fue: MISS
Se lee el byte en la direccion: 0
* El valor obtenido fue: { (123)
* El resultado de la operacion fue: MISS
Se lee el byte en la direccion: 1024
* El valor obtenido fue:  (234)
* El resultado de la operacion fue: MISS
Se lee el byte en la direccion: 2048
* El valor obtenido fue: ! (33)
* El resultado de la operacion fue: MISS
Se lee el byte en la direccion: 3072
* El valor obtenido fue: , (44)
* El resultado de la operacion fue: MISS
Se lee el byte en la direccion: 4096
* El valor obtenido fue: 7 (55)
* El resultado de la operacion fue: MISS
El miss rate es de: 100 %
```

Para explicar estas tasas de miss debemos ver cual es la división en campos de las direcciones accedidas.

Las direcciones accedidas se dividen en los siguientes campos para la configuración 1:

DIRECCIONES ACCEDIDAS PRUEBA 1			
ADDRESS	TAG	INDEX	OFFSET
0	000000	00000	00000
16384	010000	00000	00000
32768	100000	00000	00000
49152	110000	00000	00000
DIRECCIONES ACCEDIDAS PRUEBA 2			
ADDRESS	TAG	INDEX	OFFSET
0	000000	00000	00000
1024	000001	00000	00000
2048	000010	00000	00000
3072	000011	00000	00000
4096	000100	00000	00000

Observamos que todos los bloques accedidos tanto en la prueba 1 como en la prueba 2 cachean al conjunto 0. Por lo tanto nos concentraremos solo en este conjunto.

Mostramos ahora como se suceden los accesos para la prueba 1 en la configuración 1, mostramos que los primeros 4 accesos son miss de caché y completan el conjunto 0 y luego las lecturas encuentran todo cacheado, por lo tanto el miss rate debe ser de 50 %.

Configuracion 1 - Prueba 1					
	WRITES	W0	W1	W2	W3
Set 0	Tag	0	16	32	48
	Value	255	254	248	96

	R 0	W0	W1	W2	W3
Set 0	Tag	0	16	32	48
	Value	255	254	248	96

	R 16384	W0	W1	W2	W3
Set 0	Tag	0	16	32	48
	Value	255	254	248	96

	R 32768	W0	W1	W2	W3
Set 0	Tag	0	16	32	48
	Value	255	254	248	96

	R 49152	W0	W1	W2	W3
Set 0	Tag	0	16	32	48
	Value	255	254	248	96

Estudiando ahora como se suceden los accesos para la prueba 2 en la configuración 1 vemos que hacemos 5 writes en bloques que cachean al conjunto 0, entonces los primeros 4 son miss, se llena el conjunto y el quinto write también es miss porque no se escribe sobre alguno de los bloques ya cacheados y se reemplaza el primer bloque cacheado. Luego se suceden 5 reads, el primero intenta leer el bloque que fue recientemente desplazado, por lo que ocurre un miss y desplaza un bloque. El segundo read intenta leer el bloque que fue ahora desplazado por el bloque que se introdujo en el read anterior. Esto se produce para los siguientes reads también y por lo tanto el miss rate es del 100 %.

Configuracion 1 - Prueba 2					
	4 WRITES	W0	W1	W2	W3
Set 0	Tag	0	1	2	3
	Value	123	234	33	44

	W 4096, 55	W0	W1	W2	W3
Set 0	Tag	4	1	2	3
	Value	55	234	33	44

	R 0	W0	W1	W2	W3
Set 0	Tag	4	0	2	3
	Value	55	123	33	44

	R 1024	W0	W1	W2	W3
Set 0	Tag	4	0	1	3
	Value	55	123	234	44

	R 2048	W0	W1	W2	W3
Set 0	Tag	4	0	1	2
	Value	55	123	234	33

	R 3072	W0	W1	W2	W3
Set 0	Tag	3	0	1	2
	Value	44	123	234	33

	R 4096	W0	W1	W2	W3
Set 0	Tag	3	4	1	2
	Value	44	55	234	33

Los resultados de las pruebas 1 y 2 para la configuración 2 fueron los siguientes:

```
$ ./tp -c 16 -w 1 -b 128 prueba1.mem
Se escribe el caracter 255 en 0
* El resultado de la operacion fue: MISS
Se escribe el caracter 254 en 16384
* El resultado de la operacion fue: MISS
Se escribe el caracter 248 en 32768
* El resultado de la operacion fue: MISS
Se escribe el caracter 96 en 49152
* El resultado de la operacion fue: MISS
Se lee el byte en la direccion: 0
* El valor obtenido fue:  (255)
* El resultado de la operacion fue: MISS
Se lee el byte en la direccion: 16384
* El valor obtenido fue:  (254)
* El resultado de la operacion fue: MISS
Se lee el byte en la direccion: 32768
* El valor obtenido fue:  (248)
* El resultado de la operacion fue: MISS
Se lee el byte en la direccion: 49152
* El valor obtenido fue: ` (96)
* El resultado de la operacion fue: MISS
El miss rate es de: 100 %
```

```
$ ./tp -c 16 -w 1 -b 128 prueba2.mem
Se escribe el caracter 123 en 0
* El resultado de la operacion fue: MISS
Se escribe el caracter 234 en 1024
* El resultado de la operacion fue: MISS
Se escribe el caracter 33 en 2048
* El resultado de la operacion fue: MISS
Se escribe el caracter 44 en 3072
* El resultado de la operacion fue: MISS
Se escribe el caracter 55 en 4096
* El resultado de la operacion fue: MISS
Se lee el byte en la direccion: 0
* El valor obtenido fue: { (123)
* El resultado de la operacion fue: HIT
Se lee el byte en la direccion: 1024
* El valor obtenido fue:  (234)
* El resultado de la operacion fue: HIT
Se lee el byte en la direccion: 2048
* El valor obtenido fue: ! (33)
* El resultado de la operacion fue: HIT
Se lee el byte en la direccion: 3072
* El valor obtenido fue: , (44)
* El resultado de la operacion fue: HIT
Se lee el byte en la direccion: 4096
* El valor obtenido fue: 7 (55)
* El resultado de la operacion fue: HIT
El miss rate es de: 50 %
```

Para la configuración 2 la división de las direcciones es la siguiente:

DIRECCIONES ACCEDIDAS PRUEBA 1			
ADDRESS	TAG	INDEX	OFFSET
0	00	0000000	0000000
16384	01	0000000	0000000
32768	10	0000000	0000000
49152	11	0000000	0000000
DIRECCIONES ACCEDIDAS PRUEBA 2			
ADDRESS	TAG	INDEX	OFFSET
0	00	0000000	0000000
1024	00	0001000	0000000
2048	00	0010000	0000000
3072	00	0011000	0000000
4096	00	0100000	0000000

En la primer prueba accedemos únicamente al set 0 que en este caso está formado por 1 solo bloque (es de mapeo directo). Es por eso que todos los writes son misses (sobreescriben lo que ya hay en el bloque del set 0) y todos los reads también son misses (por el mismo motivo). Por lo tanto el miss rate debe ser 100 %

Configuracion2 - Prueba1		
Set 0	W 0,255	W0
	Tag	0
	Value	255
Set 0	W 16384	W0
	Tag	1
	Value	254
Set 0	W 32768, 248	W0
	Tag	2
	Value	248
Set 0	W 49152, 096	W0
	Tag	3
	Value	96
Set 0	R 0	W0
	Tag	0
	Value	255
Set 0	R 16384	W0
	Tag	1
	Value	254
Set 0	R 32768	W0
	Tag	2
	Value	248
Set 0	R 49152	W0
	Tag	3
	Value	96

En la segunda prueba accedemos siempre a conjuntos distintos en las escrituras, estos serán miss porque la caché esta inicialmente vacia. Luego, sin embargo habrá todos hits en los reads. Esto se debe a que los conjuntos que se cachearon durante los writes no se sobreescribieron. El miss rate es entonces: 50 %

Configuracion 2 - Prueba 2		
	W 0, 123	W0
Set 0	Tag	0
	Value	123
	W 1024, 234	W0
Set 4	Tag	0
	Value	234
	W 2048, 33	W0
Set 8	Tag	0
	Value	33
	W 3072, 44	W0
Set 12	Tag	0
	Value	44
	W 4096, 55	W0
Set 16	Tag	0
	Value	55

	R 0	W0
Set 0	Tag	0
	Value	123
	R 1024	W0
Set 4	Tag	0
	Value	234
	R 2048	W0
Set 8	Tag	0
	Value	33
	R 3072	W0
Set 12	Tag	0
	Value	44
	R 4096	W0
Set 16	Tag	0
	Value	55

6. Conclusiones

Al realizar el trabajo practico hemos llegado a las siguientes conclusiones.

- Hemos podido comprender mejor el funcionamiento de las caché, ya que durante el desarrollo se pudo observar la diferencia que ocurre al realizar los accesos (en cuanto a código). Siendo que el camino si se esta en cache es mucho mas directo en el flujo del programa.
- Al tener que implementar una política de reemplazo LRU, pudimos terminar de comprender como funciona y los desafíos/consecuencias que conlleva.
- Pudimos observar que las diferentes implementaciones de caché van cambiando su miss rate según la prueba que se hace, por lo que no podemos afirmar que alguna configuración sea mejor que otra, sino que depende de cada caso particular de prueba que se ejecute. Por ejemplo, si se suceden accesos a direcciones de memoria cuyos bloques cachean todos al mismo conjunto es probable que el miss rate sea menor en una cache de mayor cantidad de vías, pues cada conjunto tendrá mayor capacidad de bloques. Por otro lado en una cache de mapeo directo (1 vía) una secuencia de accesos de este tipo puede llevar a reemplazar siempre el bloque anterior y tener un miss rate elevado.

7. Referencias

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

https://github.com/brunograssano/Organizacion-de-computadoras-fiuba/releases/tag/v1.2.0

8. Apéndices

8.1. main.c

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <getopt.h>
#include <math.h>
#include "cache.h"

#define OUTPUT 'o'
#define VERSION 'v'
#define AYUDA 'h'
#define CANTIDAD_VIAS 'w'
#define CACHE_SIZE 'c'
#define BLOCK_SIZE 'b'

#define READ_BYTE 'R'
#define WRITE_BYTE 'W'

#define MAX_NOMBRE_ARCHIVO 256

#define MODO_LECTURA "r"
#define MODO_ESCRITURA "w"

#define KILOBYTE 1024
const int ERROR = -1, VACIO=0, TERMINO = -1;

void mostrarAyuda(){
  printf("Uso: \n");
  printf(" tp -h\n");
  printf(" tp -v\n");
  printf(" tp [opciones] archivo\n");
  printf("Opciones: \n");
  printf(" -v, --version    Imprime la version y termina el programa.\n");
  printf(" -h, --help       Imprime esta informacion.\n");
  printf(" -o, --output     Indica que le sigue la direccion al archivo de salida.\n");
  printf(" -w, --ways       Cantidad de vias.\n");
  printf(" -c, --cachesize  Tamanio del cache en kilobytes\n");
  printf(" -b, --blocksize  Tamanio del bloque en bytes\n");
  printf("Ejemplos: \n");
  printf(" tp -w 4 -c 8 -b 16 prueba1.mem \n");
  printf(" tp -w 1 -c 4 -b 128 prueba1.mem \n");
  printf(" tp -w 1 -c 4 -b 32 -o salida prueba1.mem \n");
}
```

```
void mostrarVersion(){
         printf("Version 1.0.0\n");
}

void determinarSalida(configuracion_t* configuracion,char* argumentos[]){
  configuracion->salida = fopen(optarg,MODO_ESCRITURA);
  if(configuracion->salida == NULL){
    fprintf(stderr, "No se pudo abrir el archivo enviado para salida, el resultado
    se mostrara por stdout\n");
    configuracion->salida = stdout;
  }
}

bool esPotenciaDeDos(unsigned int numero){
  return numero && (!(numero & (numero-1)));
}

configuracion_t manejarParametros(int cantidadArgumentos, char* argumentos[],
                                  char archivoInput[MAX_NOMBRE_ARCHIVO]){
  static struct option opcionesLargas[] = {
     {"ways", required_argument, 0, 'w'},
     {"cachesize", required_argument, 0, 'c'},
     {"help", no_argument, 0, 'h'},
     {"output", required_argument, 0, 'o'},
     {"version", no_argument, 0, 'v'},
     {"blocksize", required_argument, 0, 'b'},
     {0, 0, 0, 0}
  };
  configuracion_t configuracion;
  memset(&configuracion,0,sizeof(configuracion_t));
  int argumento, indiceOpcion = 0;
  bool pidioAyuda = false, pidioVersion = false;

  while((argumento = getopt_long(cantidadArgumentos, argumentos, "w:c:ho:vb:",
        opcionesLargas, &indiceOpcion))!=TERMINO){
      switch (argumento) {
          case CANTIDAD_VIAS:
              configuracion.vias = atoi(optarg);
              break;
          case CACHE_SIZE:
              configuracion.tamanioCache = atoi(optarg);
              break;
          case BLOCK_SIZE:
              configuracion.tamanioBloque = atoi(optarg);
              break;
          case OUTPUT:
              determinarSalida(&configuracion,argumentos);
              break;
          case VERSION:
              if(!pidioVersion){
                  configuracion.pidioOtraOpcion = true;
                  pidioVersion = true;
                  mostrarVersion();
```

```
                }
                break;
            case AYUDA:
                if(!pidioAyuda){
                    configuracion.pidioOtraOpcion = true;
                    pidioAyuda = true;
                    mostrarAyuda();
                }
                break;
            default:
                fprintf(stderr, "Puede ver ayuda enviando el parametro -h \n");
        }
    }

    if(optind<cantidadArgumentos){
      strcpy(archivoInput,argumentos[optind]);
    }

    if(configuracion.salida == NULL){
      configuracion.salida = stdout;
    }
    return configuracion;
}

bool excedeRango(int vias,int tamanioCache,int tamanioBloque){
    unsigned short bitsOffset = ceil(log(tamanioBloque)/log(2));
    unsigned short bitsIndex =
    ceil(log((tamanioCache * KILOBYTE)/(vias*tamanioBloque))/log(2));
    unsigned short bitsTag = (BITS_DIRECCION - bitsOffset - bitsIndex);
    return (bitsOffset + bitsIndex + bitsTag)>BITS_DIRECCION || bitsIndex == 0;
}

int parsearArchivo(FILE* fileInput,FILE* fileOutput){
    const char delimitadorEspacio[2] = " ",delimitadorComa[3] = ", ";
    char buffer[MAX_NOMBRE_ARCHIVO] = "";
    unsigned int direccionALeer = 0,caracter = 0;
    int estado = 0;

    init();
    int leidos = fscanf(fileInput,"%[^\n]\r\n",buffer);
    while(0<leidos && estado!=ERROR){
      if(strcmp(buffer,"init")==0 || strcmp(buffer,"init\r")==0){
        init();
        fprintf(fileOutput, "Se inicializa la cache\n");
      }
      else if(buffer[0] == READ_BYTE || buffer[0] == WRITE_BYTE){
        char* instruccion = strtok(buffer,delimitadorEspacio);
        char* primerNumero = strtok(NULL,delimitadorComa);
        direccionALeer = atoi(primerNumero);
        if(direccionALeer >= TAMANIO_MEMORIA_PRINCIPAL){
          fprintf(stderr, "La direccion %i se excede del rango de
                   la memoria\n",direccionALeer);
          estado = ERROR;
        }
```

```
      if(buffer[0] == READ_BYTE){
        unsigned char valor = read_byte(direccionALeer);
        fprintf(fileOutput, "Se lee el byte en la direccion: %i\n",direccionALeer);
        fprintf(fileOutput, "* El valor obtenido fue: %c (%d)\n", valor,valor);
      }
      else{
        char* valor = strtok(NULL,delimitadorComa);
        caracter = atoi(valor);
        write_byte(direccionALeer,caracter);
        fprintf(fileOutput, "Se escribe el caracter %d en %i\n",
                caracter,direccionALeer);
      }
      fprintf(fileOutput, "* El resultado de la operacion fue: %s\n",
              was_hit()?"HIT":"MISS");
    }
    else if(strcmp(buffer,"MR")==0 || strcmp(buffer,"MR\r")==0){
      int missRate = get_miss_rate();
      fprintf(fileOutput, "El miss rate es de: %i %% \n", missRate);
    }
    strcpy(buffer,"");
    leidos = fscanf(fileInput,"%[^\n]\n",buffer);
  }
  return estado;
}


int main(int cantidadArgumentos, char* argumentos[]){
  char archivoInput[MAX_NOMBRE_ARCHIVO] = "";
  int estado = 0;

  if(cantidadArgumentos == 1){
    fprintf(stderr, "No se enviaron argumentos. Puede ver ayuda mandando -h\n");
    return ERROR;
  }

  configuracion_t configuracion =
                  manejarParametros(cantidadArgumentos,argumentos,archivoInput);
  if(configuracion.pidioOtraOpcion){
    return estado;
  }

  if(strlen(archivoInput)==VACIO){
    fprintf(stderr, "No se envio el archivo de entrada, se termina el programa.
    Puede ver ayuda mandando -h\n");
    return ERROR;
  }

  if(!(esPotenciaDeDos(configuracion.vias) &&
       esPotenciaDeDos(configuracion.tamanioCache) &&
       esPotenciaDeDos(configuracion.tamanioBloque))){
    fprintf(stderr, "No se pueden enviar por entrada valores que
            no sean potencias de 2.\n");
    return ERROR;
```

```
  }

  if(excedeRango(configuracion.vias, configuracion.tamanioCache,
                 configuracion.tamanioBloque)){
    fprintf(stderr, "Los parametros enviados exceden la representacion
            que se tiene en la cache (16 bits)\n");
    return ERROR;
  }

  FILE* fileInput = fopen(archivoInput,MODO_LECTURA);
  if(fileInput==NULL){
      fprintf(stderr, "No se pudo abrir el archivo de entrada.\n");
      if(configuracion.salida!=stdout){
        fclose(configuracion.salida);
      }
      return ERROR;
  }

  estado = set_up_cache(configuracion);
  if(estado == ERROR){
    fclose(fileInput);
    if(configuracion.salida!=stdout){
      fclose(configuracion.salida);
    }
    return estado;
  }

  estado = parsearArchivo(fileInput,configuracion.salida);

  destroy_cache();

  fclose(fileInput);
  if(configuracion.salida!=stdout){
    fclose(configuracion.salida);
  }

  return estado;
}
```

8.2. cache.h

```
#ifndef __CACHE_H__
#define __CACHE_H__

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define TAMANIO_MEMORIA_PRINCIPAL (64*1024)
#define BITS_DIRECCION 16

typedef struct configuracion{
  bool pidioOtraOpcion;
  int vias;
  int tamanioCache;
  int tamanioBloque;
  FILE* salida;
}configuracion_t;

int set_up_cache(configuracion_t configuracion);

void destroy_cache();

bool was_hit();

void init();

unsigned int find_set(unsigned int address);

unsigned int find_lru(unsigned int setnum);

unsigned int is_dirty(unsigned int way,unsigned int setnum);

void read_block(unsigned int blocknum);

void write_block(unsigned int way,unsigned int setnum);

unsigned char read_byte(unsigned int address);

void write_byte(unsigned int address,unsigned char value);

int get_miss_rate();

#endif /* __CACHE_H__ */
```

8.3. cache.c

```
#include "cache.h"
#include <string.h>
#include <math.h>
#define KILOBYTE 1024
#define ERROR -1

typedef struct main_memory{
  unsigned char data[TAMANIO_MEMORIA_PRINCIPAL];
}main_memory_t;

typedef struct block{
  unsigned int lru;
  bool dirty;
  bool valid;
  unsigned int tag;
  unsigned char* data;
}block_t;

typedef struct cache{
  bool was_hit;
  unsigned int misses;
  unsigned int hits;
  unsigned int cachesize;
  unsigned int blocksize;
  unsigned int ways;
  unsigned int sets;
  block_t** blocks;
}cache_t;

cache_t cache;
main_memory_t main_memory;
/////------------------------------AUX------------------------------/////

void update_lru(unsigned int index,unsigned int way){
  for(int i = 0; i < cache.ways; i++){
    if(i!=way){
        cache.blocks[index][i].lru++;
    }
  }
  cache.blocks[index][way].lru = 0;
}

unsigned short getTag(unsigned int address){
  unsigned short offset_bits = ceil(log(cache.blocksize)/log(2));
  unsigned short index_bits = ceil(log(cache.sets)/log(2));
  unsigned short shift_size  = (offset_bits + index_bits);
  return address >> shift_size;
}

unsigned short getOffset(unsigned int address){
  unsigned short offset_bits = ceil(log(cache.blocksize)/log(2));
  unsigned short offset = address << (BITS_DIRECCION - offset_bits);
```

```
  offset = offset >> (BITS_DIRECCION - offset_bits);
  return offset;
}

unsigned short getAddress(unsigned int way,unsigned int setnum){
  unsigned short index_bits = ceil(log(cache.sets)/log(2));
  unsigned short shifted_tag = cache.blocks[setnum][way].tag << index_bits;
  return (shifted_tag | setnum) * cache.blocksize;
}

int search_in_cache(unsigned int index,unsigned int tag){
  int way = 0;
  cache.was_hit = false;
  while(way < cache.ways && !cache.was_hit){
    if(cache.blocks[index][way].tag == tag && cache.blocks[index][way].valid){
      cache.was_hit = true;
      cache.hits++;
    }
    else{
      way++;
    }
  }
  return way;
}

int set_up_cache(configuracion_t configuracion){
  cache.cachesize = configuracion.tamanioCache * KILOBYTE;
  cache.blocksize = configuracion.tamanioBloque;
  cache.ways = configuracion.vias;
  cache.sets = cache.cachesize/(cache.ways*cache.blocksize);

  cache.blocks = calloc(cache.sets,sizeof(block_t*));
  if(cache.blocks == NULL){
    fprintf(stderr, "Ocurrio un error al alocar la memoria.\n");
    return ERROR;
  }

  for(int i=0;i<cache.sets;i++){
    cache.blocks[i] = calloc(cache.ways,sizeof(block_t));
    if(cache.blocks[i] == NULL){
      fprintf(stderr, "Ocurrio un error al alocar la memoria.\n");
      return ERROR;
    }
  }

  for(int i=0;i<cache.sets;i++){
    for(int j=0;j<cache.ways;j++){
      cache.blocks[i][j].data = calloc(cache.blocksize,sizeof(char));
      if(cache.blocks[i][j].data == NULL){
        fprintf(stderr, "Ocurrio un error al alocar la memoria.\n");
        return ERROR;
      }
    }
  }
```

```
  return 0;
}

void destroy_cache(){
  for(int i=0;i<cache.sets;i++){
    for(int j=0;j<cache.ways;j++){
      free(cache.blocks[i][j].data);
    }
  }

  for(int i=0;i<cache.sets;i++){
    free(cache.blocks[i]);
  }
  free(cache.blocks);
}

bool was_hit(){
  return cache.was_hit;
}

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


void init(){
  cache.misses = 0;
  cache.hits = 0;
  memset(&main_memory,0,sizeof(main_memory));

  for(int i=0;i<cache.sets;i++){
    for(int j=0;j<cache.ways;j++){
      cache.blocks[i][j].valid = false;
    }
  }
}


/* La función find set(int address) debe devolver el conjunto de caché al que
mapea la dirección address.
 */
unsigned int find_set(unsigned int address){
  unsigned short offset_bits = ceil(log(cache.blocksize)/log(2));
  unsigned short index_bits = ceil(log(cache.sets)/log(2));
  unsigned short tag_bits = (BITS_DIRECCION - offset_bits - index_bits);
  unsigned short index = address << tag_bits;
  index = index >> (offset_bits + tag_bits) ;
  return index;
}


/* 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.
 */
unsigned int find_lru(unsigned int setnum){
```

```
  int highest_lru = 0, i = 0;
  unsigned int lru_block = 0;
  bool found = false;
  while(i<cache.ways && !found){
      if(!cache.blocks[setnum][i].valid){
        lru_block = i;
        found = true;
      }else if(cache.blocks[setnum][i].lru>highest_lru && cache.blocks[setnum][i].valid){
        highest_lru = cache.blocks[setnum][i].lru;
        lru_block = i;
      }
      i++;
  }
  return lru_block;
}


/* La función is dirty(int way, int setnum) debe devolver el estado del bit D
 * del bloque correspondiente.
 */
unsigned int is_dirty(unsigned int way,unsigned int setnum){
  return (unsigned int)cache.blocks[setnum][way].dirty;
}


/* 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.
 */
void read_block(unsigned int blocknum){
  int first_address = blocknum*cache.blocksize;
  int set = find_set(first_address);
  int way = find_lru(set);
  if(is_dirty(way,set)){
    write_block(way,set);
  }
  for(int i = 0; i<cache.blocksize; i++){
    cache.blocks[set][way].data[i] = main_memory.data[first_address + i];
  }
  cache.blocks[set][way].dirty = false;
  cache.blocks[set][way].valid = true;
  cache.blocks[set][way].lru = 0;
}


/* La función write block(int way, int setnum) debe escribir en memoria
 * los datos contenidos en el bloque setnum de la vıa way.
 */
void write_block(unsigned int way,unsigned int setnum){
  if(!is_dirty(way,setnum)){
    return;
  }
  unsigned int address = getAddress(way,setnum);
```

```
  for(int i=0; i<cache.blocksize; i++){
    main_memory.data[address+i] = cache.blocks[setnum][way].data[i];
  }
}


/* La función read byte(address) debe retornar el valor correspondiente
 * a la posición de memoria address, buscándolo primero en el caché.
 */
unsigned char read_byte(unsigned int address){
  unsigned short tag = getTag(address);
  unsigned short index = find_set(address);
  unsigned short offset = getOffset(address);
  unsigned int way = search_in_cache(index,tag);

  if(cache.was_hit){
    update_lru(index,way);
    return cache.blocks[index][way].data[offset];
  }

  cache.misses++;
  way = find_lru(index);
  read_block(address/cache.blocksize);
  cache.blocks[index][way].tag = tag;
  update_lru(index,way);
  return cache.blocks[index][way].data[offset];
}


/* La función write byte(int address, char value) debe escribir el
 * valor value en la posición correcta del bloque que corresponde a
 * address
 */
void write_byte(unsigned int address,unsigned char value){
  unsigned int tag = getTag(address);
  unsigned int index = find_set(address);
  unsigned int offset = getOffset(address);
  unsigned int way = search_in_cache(index,tag);

  if(cache.was_hit){
    update_lru(index,way);
    cache.blocks[index][way].dirty = true;
    cache.blocks[index][way].data[offset] = value;
    return;
  }

  cache.misses++;
  way = find_lru(index);
  read_block(address/cache.blocksize);
  cache.blocks[index][way].tag = tag;
  update_lru(index,way);
  cache.blocks[index][way].dirty = true;
  cache.blocks[index][way].data[offset] = value;
}
```

```c
/* La función get miss rate() debe devolver el porcentaje de misses desde que
 * se inicializó el cache.
 */
int get_miss_rate(){
  int total = cache.misses+cache.hits;
  if(total == 0){
    return 0;
  }
  return (cache.misses*100)/(total);
}
```

8.4. Pruebas

8.4.1. pruebas.sh

```
espacios() {
    echo
    echo
}
newline=$'\n'

echo Comienza la ejecucion de las pruebas

espacios

        echo -e "\e[1m PRUEBA 1 - Configuracion 1 \e[0m"
        echo

        ./tp -c 4 -w 4 -b 32 prueba1.mem

        echo -e "\e[1m PRUEBA 1 - Configuracion 2 \e[0m"
        echo

        ./tp -c 16 -w 1 -b 128 prueba1.mem

            echo -e "\e[1m PRUEBA 2 - Configuracion 1 \e[0m"
        echo

        ./tp -c 4 -w 4 -b 32 prueba2.mem


        echo -e "\e[1m PRUEBA 2 - Configuracion 2 \e[0m"
        echo

        ./tp -c 16 -w 1 -b 128 prueba2.mem

            echo -e "\e[1m PRUEBA 3 - Configuracion 1 \e[0m"
        echo

        ./tp -c 4 -w 4 -b 32 prueba3.mem


        echo -e "\e[1m PRUEBA 3 - Configuracion 2 \e[0m"
        echo

        ./tp -c 16 -w 1 -b 128 prueba3.mem
        echo

            echo -e "\e[1m PRUEBA 4 - Configuracion 1 \e[0m"
        echo

        ./tp -c 4 -w 4 -b 32 prueba4.mem

        echo -e "\e[1m PRUEBA 4 - Configuracion 2 \e[0m"
        echo
```

```
./tp -c 16 -w 1 -b 128 prueba4.mem

   echo -e "\e[1m PRUEBA 5 - Configuracion 1 \e[0m"
echo

./tp -c 4 -w 4 -b 32 prueba5.mem

echo -e "\e[1m PRUEBA 5 - Configuracion 2 \e[0m"
echo

./tp -c 16 -w 1 -b 128 prueba5.mem

echo
echo -e "\e[1m PRUEBA 6 - Configuracion 3 \e[0m"

echo

./tp -c 4 -w 1 -b 16 prueba6.mem

echo
echo -e "\e[1m PRUEBA 6 - Configuracion 4 \e[0m"

echo

./tp -c 4 -w 4 -b 16 prueba6.mem
```

8.4.2. tests.c

```
#include "cache.h"
#include "mi_assert.h"
#define ERROR -1

int sePideCacheSinAccesosDevuelveCero(int* contadorTotales){
  int contadorSuperadas = 0;
  configuracion_t configuracion = {false,2,4,32,NULL};
  if(set_up_cache(configuracion)==ERROR){
    return contadorSuperadas;
  }
  init();

  assert(get_miss_rate()==0,"El miss rate inicial es de 0",contadorSuperadas);
  (*contadorTotales)++;
  destroy_cache();
  return contadorSuperadas;
}

int alPedirUbicacionesDelCacheSinUsoTodasEstanLimpias(int* contadorTotales){
  int contadorSuperadas = 0;
  configuracion_t configuracion = {false,4,4,32,NULL};
  if(set_up_cache(configuracion)==ERROR){
    return contadorSuperadas;
  }
  init();

  assert(!is_dirty(3,2),"Esta limpio el bloque del conjunto 2 y via 3",contadorSuperadas);
  (*contadorTotales)++;
  assert(!is_dirty(1,4),"Esta limpio el bloque del conjunto 4 y via 1",contadorSuperadas);
  (*contadorTotales)++;
  assert(!is_dirty(2,3),"Esta limpio el bloque del conjunto 3 y via 2",contadorSuperadas);
  (*contadorTotales)++;
  assert(!is_dirty(1,1),"Esta limpio el bloque del conjunto 1 y via 1",contadorSuperadas);
  (*contadorTotales)++;
  destroy_cache();
  return contadorSuperadas;
}

int alPedirElLRUAlComienzoDevuelveLosBloquesSinUso(int* contadorTotales){
  int contadorSuperadas = 0;
  configuracion_t configuracion = {false,2,2,1,NULL};
  if(set_up_cache(configuracion)==ERROR){
    return contadorSuperadas;
  }
  init();

  assert(find_lru(0)==0,"Devolvio el bloque 0 del primer conjunto",contadorSuperadas);
  (*contadorTotales)++;
  write_byte(0,'A');

  assert(find_lru(0)==1,"Devolvio el bloque 1 del primer conjunto",contadorSuperadas);
  (*contadorTotales)++;
```

```
  write_byte(1,'S');

  int conjunto = find_set(32);
  assert(find_lru(conjunto)==0,"Devolvio el bloque 0 de otro conjunto",contadorSuperadas);
  (*contadorTotales)++;
  write_byte(32,'D');

  assert(find_lru(conjunto)==1,"Devolvio el bloque 1 del otro conjunto",contadorSuperadas);
  (*contadorTotales)++;
  write_byte(4128,'U'); // mapea al mismo conjunto que el 32
  write_byte(4128,'d');
  write_byte(4128,'g');

  assert(find_lru(conjunto)==0,"Se usa un bloque y se pide el LRU, devuelve el 0",contadorSuperad
  (*contadorTotales)++;

  destroy_cache();
  return contadorSuperadas;
}

int escriboBytesEnDiferentesPosicionesYAlPedirlosLosDevuelve(int* contadorTotales){
  int contadorSuperadas = 0;
  configuracion_t configuracion = {false,4,4,1,NULL};
  if(set_up_cache(configuracion)==ERROR){
    return contadorSuperadas;
  }
  init();

  write_byte(0x0,'A');
  write_byte(0x2,'B');
  write_byte(0x4,'C');
  write_byte(0x6,'D');
  write_byte(0x8,'E');

  assert(read_byte(0x0)=='A',"Devolvio el caracter esperado (A)",contadorSuperadas);
  (*contadorTotales)++;

  assert(read_byte(0x2)=='B',"Devolvio el caracter esperado (B)",contadorSuperadas);
  (*contadorTotales)++;

  assert(read_byte(0x4)=='C',"Devolvio el caracter esperado (C)",contadorSuperadas);
  (*contadorTotales)++;

  assert(read_byte(0x6)=='D',"Devolvio el caracter esperado (D)",contadorSuperadas);
  (*contadorTotales)++;

  assert(read_byte(0x8)=='E',"Devolvio el caracter esperado (E)",contadorSuperadas);
  (*contadorTotales)++;

  write_byte(0x0,'H');
  assert(read_byte(0x0)=='H',"Devolvio el caracter esperado al remplazar un valor (H)",contadorSu
  (*contadorTotales)++;

  write_byte(0x8,'W');
```

```
  assert(read_byte(0x8)=='W',"Devolvio el caracter esperado al remplazar un valor (W)",contadorSu
  (*contadorTotales)++;

  assert(get_miss_rate()==35,"El miss rate devuelto es el esperado (35)",contadorSuperadas);
  (*contadorTotales)++;
  destroy_cache();
  return contadorSuperadas;
}

int seVaModificandoLaCacheYPidiendoElMissRate(int* contadorTotales){
  int contadorSuperadas = 0;
  configuracion_t configuracion = {false,1,16,32,NULL};
  if(set_up_cache(configuracion)==ERROR){
    return contadorSuperadas;
  }
  init();

  write_byte(0,'E'); //MISS
  assert(get_miss_rate()==100,"Devolvio el miss rate esperado (100)",contadorSuperadas);
  (*contadorTotales)++;
  assert(!was_hit(),"Fue miss",contadorSuperadas);
  (*contadorTotales)++;

  write_byte(1,'a'); //HIT
  assert(get_miss_rate()==50,"Devolvio el miss rate esperado (50)",contadorSuperadas);
  (*contadorTotales)++;
  assert(was_hit(),"Fue hit",contadorSuperadas);
  (*contadorTotales)++;
  assert(is_dirty(0,find_set(1)),"Esta marcado como dirty al escribir",contadorSuperadas);
  (*contadorTotales)++;

  assert(!is_dirty(0,find_set(32)),"Esta limpio el bloque de la direccion 32",contadorSuperadas);
  (*contadorTotales)++;
  write_byte(32,'R'); //MISS
  assert(get_miss_rate()==66,"Devolvio el miss rate esperado (66)",contadorSuperadas);
  (*contadorTotales)++;
  assert(!was_hit(),"Fue miss",contadorSuperadas);
  (*contadorTotales)++;
  assert(is_dirty(0,find_set(32)),"Esta dirty el bloque de la direccion 32",contadorSuperadas);
  (*contadorTotales)++;

  write_byte(64,'y'); //MISS
  assert(get_miss_rate()==75,"Devolvio el miss rate esperado (75)",contadorSuperadas);
  (*contadorTotales)++;
  assert(!was_hit(),"Fue miss",contadorSuperadas);
  (*contadorTotales)++;

  write_byte(66,'7'); //HIT
  assert(get_miss_rate()==60,"Devolvio el miss rate esperado (60)",contadorSuperadas);
  (*contadorTotales)++;
  assert(was_hit(),"Fue hit",contadorSuperadas);
  (*contadorTotales)++;

  write_byte(120,'1'); //MISS
```

```
  write_byte(1024,'2'); //MISS
  write_byte(2048,'3'); //MISS
  write_byte(2400,'4'); //MISS
  write_byte(2300,'5'); //MISS
  assert(get_miss_rate()==80,"Devolvio el miss rate esperado (80)",contadorSuperadas);
  (*contadorTotales)++;

  destroy_cache();
  return contadorSuperadas;
}

int main(){
  int contadorSuperadas = 0,contadorTotales = 0;

  contadorSuperadas += sePideCacheSinAccesosDevuelveCero(&contadorTotales);
  contadorSuperadas += alPedirUbicacionesDelCacheSinUsoTodasEstanLimpias(&contadorTotales);
  contadorSuperadas += alPedirElLRUAlComienzoDevuelveLosBloquesSinUso(&contadorTotales);
  contadorSuperadas += escriboBytesEnDiferentesPosicionesYAlPedirlosLosDevuelve(&contadorTotales)
  contadorSuperadas += seVaModificandoLaCacheYPidiendoElMissRate(&contadorTotales);

  verificar_si_paso_las_pruebas(contadorSuperadas,contadorTotales);
}
```

8.5. 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, en que se puedan pasar por parámetro el número de vías, la capacidad y el tamaño de bloque. La política de reemplazo será LRU y la política de escritura

será 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 V, el bit D, el *tag*, y un campo que permita implementar la política de LRU.

6. Programa

6.1. Comportamiento deseado

Primero, usamos la opción `-h` para ver el mensaje de ayuda:

```
$ tp2 -h
Usage:
  tp2 -h
  tp2 -V
  tp2 options archivo
Options:
  -h,  --help       Imprime ayuda.
  -V,  --version    Versión del programa.
  -o,  --output     Archivo de salida.
  -w,  --ways       Cantidad de vías.
  -cs, --cachesize  Tamaño del caché en kilobytes.
  -bs, --blocksize  Tamaño de bloque en bytes.
Examples:
  tp2 -w 4 -cs 8 -bs 16 prueba1.mem
```

El tamaño del caché es en total, sin contar metadatos: si tiene 4 KB de tamaño , 4 vías y bloques de 256 bytes, tendrá 4 bloques por vía. Si falta alguno de los parámetros del cache o el archivo de entrada no es suministrado, el programa debe reportarlo como un error.

6.2. Primitivas de caché

Se deben implementar las siguientes primitivas:

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

- La función `init()` debe inicializar los bloques de la caché como inválidos, la memoria simulada en 0 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 en memoria los datos contenidos en el bloque `setnum` de la vía `way`.
- La función `read_byte(address)` debe retornar el valor correspondiente a la posición de memoria `address`, buscándolo primero en el caché.
- 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 funciones 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:

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

- Los comandos de la forma "init" se ejecutan llamando a la función `init` para inicializar la caché y la memoria simulada.
- Los comandos de la forma "`R ddddd`" se ejecutan llamando a la función `read_byte(ddddd)` e imprimiendo el resultado y si es hit o miss.
- Los comandos de la forma "`W ddddd, vvv`" se ejecutan llamando a la función `write_byte(int ddddd, char vvv)` e imprimiendo si es hit o miss.

- 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. Pruebas

Se deberá incluir la salida que produzca el programa con los siguientes archivos de prueba, para las siguientes cachés: [4 KB, 4WSA, 32bytes] y [16KB, una vía, 128 bytes].

- prueba1.mem
- prueba2.mem

8. Informe

El informe deberá incluir:

- Este enunciado;
- Una descripción detallada de las decisiones de diseño y el comportamiento deseado para el caché.
- El código fuente completo del programa.

9. Fecha de entrega

La fecha de entrega y presentación es el jueves 17 de Diciembre de 2020.

Referencias

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