Trabajo práctico 2: Memorias Caché

Menniti, Sebastián Ezequiel - *Padrón 93445*
mennitise@gmail.com
Pérez, Miguel - *Padrón 94708*
miguelangelperez909@gmail.com
Prystupiuk, Maximiliano - *Padrón 94853*
mprystupiuk@gmail.com

2do. Cuatrimestre de 2018

66.20 Organización de Computadoras
Facultad de Ingeniería, Universidad de Buenos Aires

Repositorio
https://github.com/mennitise/cache4WSA

1 de noviembre de 2018

Resumen

El trabajo consiste en simular una memoria caché asociativa por conjuntos de cuatro vías, de 4KB de capacidad, bloques de 64 bytes, política de reemplazo LRU y política de escritura WB/WA. Y también de la simulación de una memoria principal de 64 KB con espacio de direcciones de 16 bits. Dichas simulaciones se realizarán utilizano el lenguaje C.

Índice

1. Introducción

Para este trabajo realizamos una simulacion de una memoria cache, una memoria principal, y un programa que, utilizando estas memorias, lee un archivo y realiza lecturas, escrituras y el calculo del miss rate correspondiente.

2. Desarrollo

2.1. Memoria Caché

Se implemento una simulacion de una memoria caché asociativa por conjuntos de cuatro vías, de 4KB de capacidad, bloques de 64 bytes, política de reemplazo LRU y política de escritura WB/WA.

Asumiendo que el espacio de direcciones es de 16 bits, y hay entonces una memoria principal a simular con un tamaño de 64KB.

Estas memorias se implementaron como variables globales. Cada bloque de la memoria caché cuenta con su metadata, incluyendo el bit D, el tag, y un campo que permite implementar la política de LRU.

2.2. Primitivas

```
void init()
```

Inicializa los bloques de la caché como inválidos y la tasa de misses a 0.

```
int find_set(int address)
```

Devuelve el conjunto de caché al que mapea la dirección address.

```
int find_lru(int setnum)
```

Devuelve 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.

```
int is_dirty(int way, int setnum)
```

Devuelve el estado del bit D del bloque correspondiente.

```
void read_block(int blocknum)
```

Lee el bloque blocknum de memoria y lo guarda en el lugar que le corresponda en la memoria caché.

```
void write_block(int way, int setnum)
```

Escribe los datos contenidos en el bloque setnum de la vía way.

```
int read_byte(int address)
```

Retorna el valor correspondiente a la posición address.

```
int write_byte(int address, char value)
```

Escribe el valor value en la posición correcta del bloque que corresponde a address.

```
int get_miss_rate()
```

Devuelve el porcentaje de misses desde que se inicializó el cache.

2.3. Programa

Con la implementacion de las memorias nombradas anteriormente se desarrollo un programa que lee de un archivo una serie de comandos,que deben tener 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, además, chequea que los valores de los argumentos a los comandos estén dentro del rango de direcciones y valores antes de llamar a la funciones, e imprimen un mensaje de error informativo cuando corresponda.

2.4. Correr el programa

Todas las instrucciones dadas a continuacion deben correrse en la carpeta /src del proyecto.

2.4.1. Proceso de Compilación

Para la compilación del programa alcanza solo con la corrida del comando make, de la siguiente forma:

```
$ make
```

2.4.2. Corrida

Para la corrida del programa, luego de haber compilado, se debe utilizar, para un archivo ’archivo.mem’, el siguiente comando:

```
$ ./cache archivo.mem
```

2.4.3. Pruebas

Para la corrida del script de pruebas se debe ejecutar el siguiente comando:

```
$ ./run_all_tests.sh
```

2.4.4. Tiempos

Para la corrida del script de medicion de tiempos de corrida se debe ejecutar el siguiente comando:

```
$ ./times.sh
```

3. Código fuente

3.1. Memoria principal

3.1.1. Descripción

Código de la implementación de la memoria principal.

3.1.2. Contenido

```h
#ifndef MAIN_MEMORY_H
#define MAIN_MEMORY_H

#define MAIN_MEMORY_SIZE        64 * 1024 // 64KB
#define MAIN_MEMORY_BLOCK_SIZE  64 // 64B
#define MAIN_MEMORY_BLOCKS      ( 64 * 1024 ) / 64

typedef struct main_memory_block{
    char* data;
}main_memory_block_t;

typedef struct main_memory{
    int size;
    main_memory_block_t* blocks[MAIN_MEMORY_BLOCKS];
    size_t blocks_amount;
}main_memory_t;

extern main_memory_t* MAIN_MEMORY;

/* The init() function initialize the blocks of the main memory
    as invalid and the rate of misses to 0. */
void init_main_memory();

/* The destroy() function destroys the Main Memory*/
void destroy_main_memory();

#endif // MAIN_MEMORY_H
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#include "main_memory.h"

/* ———————— MAIN MEMORY BLOCK DEFINITION ———————— */

main_memory_block_t* init_main_memory_block(){
    main_memory_block_t* block = malloc(sizeof(main_memory_block_t));
    if (!block){
        printf("ERROR: Can't Initialize blocks from main memory\n");
        abort();
    }

    block->data = malloc(MAIN_MEMORY_BLOCK_SIZE * sizeof(char*));
    if (!block->data){
        printf("ERROR: Can't Initialize data blocks from main memory\n");
        abort();
    }
```

```
22
23      return block;
24  }
25
26  void destroy_main_memory_block(main_memory_block_t* block){
27      if(block) free(block->data);
28      free(block);
29  }
30
31  /* ------------------------------------------------------------ */
32
33  /* ------------------- MAIN MEMORY DEFINITION ------------------- */
34
35  void init_main_memory(){
36      // printf("Initialize Main Memory...\n");
37
38      MAIN_MEMORY = malloc(sizeof(main_memory_t));
39      if (!MAIN_MEMORY){
40          printf("ERROR: Can't Initialize Main Memory\n");
41          abort();
42      }
43
44      MAIN_MEMORY->blocks_amount = MAIN_MEMORY_BLOCKS;
45      MAIN_MEMORY->size = 64 * 1024;
46
47      for (int i = 0; i < MAIN_MEMORY->blocks_amount; ++i){
48          MAIN_MEMORY->blocks[i] = init_main_memory_block();
49      }
50  }
51
52  void destroy_main_memory(){
53      if (MAIN_MEMORY){
54          for (int i = 0; i < MAIN_MEMORY->blocks_amount; ++i){
55              destroy_main_memory_block(MAIN_MEMORY->blocks[i]);
56          }
57      }
58      free(MAIN_MEMORY);
59      // puts("Main Memory destroyed!");
60  }
61
62  /* ------------------------------------------------------------ */
```

3.2. Memoria Caché

3.2.1. Descripción

Código de la implementación de la memoria caché.

3.2.2. Contenido

```
1  #ifndef CACHE_H
2  #define CACHE_H
3
4  typedef struct cache cache_t;
5
6  extern cache_t* CACHE;
7
8  /* The init() function initialize the blocks of the cache
9      as invalid and the rate of misses to 0. */
```

```
10 void init();
11
12 /* The find_set(int address) function returns the cache
13     set that to which the address address maps. */
14 int find_set(int address);
15
16 /* The find_lru(int setnum) function returns the least
17     recently used block within a set (or one of them if
18     there is more than one), using the corresponding field
19     in the metadata of the blocks in the set. */
20 int find_lru(int setnum);
21
22 /* The is_dirty(int way, int blocknum) function returns
23     the state of the D bit of the corresponding block. */
24 int is_dirty(int way, int setnum);
25
26 /* The read_block(int blocknum) function reads the
27     block blocknum from memory and stores it in the
28     corresponding place in the cache. */
29 void read_block(int blocknum);
30
31 /* The write_block function (int way, int setnum)
32     writes the data contained in the setnum block of
33     the way. */
34 void write_block(int way, int setnum);
35
36 /* The read_byte(address) function returns the
37     value corresponding to the address position. */
38 int read_byte(int address);
39
40 /* The write_byte(int address, char value) function
41     writes the value value to the correct position of
42     the block corresponding to the address. */
43 int write_byte(int address, char value);
44
45 /* The get_miss_rate() function returns the
46     percentage of misses since the cache was
47     initialized. */
48 int get_miss_rate();
49
50 /* The destroy() function destroys the Cache*/
51 void destroy();
52
53 #endif // CACHE_H
```

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include <time.h>
5  #include <math.h>
6  #include <unistd.h>
7
8  #include "main_memory.h"
9  #include "cache.h"
10
11 #define WAYS            4
12 #define CACHE_SIZE      4 * 1024 // 4KB
13 #define BLOCK_SIZE      64 // 64B
14 #define BLOCKS          ( 4 * 1024 ) / 64
```

```
15
16 #define BITS_TAG          6
17 #define BITS_TAG_INIT     0
18 #define BITS_TAG_END      5
19
20 #define BITS_INDEX        4
21 #define BITS_INDEX_INIT   6
22 #define BITS_INDEX_END    9
23
24 #define BITS_OFFSET       6
25 #define BITS_OFFSET_INIT  10
26 #define BITS_OFFSET_END   15
27
28 #define BITS_ADDRESS      16
29
30 /*
31       ------------------
32      | tag   |idx |offset|
33      |-------|----|------|
34      |       |    |      |
35      |-------|----|------|
36      |-------|----|------|
37
38 */
39
40 /* ---------------------- UTILITIES ---------------------- */
41
42 char* int_to_binary(int n, size_t bits){
43     int c, d, count;
44     char *pointer;
45
46     count = 0;
47     pointer = (char*)malloc(bits + 1);
48     if (!pointer){
49         puts("ERROR: Can't Initialize blocks from cache");
50         abort();
51     }
52
53     for (c = bits - 1; c >= 0; c--){
54         d = n >> c;
55
56         if (d & 1)
57             *(pointer+count) = 1 + '0';
58         else
59             *(pointer+count) = 0 + '0';
60
61         count++;
62     }
63     *(pointer+count) = '\0';
64
65     return  pointer;
66 }
67
68 int binary_to_int(char* bin, size_t bits){
69     int result = 0;
70     int count = bits - 1;
71     for (int i = 0; i < bits; ++i){
72         if (bin[i] == '1'){
73             result += pow(2, count);
```

```
74          }
75          count--;
76      }
77      return result;
78  }
79
80  char* get_from_address(char* address, int len, int begin, int end){
81      char* aux = (char*)malloc(len + 1);
82      if (!aux) return NULL;
83      int count = 0;
84      for (int i = begin; i <= end; ++i){
85          *(aux + count) = address[i];
86          count++;
87      }
88      return aux;
89  }
90
91  char* get_tag(char* address){
92      return get_from_address(address, BITS_TAG, BITS_TAG_INIT, BITS_TAG_END)
            ;
93  }
94
95  char* get_index(char* address){
96      return get_from_address(address, BITS_INDEX, BITS_INDEX_INIT,
            BITS_INDEX_END);
97  }
98
99  char* get_offset(char* address){
100     return get_from_address(address, BITS_OFFSET, BITS_OFFSET_INIT,
            BITS_OFFSET_END);
101 }
102
103 /* ----------------------BLOCK DEFINITION---------------------- */
104
105 typedef struct block{
106     time_t lastUpdate;
107     int valid;
108     int dirty;
109     int tag;
110     char* data;
111 }block_t;
112
113 block_t* init_block(){
114
115     block_t* block = malloc(sizeof(block_t));
116     if (!block){
117         puts("ERROR: Can't Initialize blocks from cache");
118         abort();
119     }
120
121     block->data = malloc(BLOCK_SIZE * sizeof(char*));
122     if (!block->data){
123         puts("ERROR: Can't Initialize data blocks from cache");
124         abort();
125     }
126
127     block->lastUpdate = time(NULL);
128     block->valid = 0;
129     block->dirty = 0;
```

```
130
131     return block;
132 }
133
134 void destroy_block(block_t* block){
135     if(block) free(block->data);
136     free(block);
137 }
138
139 /* ------------------------------------------------------------ */
140
141 /* ------------------- SET DEFINITION ------------------------- */
142
143 typedef struct cache_set{
144     block_t* blocks[BLOCKS];
145     size_t blocks_amount;
146 }cache_set_t;
147
148 cache_set_t* init_cache_set(){
149
150     cache_set_t* cache_set = malloc(sizeof(cache_set_t));
151     if (!cache_set){
152         puts("ERROR: Can't Initialize sets from cache");
153         abort();
154     }
155
156     cache_set->blocks_amount = BLOCKS;
157
158     for (int i = 0; i < cache_set->blocks_amount; ++i){
159         cache_set->blocks[i] = init_block();
160     }
161
162     return cache_set;
163 }
164
165 void destroy_cache_set(cache_set_t* cache_set){
166     if (cache_set){
167         for (int i = 0; i < cache_set->blocks_amount; ++i){
168             destroy_block(cache_set->blocks[i]);
169         }
170     }
171     cache_set->blocks_amount = 0;
172     free(cache_set);
173 }
174
175 /* ------------------------------------------------------------ */
176
177 /* ------------------- CACHE DEFINITION ----------------------- */
178
179 typedef struct cache{
180     int size;
181     cache_set_t* ways[WAYS];
182     size_t amount_ways;
183     int misses;
184     int hits;
185
186 }cache_t;
187
188 void init(){
```

```
189     // puts("Initialize Cache...");
190
191     CACHE = malloc(sizeof(cache_t));
192     if (!CACHE){
193         puts("ERROR: Can't Initialize Cache");
194         abort();
195     }
196
197     CACHE->amount_ways = WAYS;
198     CACHE->size = 64 * 1024;
199     CACHE->hits = 0;
200     CACHE->misses = 0;
201
202     for (int i = 0; i < CACHE->amount_ways; ++i){
203         CACHE->ways[i] = init_cache_set();
204     }
205 }
206
207 void destroy(){
208     if (CACHE){
209         for (int i = 0; i < CACHE->amount_ways; ++i){
210             destroy_cache_set(CACHE->ways[i]);
211         }
212     }
213     free(CACHE);
214     // puts("Cache destroyed!");
215 }
216
217 /* ------------------------------------------------------------ */
218
219 int find_set(int address){
220     char* bin_address = int_to_binary(address, BITS_ADDRESS);
221
222     char* tag = get_tag(bin_address);
223     char* index = get_index(bin_address);
224     char* offset = get_offset(bin_address);
225
226     int founded = -1;
227
228     if (!tag || !index || !offset){
229         puts("ERROR: Don't have space for initialize variables");
230         abort();
231     }
232
233     for (int i = 0; i < CACHE->amount_ways; ++i){
234         if ( (CACHE->ways[i]->blocks[binary_to_int(index, BITS_INDEX)]->
                valid == 1) &&
235               (CACHE->ways[i]->blocks[binary_to_int(index, BITS_INDEX)]->tag
                    == binary_to_int(tag, BITS_TAG))
236         ){
237             founded = i;
238             break;
239         }
240     }
241
242     free(tag);
243     free(index);
244     free(offset);
245
```

```
246     free(bin_address);
247     return founded;
248 }
249
250 int find_lru(int setnum){
251     if (!CACHE){
252         puts("ERROR: The Cache isn't initialized");
253         return 0;
254     }
255
256     int way_lru_block = 0;
257
258     time_t timeA, timeB;
259     double seconds;
260
261     for (int i = 0; i < CACHE->amount_ways; ++i){
262         timeA = CACHE->ways[i]->blocks[setnum]->lastUpdate;
263         timeB = CACHE->ways[way_lru_block]->blocks[setnum]->lastUpdate;
264         seconds = difftime(timeA, timeB);
265
266         // printf("%ld %ld %f\n", timeA, timeB, seconds);
267
268         if (seconds < 0) {
269             way_lru_block = i;
270         }
271     }
272     return way_lru_block;
273 }
274
275 int is_dirty(int way, int setnum){
276     if(!CACHE){
277         puts("ERROR: The Cache isn't initialized");
278         return 0;
279     }
280
281     if(CACHE->amount_ways < way){
282         puts("ERROR: The Cache Set especified don't exists");
283         return 0;
284     }else if(!CACHE->ways[way-1]){
285         puts("ERROR: The Cache Set especified isn't initialized");
286         return 0;
287     }
288
289     if(CACHE->ways[way-1]->blocks_amount < setnum){
290         puts("ERROR: The Block especified in the Cache Set don't exists");
291         return 0;
292     }else if(!CACHE->ways[way-1]->blocks[setnum-1]){
293         puts("ERROR: This Block especified in the Cache Set isn't
                initialized");
294         return 0;
295     }
296
297     return CACHE->ways[way-1]->blocks[setnum-1]->dirty;
298 }
299
300 void read_block(int blocknum){
301
302     if(MAIN_MEMORY->blocks_amount < blocknum){
303         puts("ERROR: The Main Memory block especified doesn't exists");
```

```
304         return ;
305     }else if(!MAIN_MEMORY->blocks[blocknum-1]){
306         puts("ERROR: The Main Memory data especified isn't initialized");
307         return ;
308     }
309
310     // Simulamos tiempo de busqueda en memoria principal
311     sleep(4);
312
313     // OBTENGO EL TAG E INDEX DEL BLOCKNUM. SON 10 BITS LOS QUE IDENTIFICA
            A UN BLOQUE DE LA MEMORIA PRINCIPAL
314     char* bin_blocknum = int_to_binary(blocknum, BITS_TAG + BITS_INDEX);
315     char* tag = get_tag(bin_blocknum);
316     char* index = get_index(bin_blocknum);
317
318     int memory_changed = 0;
319
320     //LEO LOS DATOS DE MEMORIA PRINCIPAL
321     char* data_in_memory = malloc(BLOCK_SIZE * sizeof(char));
322     strcpy(data_in_memory, MAIN_MEMORY->blocks[blocknum]->data);
323
324     // VER SI HAY ALGUN BLOQUE LIBRE EN CACHE
325     for (int i = 0; i < CACHE->amount_ways; ++i){
326         if (CACHE->ways[i]->blocks[binary_to_int(index, BITS_INDEX)]->valid
                == 0){
327             strcpy(CACHE->ways[i]->blocks[binary_to_int(index, BITS_INDEX)
                    ]->data, data_in_memory);
328             CACHE->ways[i]->blocks[binary_to_int(index, BITS_INDEX)]->dirty
                    = 0; //Quedaria igual que en memoria principal
329             CACHE->ways[i]->blocks[binary_to_int(index, BITS_INDEX)]->valid
                    = 1;
330             CACHE->ways[i]->blocks[binary_to_int(index, BITS_INDEX)]->
                    lastUpdate = time(NULL);
331             CACHE->ways[i]->blocks[binary_to_int(index, BITS_INDEX)]->tag =
                     binary_to_int(tag, BITS_TAG);
332             memory_changed = 1;
333             break;
334         }
335     }
336
337     // SI NO SE ENCONTRO BLOQUE LIBRE HAY QUE AGARRAR EL BLOQUE LRU
338     if (memory_changed != 1) {
339         int way_lru = find_lru(binary_to_int(index, BITS_INDEX));
340         if (CACHE->ways[way_lru]->blocks[binary_to_int(index, BITS_INDEX)
                ]->dirty == 1){
341             //strcpy(MAIN_MEMORY->blocks[binary_to_int(bin_blocknum,
                    BITS_TAG + BITS_INDEX)]->data, CACHE->ways[way_lru]->blocks[
                    binary_to_int(index, BITS_INDEX)]->data);
342             write_block(way_lru,binary_to_int(index, BITS_INDEX));
343         }
344
345         strcpy(CACHE->ways[way_lru]->blocks[binary_to_int(index, BITS_INDEX
                )]->data, data_in_memory);
346         CACHE->ways[way_lru]->blocks[binary_to_int(index, BITS_INDEX)]->
                dirty = 0;
347         CACHE->ways[way_lru]->blocks[binary_to_int(index, BITS_INDEX)]->
                valid = 1;
348         CACHE->ways[way_lru]->blocks[binary_to_int(index, BITS_INDEX)]->
                lastUpdate = time(NULL);
```

```
349         CACHE->ways[way_lru]->blocks[binary_to_int(index, BITS_INDEX)]->tag
                = binary_to_int(tag, BITS_TAG);
350     }
351 }
352
353 void write_block(int way, int setnum){
354
355     if(!CACHE){
356         puts("ERROR: The Cache isn't initialized");
357         return ;
358     }
359
360     if(CACHE->amount_ways < way){
361         puts("ERROR: The Cache Set especified don't exists");
362         return ;
363     }else if(!CACHE->ways[way]){
364         puts("ERROR: The Cache Set especified isn't initialized");
365         return ;
366     }
367
368     if(CACHE->ways[way]->blocks_amount < setnum){
369         puts("ERROR: The Block especified in the Cache Set don't exists");
370         return ;
371     }else if(!CACHE->ways[way]->blocks[setnum]){
372         puts("ERROR: This Block especified in the Cache Set isn't
                initialized");
373         return ;
374     }
375
376     // Simulamos tiempo de busqueda en cache
377     sleep(4);
378
379     // BUSQUEDA DEL BLOQUE EN CACHE
380     if (CACHE->ways[way]->blocks[setnum]->valid == 1){
381         // printf("     Write block: Encuentra el bloque en cache\n");
382         char* data = CACHE->ways[way]->blocks[setnum]->data;
383         int tag = CACHE->ways[way]->blocks[setnum]->tag;
384         char* bin_tag = int_to_binary(tag, BITS_TAG);
385         char* bin_index = int_to_binary(setnum, BITS_INDEX);
386
387         // Obtengo el numero de bloque de la memoria principal
388         char* bin_blocknum;
389         bin_blocknum = malloc(strlen(bin_tag)+strlen(bin_index)+1);
390         strcpy(bin_blocknum, bin_tag);
391         strcat(bin_blocknum, bin_index);
392
393         int blocknum = binary_to_int(bin_blocknum, BITS_TAG + BITS_INDEX);
394
395         // Guardo una copia de los datos en el bloque de la memoria
                principal
396         strcpy(MAIN_MEMORY->blocks[blocknum]->data, data);
397
398         free(bin_blocknum);
399     }
400     else{
401         // printf("     Write Block: No se encontro el bloque en cache\n");
402     }
403 }
404
```

```
405 int read_byte(int address){
406     char* bin_address = int_to_binary(address, BITS_ADDRESS);
407 
408     char* tag = get_tag(bin_address);
409     char* index = get_index(bin_address);
410     char* offset = get_offset(bin_address);
411 
412     if (!tag || !index || !offset){
413         puts("ERROR: Don't have space for initialize variables");
414         abort();
415     }
416 
417     char value;
418     int set;
419 
420     // Simulamos tiempo de busqueda en cache
421     sleep(1);
422 
423     // BUSCA EL VALOR EN CACHE PARA RETORNARLO
424     set = find_set(address);
425     if (set != -1){
426         CACHE->hits++;
427         value = *(CACHE->ways[set]->blocks[binary_to_int(index, BITS_INDEX)
                ]->data + binary_to_int(offset, BITS_OFFSET));
428     } else {
429         CACHE->misses++;
430         read_block(binary_to_int(bin_address, BITS_TAG + BITS_INDEX));
431         set = find_set(address);
432         if (set != -1){
433             value = *(CACHE->ways[set]->blocks[binary_to_int(index,
                    BITS_INDEX)]->data + binary_to_int(offset, BITS_OFFSET));
434         } else {
435             puts("ERROR: Reading block from main memory");
436             abort();
437         }
438     }
439 
440     free(tag);
441     free(index);
442     free(offset);
443 
444     free(bin_address);
445     return (int) value;
446 }
447 
448 int write_byte(int address, char value){
449     char* bin_address = int_to_binary(address, BITS_ADDRESS);
450     char* tag = get_tag(bin_address);
451     char* index = get_index(bin_address);
452     char* offset = get_offset(bin_address);
453 
454     if (!tag || !index || !offset){
455         puts("ERROR: Don't have space for initialize variables");
456         abort();
457     }
458 
459     char return_value;
460 
461     // Simulamos tiempo de busqueda en cache
```

```
462     sleep(1);
463
464     // BUSCAR SI EL BLOQUE SE ENCUENTRA EN CACHE PARA ESCRIBIR EL NUEVO
            VALOR
465     int set = find_set(address);
466     if (set != -1){
467         CACHE->hits++;
468         *(CACHE->ways[set]->blocks[binary_to_int(index, BITS_INDEX)]->data
                + binary_to_int(offset, BITS_OFFSET)) = value;
469         return_value = *(CACHE->ways[set]->blocks[binary_to_int(index,
                BITS_INDEX)]->data + binary_to_int(offset, BITS_OFFSET));
470         CACHE->ways[set]->blocks[binary_to_int(index, BITS_INDEX)]->dirty =
                1;
471         CACHE->ways[set]->blocks[binary_to_int(index, BITS_INDEX)]->valid =
                1;
472         CACHE->ways[set]->blocks[binary_to_int(index, BITS_INDEX)]->
                lastUpdate = time(NULL);
473         CACHE->ways[set]->blocks[binary_to_int(index, BITS_INDEX)]->tag =
                binary_to_int(tag, BITS_TAG);
474     } else {
475         // NO SE ENCUENTRA EL BLOQUE EN CACHE, LO TRAE DE MEMORIA
476         CACHE->misses++;
477         read_block(binary_to_int(bin_address, BITS_TAG + BITS_INDEX));
478         set = find_set(address);
479         if (set != -1){
480             *(CACHE->ways[set]->blocks[binary_to_int(index, BITS_INDEX)]->
                    data + binary_to_int(offset, BITS_OFFSET)) = value;
481             return_value = *(CACHE->ways[set]->blocks[binary_to_int(index,
                    BITS_INDEX)]->data + binary_to_int(offset, BITS_OFFSET));
482             CACHE->ways[set]->blocks[binary_to_int(index, BITS_INDEX)]->
                    dirty = 1;
483             CACHE->ways[set]->blocks[binary_to_int(index, BITS_INDEX)]->
                    valid = 1;
484             CACHE->ways[set]->blocks[binary_to_int(index, BITS_INDEX)]->
                    lastUpdate = time(NULL);
485             CACHE->ways[set]->blocks[binary_to_int(index, BITS_INDEX)]->tag
                     = binary_to_int(tag, BITS_TAG);
486         } else {
487             puts("ERROR: Reading block from main memory");
488             abort();
489         }
490     }
491
492     free(tag);
493     free(index);
494     free(offset);
495
496     free(bin_address);
497     return (int) return_value;
498 }
499
500 int get_miss_rate(){
501     if ((CACHE->misses + CACHE->hits) == 0) return 0;
502     return (float) CACHE->misses / ( (float) CACHE->misses + (float) CACHE
            ->hits ) * 100;
503 }
```

4. Casos de prueba

Para los casos de prueba de este trabajo realizamos un script que compila y corre el programa con varias entradas de archivos de prueba.

Este script se puede correr desde el proyecto (como se explica anteriormente en la sección Correr el programa, en el apartado pruebas) y ver los resultados, en donde se puede observar que el programa pasa exitosamente todos los casos probados.

Para obtener los resultados esperados realizamos un estudio de los casos de prueba dandonos los siguientes resultados:

4.1. Resultados

File	Instruction	Tag	Index	Offset	Miss = 1 / Hit = 0
pruebas1.mem	W 0, 255	0	0	0	1
	W 1024, 254	1	0	0	1
	W 2048, 248	2	0	0	1
	W 4096, 096	4	0	0	1
	W 8192, 192	8	0	0	1
	R 0	0	0	0	1
	R 1024	1	0	0	1
	R 2048	2	0	0	1
	R 8192	8	0	0	0
	MR	**89%**			
pruebas2.mem	R 0	0	0	0	1
	R 31	0	0	31	0
	W 64, 10	0	1	0	1
	R 64	0	1	0	0
	W 64, 20	0	1	0	0
	R 64	0	1	0	0
	MR	**33%**			
pruebas3.mem	W 128, 1	0	2	0	1
	W 129, 2	0	2	1	0
	W 130, 3	0	2	2	0
	W 131, 4	0	2	3	0
	R 1152	1	2	0	1
	R 2176	2	2	0	1
	R 3200	3	2	0	1
	R 4224	4	2	0	1
	R 128	0	2	0	1
	R 129	0	2	1	0
	R 130	0	2	2	0
	R 131	0	2	3	0
	MR	**50%**			
pruebas4.mem	W 0, 256	0	0	0	ERROR - long value
	W 1, 2	0	0	1	
	W 2, 3	0	0	2	
	W 3, 4	0	0	3	
	W 4, 5	0	0	4	
	R 0	0	0	0	
	R 1	0	0	1	
	R 2	0	0	2	
	R 3	0	0	3	
	R 4	0	0	4	
	R 4096	4	0	0	
	R 8192	8	0	0	
	R 0	0	0	0	
	R 1	0	0	1	
	R 2	0	0	2	
	R 3	0	0	3	
	R 4	0	0	4	
	MR	**ERROR**			
	R 131072	0	0	0	ERROR - long address
	R 4096	4	0	0	
	R 8192	8	0	0	

pruebas5.mem	R 4096	4	0	0	
	R 0	0	0	0	
	R 4096	4	0	0	
	MR	**ERROR**			

5. Mediciones

Para la medicion de tiempos, primero debemos tener en cuenta que se tomo en la implementacion de la cache y la memoria principal una simulacion del tiempo de lecura/escritura a disco y cache, haciendo que, para notar la diferencia, la velocidad de lectura/escritura a cache sea 4 veces mas rapida que a memoria principal. Sabiendo esto, armamos un scrit con tres casos de prueba, y tomamos medicion de los tiempos de ejecucion, arrojandonos los siguientes resultados:

- La corrida con un archivo con miss rate de 1 tardo en ejecutarse 70 segundos.
- La corrida con archivo con miss rate de 0.5 tardo en ejecutarse 34 segundos.
- La corrida con archivo con miss rate de 0.1 tardo en ejecutarse 14 segundos.

Podemos observar con estos resultados como la velocidad varia notablemente dependiendo del miss rate que obtengamos

El script utilizado para medir tiempos se encuentra subido en el repositorio del trabajo.

6. Conclusiones

Como podemos notar en la sección Mediciones, al realizar lecturas/escrituras a memoria principal, teniendo una memoria cache de por medio podemos mejorar en cantidades muy notables el tiempo de ejecucion, ya que los tiempos de acceso a memoria principal superan ampliamente los tiempos de acceso a memoria cache.

También observamos que cuanto menor sea el miss rate de la memoria cache mas hits se produciran, y por lo tanto, mas rapida sera la ejecucion del programa.

Ademas sabemos que las diferencias de velocidad de lectura/escritura a cache y a memoria, en la realidad, son mucho mas amplias que las planteadas por nosotros en este trabajo para simular esta brecha, lo que haria mucho mas lenta la ejecucion a mayor miss rate, por lo tanto la implementación de una memoria cache para mejorar estos tiempos es una implementación muy productiva.