Trabajo Práctico Nro 2
Organización de Computadoras

Lambre, Juan Manuel *95978*
juanmlambre@gmail.com
Israel, Pablo *95849*
pabloisrael94@gmail.com

Resumen

El siguiente informe explica y detalla el trabajo práctico número 2 de la materia.

Índice

1. Objetivo

Familiarizarse con el funcionamiento de la memoria cache implementando una simulación de una cache dada.

2. Introducción

2.1. Memoria Cache

Una caché es un componente de hardware o software que almacena datos para que las solicitudes futuras de esos datos se puedan atender con mayor rapidez, los datos almacenados en un caché pueden ser el resultado de un cálculo anterior o el duplicado de datos almacenados en otro lugar, generalmente, de velocidad de acceso más rápido. Se produce un acierto de caché cuando los datos solicitados se pueden encontrar en esta, mientras que un error de caché ocurre cuando no están dichos datos y se debe ir a buscar el dato a memoria principal. La lectura de la caché es más rápido que volver a calcular un resultado o leer desde un almacén de datos más lento. Por lo tanto, cuantas más solicitudes se puedan atender desde la memoria caché, más rápido funcionará el sistema.

2.1.1. Política de ubicación

Decide dónde debe colocarse un bloque de memoria principal que entra en la memoria caché. Las más utilizadas son:

Directa Al bloque i-ésimo de memoria principal le corresponde la posición i módulo n, donde n es el número de bloques de la memoria caché. Cada bloque de la memoria principal tiene su posición en la caché y siempre en el mismo sitio. Su inconveniente es que cada bloque tiene asignada una posición fija en la memoria caché y ante continuas referencias a palabras de dos bloques con la misma localización en caché, hay continuos fallos habiendo sitio libre en la caché.

Asociativa Los bloques de la memoria principal se alojan en cualquier bloque de la memoria caché, comprobando solamente la etiqueta de todos y cada uno de los bloques para verificar acierto. Su principal inconveniente es la cantidad de comparaciones que realiza.

Asociativa por conjuntos Cada bloque de la memoria principal tiene asignado un conjunto de la caché, pero se puede ubicar en cualquiera de los bloques que pertenecen a dicho conjunto. Ello permite mayor flexibilidad que la correspondencia directa y menor cantidad de comparaciones que la totalmente asociativa.

2.1.2. Política de reemplazo

Aleatoria el bloque es reemplazado de forma aleatoria.

Fifo se usa el algoritmo First In First Out (FIFO) (primero en entrar primero en salir) para determinar qué bloque debe abandonar la caché. Este algoritmo generalmente es poco eficiente.

Usado menos recientemente (LRU) Sustituye el bloque que hace más tiempo que no se ha usado en la caché, traeremos a caché el bloque en cuestión y lo modificaremos ahí.

Usado con menor frecuencia (LFU) Sustituye el bloque que ha experimentado menos referencias.

3. Código

3.1. Diseño del código

La memoria a simular es una cache asociativa por conjuntos de cuatro vías, de 4KB de capacidad, bloques de 64 bytes, política de reemplazo LRU y política de escritura WB/WA. Se asume que el espacio de direcciones es de 16 bits, y hay entonces una memoria principal a simular con un tamaño de 64KB. Estas memorias pueden ser implementadas como variables globales. Cada bloque de la memoria cache deber a contar con su metadata, incluyendo el bit D, el tag, y un campo que permita implementar la política de LRU.

El diseño del código fue realizado orientado a pseudo objetos mediante la utilización de structs dado que C no soporta el uso de objetos. Se modelo el dominio del problema con las siguientes entidades:

- Command: Esta entidad es la encargada de parsear las instrucciones que provienen del archivo de entrada.
- Cache: Esta entidad es el cache que simulamos, el cual contiene cuatro vías (cuatros vectores), un contador de la cantidad de accesos, otro de cantidad de misses y por último una referencia a la memoria para que pueda acceder a ella cuando lo necesite.
- Block: Esta entidad es simplemente un bloque de la memoria cache. Cuando se tiene que ir a buscar un dato al cache se busca mediante el índice en las 4 vías y se obtiene el bloque el cual es un vector y para obtener finalmente el dato en cuestión se utiliza el offset.
- Memory: Esta entidad simula la memoria principal a la cuál se accede directamente con la dirección del dato. También contiene un vector para almacenar la información.

3.2. Especificaciones del cache

Las especificaciones del cache se definen en el archivo $config.h$ como macros del preprocesador. Estas son:

- Tamaño de cache: 4KB
- Tamaño de bloque: 64B
- Cantidad de vías: 4
- Tamaño de la memoria ppal: 64KB

De estas especificaciones se desprenden algunas propiedades tenidas en cuenta:

- Bits de dirección: 16
- Bits para el tag: 6
- Bits para el índice: 4
- Bits para el offset: 6

3.3. Repositorio

`https://github.com/JuanmaLambre/6620-tp2`

3.4. Instructivo de instalación

3.4.1. Cómo compilar el código

Ejecutar

```
$ bash build
```

Esto compila el código y genera el ejecutable *cache*. Se puede agregar el parámetro *debug* para compilar con flags de debugging

3.4.2. Cómo utilizar el programa

Uso:

```
$ ./cache direccion/del/archivo.mem
```

4. Casos de prueba

4.1. Prueba 1

```
W 0, 255
W 1024, 254
W 2048, 248
W 4096, 096
W 8192, 192
R 0
R 1024
R 2048
R 8192
MR
```

prueba1.mem

```
~/6620-tp2 $ ./cache test/inputs/prueba1.mem
255
254
248
96
192
255
254
248
192
MissRate 88%
~/6620-tp2 $
```

Output para prueba1.mem

Para la ejecución del programa com prueba1.mem, todos los comandos de *write* son misses compulsivos, dado que cada bloque es de 64B. Además, para todas las direcciones los bits de index (6-9) son 0, por lo que los bloques se almacenarán en el mismo set. Cuando el programa llegue a la escritura en 8192 el set 0 ya estará lleno, y el último bloque usado sera el bloque 0 en la vía 0, que será reemplazado.

Debido a este reemplazo, la primera lectura dará miss, pero reemplazará al último bloque usado, que será el de la vía 1 (bloque que comienza en 1024). Consecuentemente, la siguiente lectura dará miss, y reemplazará al bloque de la dirección 2048 en la vía 2. Lo mismo sucederá con la lectura de dirección 2048, que se ubicará en la vía 3. Sin embargo, la última lectura de la dirección 8192 dará hit dado que se encontrará en la vía 0.

En total se tienen entonces 9 accesos de los cuales 8 son misses, lo que resulta en un 88 % de miss rate.

4.2. Prueba 2

```
R 0
R 31
W 64, 10
R 64
W 64, 20
R 64
MR
```

prueba2.mem

```
~/6620-tp2 $ ./cache test/inputs/prueba2.mem
0
0
10
10
20
20
MissRate 33%
~/6620-tp2 $
```

Output para prueba2.mem

La primer lectura en la dirección 0 será un miss compulsivo, pero traerá a cache los primeros 64B de memoria. Por esta razón, la siguiente lectura en la dirección 31 será hit. Ambas devuelven 0 porque así se inicializa la memoria.

La primera escritura es en la dirección 64, por lo que será otro miss compulsivo. Sin embargo, todas las siguientes operaciones se hacen sobre esta misma dirección, por lo que serán hits.

En definitiva, hay 2 misses en 6 accesos, resultando en un 33 % de miss rate.

4.3. Prueba 3

```
W 128, 1
W 129, 2
W 130, 3
W 131, 4
R 1152
R 2176
R 3200
R 4224
R 128
R 129
R 130
R 131
MR
```

prueba3.mem

```
~/6620-tp2 $ ./cache test/inputs/prueba3.mem
1
2
3
4
0
0
0
0
1
2
3
4
MissRate 50%
~/6620-tp2 $
```

Output para prueba3.mem

En esta prueba todas las direcciones comparten en mismo set (index=2), por lo que compiten constantemente por el espacio en la cache.

La primera escritura es un miss compulsivo y se almacena en la vía 0. Sin embargo las siguientes tres perteneces al mismo bloque y son hit.

Las primeras cuatro lecturas son misses compulsivos también, y se van almacenando en las vías libres. Para la lectura en la dirección 4224, todas las vías están ocupadas, y la menos recientemente usada será la primera de todas (vía 0).

Cuando llega la lectura de la dirección 128, el bloque no estará presente en la vía 0 porque acaba de ser reemplazada, lo que provoca un miss. Pero, al igual que al principio, las siguientes y últimas tres lecturas perteneces al mismo bloque, resultando en hit.

En total, de 12 accesos a memoria hay 6 que son hit, y el miss rate es de 50 %

4.4. Prueba 4

```
W 0, 256
W 1, 2
W 2, 3
W 3, 4
W 4, 5
R 0
R 1
R 2
R 3
R 4
R 4096
R 8192
R 0
R 1
R 2
R 3
R 4
MR
```

prueba4.mem

```
~/6620-tp2 $ ./cache test/inputs/prueba4.mem
0
2
3
4
5
0
2
3
4
5
0
0
0
2
3
4
5
MissRate 17%
~/6620-tp2 $
```

Output para prueba4.mem

La primera escritura de la prueba 4 inevitablemente es un miss compulsivo, pero tanto las escrituras que le siguen como las primeras 5 lecturas se realizan sobre el mismo bloque, por lo que son hits.

Las lecturas sobre las direcciones 4096 y 8192 son misses compulsivos y se guardan en el mismo conjunto que el bloque 0, pero habiendo 4 vías disponibles no se reemplaza ningún bloque en cache. Esto permite que las últimas cinco lecturas sean hits, dado que el bloque 0 ya estaba almacenado en cache.

Se tienen entonces 17 lecturas, de las cuales sólo 3 son misses, por lo que el miss rate es de 17 %.

4.5. Prueba 5

```
R 131072
R 4096
R 8192
R 4096
R 0
R 4096
MR
```

prueba5.mem

```
~/6620-tp2 $ ./cache test/inputs/prueba5.mem
Address 131072 overflows
0
0
0
0
0
MissRate 60%
~/6620-tp2 $
```

Output para prueba5.mem

En estas pruebas, la primera lectura es en una dirección que excede el tamaño de la memoria principal, por lo que se imprime por stderr un mensaje de error y se descarta el comando. Vale aclarar que no cuenta como un acceso a memoria.

Las direcciones de los comandos siguientes son sólo tres, por lo que entran en la cache aunque pertenezcan al mismo conjunto (cosa que sucede). Los misses a las direcciones 0, 4096 y 8192 son compulsivos, y las demás lecturas son a la dirección 4096, que está en cache.

En definitiva, hay 5 accesos a memoria de los cuales sólo 2 son hits. El miss rate para estas pruebas es de 60 %

5. Conclusiones

El trabajo realizado nos permitió conocer y estudiar en un nivel práctico como funciona una memoria cache y el proceso necesario para obtener la información requerida. Pudimos reproducir todo el proceso que se realiza desde que llega una dirección de memoria para obtener hasta obtenerlo pasando por el cache y yendo a memoria principal de ser necesario (Miss).

Al ser una simulación en vez de ser un cache real, no se obtiene el resultado que uno busca cuando agrega una memoria cache el cuál es reducir los tiempos significativamente. En esta simulación tanto cuando se va a buscar un dato al cache o cuando se va a buscar a memoria principal siempre accede al mismo lugar físico (no hay dos tipos de memorias). Y si bien existe un tiempo de miss penalty cuando se quiere traer un bloque de memoria (dado que se copian los datos), la simulación no imita el proceso físico que el hardware realiza. Es por esto que la simulación no es fiel a la hora de medir tiempos.

Lo que sí permite la simulación es poder comparar los miss rates de diferentes arquitecturas que tienen especificaciones diferentes, aunque dicha medición y comparación escape al alcance del trabajo.

6. Anexo - Código fuente

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

#include "Cache.c"
#include "Memory.c"
#include "Command.c"


char* read_line(FILE* input) {
    char* str = (char*) malloc(sizeof(char));
    *str = '\0';
    int len = 0;
    char c;

    if (feof(input)) return NULL;

    while (EOF != (c=fgetc(input)) && c != '\n') {
        ++len;
        str = (char*) realloc(str, sizeof(char)*(len+1));
        str[len-1] = c;
        str[len] = '\0';
    }

    if (c == EOF && len == 0) {
        free(str);
        return NULL;
    }

    return str;
}

int load_commands(FILE* input, Command** ptr) {
    Command* cmds = NULL;
    int count = 0;
    char* line = NULL;

    while (line = read_line(input)) {
        ++count;
        cmds = (Command*) realloc(cmds, sizeof(Command)*count);
        Command_init(&cmds[count-1], line);
        free(line);
    }

    *ptr = cmds;
    return count;
}


int main(int argc, char** argv) {
    if (argc < 2) {
        fprintf(stderr, "No file specified\n");
        return 1;
    }

    init();

    // Get commands
    Command* commands;
    int cmd_count = 0;
```

```
60      FILE* input = fopen(argv[1], "r");
61      if (0) {
62          fprintf(stderr, "Error opening file %s", argv[1]);
63          return 1;
64      }
65      cmd_count = load_commands(input, &commands);
66
67      // Execute commands
68      for (int i = 0; i < cmd_count; ++i) {
69          Command_execute(commands+i);
70      }
71
72      for (int i = 0; i < cmd_count; ++i) {
73          Command_destroy(commands+i);
74      }
75
76      return 0;
77  }
```

src/main.c

```c
#ifndef COMMAND_C
#define COMMAND_C

#include <string.h>

typedef unsigned int uint;
typedef unsigned char uchar;

typedef struct {
    char* op;
    uint* argv;
    int argc;
} Command;


int is_eoarg_char(char c) {
    return c == ' ' || c == '\n' || c == '\t' || c == ',';
}


void Command_init(Command* self, char* cmdstr) {
    int start = 0, end = 0, cmdlen = strlen(cmdstr);

    // Default values
    self->op = NULL;
    self->argv = NULL;
    self->argc = 0;

    // Get the operation
    int eoarg = 0;
    while (end < cmdlen && !eoarg) {
        eoarg = is_eoarg_char(cmdstr[++end]);
    }
    self->op = malloc(end-start+1);
    strncpy(self->op, cmdstr, end-start);
    self->op[end-start] = '\0';

    // Now we get the arguments
    while (end < cmdlen) {
        start = end;
        while (start < cmdlen && eoarg) {
            eoarg = is_eoarg_char(cmdstr[++start]);
        }
        end = start;
        while (end < cmdlen && !eoarg) {
            eoarg = is_eoarg_char(cmdstr[++end]);
        }
        if (start < cmdlen) {
            ++self->argc;
            self->argv = (uint*) realloc(self->argv, sizeof(uint)*self->argc);
            char buffer[16];
            memset(buffer, '\0', 16);
            strncpy(buffer, cmdstr+start, end-start);
            self->argv[self->argc-1] = atoi(buffer);
        }
    }
}

void Command_execute(Command* self) {
    if (strcmp(self->op, "R") == 0) {
        int value = (uchar) read_byte(self->argv[0]);
```

```
62          if (value >= 0)
63              printf("%d\n", value);
64      } else if (strcmp(self->op, "W") == 0) {
65          int res = write_byte(self->argv[0], self->argv[1]);
66          printf("%d\n", res);
67      } else if (strcmp(self->op, "MR") == 0) {
68          printf("MissRate %d%%\n", get_miss_rate());
69      } else {
70          fprintf(stderr, "Unknown command: %s\n", self->op);
71      }
72  }
73
74  void Command_destroy(Command* self) {
75      free(self->op);
76      if (self->argc > 0)
77          free(self->argv);
78  }
79
80
81  #endif
```

src/Command.c

```c
#include "config.h"
#include "Block.c"
#include "Memory.c"

#include "math.h"

#define NO_BLOCKS (CACHE_SIZE/NO_WAYS/BLOCK_SIZE)
#define INDEX_SIZE (log(NO_BLOCKS)/log(2))
#define OFFSET_SIZE (log(BLOCK_SIZE)/log(2))
#define TAG_SIZE ((int)(log(MEMORY_SIZE)/log(2))-INDEX_SIZE-OFFSET_SIZE)


typedef unsigned char uchar;

typedef struct {
    Memory memory;
    Block blocks[NO_WAYS][NO_BLOCKS];
    int access_count;
    int misses_count;
} Cache;

Cache self;


void init() {
    self.access_count = 0;
    self.misses_count = 0;
    for (int w = 0; w < NO_WAYS; ++w) {
        for (int b = 0; b < NO_BLOCKS; ++b) {
            Block_init(&self.blocks[w][b]);
        }
    }
    Memory_init(&self.memory);
}

int get_index(int addr) {
    return (int)(addr / pow(2, OFFSET_SIZE)) % (int)(pow(2, INDEX_SIZE));
}

int get_offset(int addr) {
    return addr % BLOCK_SIZE;
}

int get_tag(int addr) {
    return (int)(addr / pow(2, INDEX_SIZE + OFFSET_SIZE));
}

void read_block(int blocknum) {
    // Doesnt check if block is dirty
    int tag = get_tag(blocknum * BLOCK_SIZE);
    int set = get_index(blocknum * BLOCK_SIZE);
    int way = find_lru(set);
    Block_read(&self.blocks[way][set], &self.memory.blocks[blocknum], tag);
}

Block* cache_address(int addr) {
    int set = find_set(addr);
    int way = find_lru(set);
    if (is_valid(way, set) && is_dirty(way, set)) {
        write_block(way, set);
    }
```

```
62
63      read_block(get_blocknum(addr));
64      return &self.blocks[way][set];
65  }
66
67  int read_byte(int address) {
68      if (address >= MEMORY_SIZE) {
69          fprintf(stderr, "Address %d overflows\n", address);
70          return -1;
71      }
72      int tag = get_tag(address);
73      int offset = get_offset(address);
74      ++self.access_count;
75      int set = find_set(address);
76      for (int way = 0; way < NO_WAYS; ++way) {
77          Block block = self.blocks[way][set];
78          if (block.valid && block.tag == tag) {
79              Block_update_lru(&block);
80              return block.data[offset];
81          }
82      }
83      // Miss in cache
84      ++self.misses_count;
85      Block *b = cache_address(address); // This updates lru
86      return b->data[offset];
87  }
88
89  int find_lru(int setnum) {
90      int minway = 0;
91      for (int w = 0; w < NO_WAYS; ++w) {
92          Block b = self.blocks[w][setnum];
93          if (!b.valid)
94              return w;
95          else if (b.last_used < self.blocks[minway][setnum].last_used)
96              minway = w;
97      }
98      return minway;
99  }
100
101 int is_valid(int way, int set) {
102     return self.blocks[way][set].valid;
103 }
104
105 int is_dirty(int way, int set) {
106     return self.blocks[way][set].bit_D;
107 }
108
109 int get_blocknum(int address) {
110     return (int)(address/BLOCK_SIZE);
111 }
112
113 void write_block(int way, int set) {
114     Block block = self.blocks[way][set];
115     int block_num = (int)(block.tag * pow(2, INDEX_SIZE) + set);
116     Memory_write_block(&self.memory, &block, block_num);
117 }
118
119 int write_byte(int address, char value) {
120     // Write-Allocate politic => update cache to write
121     if (address >= MEMORY_SIZE) {
122         fprintf(stderr, "Address %d overflows\n", address);
123         return -1;
```

```
124     }
125     Block* block = NULL;
126     int tag = get_tag(address);
127     int offset = get_offset(address);
128     int set = find_set(address);
129     int way = 0;
130     ++self.access_count;
131 
132     for (; way < NO_WAYS && !block; ++way) {
133         Block *b = &self.blocks[way][set];
134         if (b->valid && b->tag == tag)
135             block = b;
136     }
137 
138     if (!block) {
139         ++self.misses_count;
140         block = cache_address(address);
141     }
142 
143     block->data[offset] = value;
144     block->bit_D = 1;
145     Block_update_lru(&block);
146     return (unsigned char) value;
147 }
148 
149 int find_set(int addr) {
150     return get_index(addr);
151 }
152 
153 int get_miss_rate() {
154     return (int)(100.0*self.misses_count/self.access_count);
155 }
```

src/Cache.c

```
1 #define CACHE_SIZE (4*1024)
2 #define NO_WAYS 4
3 #define BLOCK_SIZE 64
4 #define MEMORY_SIZE (64*1024)
```

src/config.h

```c
#include "config.h"

int LRU_COUNT = 1;

typedef struct {
    int last_used;
    char valid;
    char data[BLOCK_SIZE];
    int tag;
    char bit_D;
} Block;

void Block_init(Block *self) {
    self->last_used = 0;
    self->tag = -1;
    self->bit_D = 0;
    self->valid = 0;
    for (int i = 0; i < BLOCK_SIZE; ++i) {
        self->data[i] = 0;
    }
}

void Block_update_lru(Block *self) {
    self->last_used = LRU_COUNT++;
}

void Block_read(Block *self, Block *src, int tag) {
    Block_init(self);
    self->valid = 1;
    Block_update_lru(self);
    self->tag = tag;
    memcpy(self->data, src->data, BLOCK_SIZE);
}
```

src/Block.c

```c
#ifndef MEMORY_C
#define MEMORY_C

#include "config.h"

#define BLOCKS_COUNT MEMORY_SIZE/BLOCK_SIZE

typedef struct {
    Block blocks[BLOCKS_COUNT];
} Memory;

void Memory_init(Memory *self) {
    for (int b = 0; b < BLOCKS_COUNT; ++b) {
        Block_init(&self->blocks[b]);
    }
}

void Memory_write_block(Memory *self, Block *src, int blockNo) {
    Block *memblock = self->blocks + blockNo;
    memcpy(memblock->data, src->data, BLOCK_SIZE);
}

#endif
```

src/Memory.c