Memorias caché

66:20 Organización de Computadoras

Trabajo práctico 2

Axel Lijdens (95772)
Eduardo R Madariaga (90824)
Mauro Toscano (96890)

Univesidad de Buenos Aires - FIUBA

Índice

Objetivos

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

Alcance

Este trabajo práctico es de elaboración grupal, evaluación individual, y de carácter obligatorio para todos alumnos del curso.

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.

Recursos

Este trabajo práctico debe ser implementado en C, y correr al menos en Linux.

Fecha de entrega

La última fecha de entrega y presentación será el jueves 17 de mayo de 2018.

Introducción

La memoria a simular es una caché [1] asociativa por conjuntos de dos vías, de 1KB de capacidad, bloques de 32 bytes, política de reemplazo LRU y política de escritura WT/$\sim$ WA. Se asume que el espacio de direcciones es de 12 bits, y hay entonces una memoria principal a simular con un tamaño de 4KB. Estas memorias pueden ser implementadas como variables globales. Cada bloque de la memoria caché deberá contar con su metadata, incluyendo el bit V y el tag.

Implementación del programa

Para el diseño del programa se empezó por establecer los parámetros de entrada necesarios:

- Un archivo de entrada

Teniendo en cuenta lo propuesto se escribió el siguiente mensaje de ayuda:

```
1 Uso: ./tp [archivo]
```

Listing 1: Mensaje de ayuda del programa

Asimismo, se debieron tener en cuenta y validar los siguientes casos en donde algún factor es incorrecto y el mensaje devuelto por el probrama debe ser explicativo del error:

- **No se pudo abrir algun archivo:** Este error indica que el archivo indicado por parámetro de entrada no pudo ser abierto.
- **Error de formato:** El archivo de entrada no respetaba el formato especificado en el enunciado.

Una vez que el archivo se abrió exitosamente, se procede a parsear y ejecutar las acciones en forma secuencial. El programa imprime un detalle de los accesos a la caché simulada con el siguiente formato:

- *WM:* Indica un Write-Miss en la caché.
- *WH[n][m][o]:* Indica un Write-Hit en la vía n, bloque m y offset o.
- *RM[n][m]:* Indica un Read-Miss y una consecuente carga del bloque m en la vía n.
- *RH[n][m][o]:* Indica un Read-Hit en la caché en la vía n, bloque m y offset o.
- *MR:* Corresponde al Miss rate calculado.

Además, las operaciones de lectura indican el valor leído (desde la caché en caso de un RH o desde memoria en caso de un RM).

Como solamente hay 2 vías, fue posible implementar el reemplazo LRU utilizando solamente 1 bit, el cuál indica el estado "nuevo"si está en 1 y el estado "viejo"si está en cero. De esta forma no resultó necesario implementar un algoritmo complejo para llevar la cuenta de uso de cada bloque.

Debido a que los bloques son de 32 bytes, se requieren 5 bits para el offset. La caché tiene un tamaño de 1024 bytes, cada vía tiene 512 bytes. Los bloques son de 32 bytes y por lo tanto hay 16 grupos, por lo que se necesitan 4 bits para el índice.

Para direccionar la memoria principal se requieren 12 bits en total, por lo tanto los 3 bits restantes se utilizan para el tag.

tag (3 bits)	index (4 bits)	offset (5 bits)

Cuadro 1: Dirección de memoria

tag (3 bits)	V (1 bit)	LRU (1 bit)	datos (32 bytes)

Cuadro 2: Bloque en la caché

Compilación del programa

Para compilar el programa basta ejecutar la siguiente línea:

make

Corridas de prueba

A continuación se muestran los resultados de las ejecuciones del programa para cada archivo de pruebas:

```
1  W 0, 16
2  R 0
3  R 1024
4  R 8
5  R 2050
6  R 3074
7  W 8, 12
8  R 8
9  R 8
10 W 3072, 255
11 W 2048, 10
12 R 0
13 MR
```

Listing 2: prueba1.mem

```
1  WM
2  RM[0][0]     -> 16
3  RM[1][0]     -> 0
4  RH[0][0][8]  -> 0
5  RM[1][0]     -> 0
6  RM[0][0]     -> 0
7  WM
8  RM[1][0]     -> 12
9  RH[1][0][8]  -> 12
10 WH[0][0][0]=255
```

```
WM
RH[1][0][0]  -> 16
MR=67
```

Listing 3: Resultado prueba1.mem

```
R 0
R 31
W 32, 10
R 32
W 32, 20
R 32
R 1040
R 2064
R 32
R 32
MR
```

Listing 4: prueba2.mem

```
RM[0][0]     -> 0
RH[0][0][31]  -> 0
WM
RM[0][1]     -> 10
WH[0][1][0]=20
RH[0][1][0]  -> 20
RM[1][0]     -> 0
RM[0][0]     -> 0
RH[0][1][0]  -> 20
RH[0][1][0]  -> 20
MR=50
```

Listing 5: Resultado prueba2.mem

```
W 0, 1
W 1, 2
W 2, 3
W 3, 4
W 4, 5
R 0
R 1
R 2
R 3
R 4
MR
```

Listing 6: prueba3.mem

```
WM
WM
WM
WM
WM
RM[0][0]     -> 1
RH[0][0][1]  -> 2
RH[0][0][2]  -> 3
RH[0][0][3]  -> 4
RH[0][0][4]  -> 5
MR=60
```

Listing 7: Resultado prueba3.mem

```
W 0, 1
W 1, 2
W 2, 3
W 3, 4
W 4, 5
R 0
R 1
R 2
R 3
R 4
R 1024
R 2048
R 0
R 1
R 2
R 3
R 4
MR
```

Listing 8: prueba4.mem

```
WM
WM
WM
WM
WM
RM[0][0]     -> 1
RH[0][0][1]  -> 2
RH[0][0][2]  -> 3
RH[0][0][3]  -> 4
RH[0][0][4]  -> 5
RM[1][0]     -> 0
```

```
RM[0][0]     -> 0
RM[1][0]     -> 1
RH[1][0][1]  -> 2
RH[1][0][2]  -> 3
RH[1][0][3]  -> 4
RH[1][0][4]  -> 5
MR=53
```

Listing 9: Resultado prueba4.mem

```
R 0
R 1024
R 3072
R 2048
R 0
R 3072
MR
```

Listing 10: prueba5.mem

```
RM[0][0]     -> 0
RM[1][0]     -> 0
RM[0][0]     -> 0
RM[1][0]     -> 0
RM[0][0]     -> 0
RM[1][0]     -> 0
MR=100
```

Listing 11: Resultado prueba5.mem

Conclusiones

La implementación de la simulación nos premitió estudiar mas en detalle el funcionamiento de una caché de 2 vías. Las cachés del estilo WT/ WA tienen la ventaja de ser simples de implementar, ya que no hace falta considerar el estado "dirty", porque los datos siempre se escriben a la memoria principal. La desventaja es que no hay mejora de velocidad en las escrituras (se comportan de la misma forma que si no existiera la caché).

Problemas encontrados a lo largo del proyecto

Es dificil llevar la cuenta de los bloques actualizados/reemplazados por las lecturas, por lo que desarrollar una salida detallada fue de gran utilidad para detectar errores de programación.

Códigos fuente

Repositorio: https://github.com/MauroFab/orga-6620-tp2

MakeFile

```
1
2  tp: main.c cache.c cache.h
3    @gcc -ggdb -std=c99 -Wall -Wpedantic -Werror main.c
         cache.c -o tp -lm
4    @echo "LD tp"
5
6  clean:
7    @rm -f tp
8
9  .PHONY: clean
```

Listing 12: makefile

Main

```
1  #include <stdbool.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include "cache.h"
6
7  #define MEM_SIZE 4096  // 4 KiB
8
9  char memory[MEM_SIZE];
10
11 /*
12  * Reinicia la tasa miss e invalida la cache
13  */
14 void init() {
15   cache_init();
16 }
17
18 /*
19  * Si el adress esta en la cache devuelve su valor
20  * Si no lo carga de memory y devuelve -1
21  */
22 static char read_byte(int address) {
23   if (address >= MEM_SIZE) {
24     return -1;
```

```
25    }
26    char rv = cache_read_byte(address, memory);
27    if (rv == -1) {
28      return memory[address];
29    }
30    return rv;
31  }
32
33  static int write_byte(int address, unsigned char
       value) {
34    if (address >= MEM_SIZE) {
35      return -1;
36    }
37    return cache_write_byte(address, value, memory);
38  }
39
40  static bool _process_file(FILE *file) {
41    while (!feof(file)) {
42      unsigned int address;
43      unsigned int value;
44      char command = '\0';
45
46      if (fscanf(file, "%c", &command) != 1) {
47        /* verifica que se trate el fin de archivo */
48        return (feof(file) > 0);
49      }
50
51      switch (command) {
52        case 'R':
53          if (fscanf(file, "%u", &address) != 1) {
54            return false;
55          }
56          printf("%d\n", read_byte(address));
57          break;
58
59        case 'W':
60          if (fscanf(file, "%u, %u", &address, &value)
                != 2) {
61            return false;
62          }
63          write_byte(address, value);
64          break;
65
66        case 'M':
67          if (fscanf(file, "%c", &command) != 1) {
```

```
68            return false;
69          }
70          if (command != 'R') {
71            return false;
72          }
73
74          printf("MR=%d\n", cache_get_miss_rate());
75          break;
76
77        default:
78          return false;
79      }
80
81      /* lee el fin de linea */
82      if (fread(&value, 1, 1, file) == 0) {
83        break;
84      }
85    }
86
87    return true;
88  }
89
90  int main(int argc, const char *argv[]) {
91    if (argc != 2) {
92      printf("Uso: %s [archivo]\n", argv[0]);
93      return EXIT_FAILURE;
94    }
95
96    const char *filename = argv[1];
97    FILE *file = fopen(filename, "r");
98    if (file == NULL) {
99      printf("No se pudo abrir el archivo.\n");
100     return EXIT_FAILURE;
101   }
102
103   init();
104
105   if (!_process_file(file)) {
106     goto error;
107   }
108
109   fclose(file);
110   return EXIT_SUCCESS;
111
112 error:
```

```
113   fclose(file);
114   printf("Error de formato.\n");
115   return EXIT_FAILURE;
116 }
```

Listing 13: makefile

```
1
2  #ifndef CACHE_H_
3  #define CACHE_H_
4
5  #include <stdint.h>
6
7  /**
8   * bits: 31              11    9        5         0
9   *       ---------------------------------------
10  *       |               | tag | index | offset |
11  *       ---------------------------------------
12  */
13
14 /** Tamaño total de la cache */
15 #define CACHE_SIZE 1024
16 /** Tamaño de cada bloque de la cache */
17 #define CACHE_BS 32
18 /** Número de vías. */
19 #define CACHE_WAYS_NUM 2
20
21 /** Tamaño de cada vía. */
22 #define CACHE_WAY_SIZE (CACHE_SIZE / CACHE_WAYS_NUM)
23 #define CACHE_BLOCKS_PER_WAY ((CACHE_SIZE /
       CACHE_WAYS_NUM) / CACHE_BS)
24
25 void cache_init();
26 char cache_read_byte(int address, const char *memory)
       ;
27 int cache_write_byte(int address, unsigned char value
       , char *memory);
28 int cache_get_miss_rate();
29
30 #endif
```

Listing 14: makefile

```
1
2  #include "cache.h"
3  #include <math.h>
```

```
4  #include <stdint.h>
5  #include <string.h>
6
7  #define ENABLE_LOG 1
8
9  #if ENABLE_LOG
10 #include <stdio.h>
11 #define LOG(...) (printf(__VA_ARGS__))
12 #else
13 #define LOG(...) \
14   do {           \
15   } while (0)
16 #endif
17
18 #define BLOCK(way, index) (cache.entries[way][index])
19
20 /** Marca la entrada del cache como válida. */
21 #define SET_VALID(way, index) (BLOCK(way, index).
       is_valid = 1)
22 /** Devuelve "true" si el cache está marcado como vá
       lido. */
23 #define IS_VALID(way, index) (BLOCK(way, index).
       is_valid == 1)
24
25 /** Marca la entrada del cache como nueva (LRU). */
26 #define SET_NEW(way, index) (BLOCK(way, index).is_new
       = 1)
27 /** Marca la entrada del cache como vieja (LRU). */
28 #define SET_OLD(way, index) (BLOCK(way, index).is_new
       = 0)
29 /** Retorns "true" si la entrada es vieja. */
30 #define IS_OLD(way, index) (BLOCK(way, index).is_new
       == 0)
31
32 /** Obtiene el offset desde la dirección de memoria.
       */
33 #define OFFSET(address) (address & 0x1F)
34 /** Obtiene el index desde la dirección de memoria.
       */
35 #define INDEX(address) ((address >> 5) & 0x0F)
36 /** Obtiene el tag desde la dirección de memoria. */
37 #define TAG(address) ((address >> 9) & 0x07)
38
39 /** Bloque de datos en el cache. */
40 typedef uint8_t cache_block_t[CACHE_BS];
```

```
41
42 /** Entrada en el cache. */
43 typedef struct {
44   uint8_t tag : 3;
45   uint8_t is_valid : 1;
46   uint8_t is_new : 1;
47   cache_block_t data;
48 } cache_entry_t;
49
50 /** Cache. */
51 typedef struct {
52   cache_entry_t entries[CACHE_WAYS_NUM][
        CACHE_BLOCKS_PER_WAY];
53   double miss_count;
54   double hit_count;
55 } cache_t;
56
57 /** Instancia global del cache. */
58 static cache_t cache;
59
60 /**
61  * @brief Carga un bloque de memoria principal a a la
        cache.
62  *
63  * @param index El indice del conjunto de la cache.
64  * @param tag El tag del bloque.
65  * @param mem Memoria principal.
66  * @return La vía en la cual se cargó el bloque.
67  */
68 static void _load_block(int index, int tag, const
      char *mem) {
69   int way = 0;
70   if (IS_OLD(0, index)) {
71     way = 0;
72   } else {
73     way = 1;
74   }
75
76   cache_entry_t *entry = &cache.entries[way][index];
77
78   /* copia los datos y actualiza los metadatos */
79   memcpy(entry->data, mem, CACHE_BS);
80
81   SET_NEW(way, index);
82   SET_OLD(!way, index);
```

```
83    SET_VALID(way, index);
84    entry->tag = tag;
85
86    LOG("RM[%d][%d]    -> ", way, index);
87  }
88
89  /**
90   * @brief Lee un byte del cache y actualiza los
        metadatos.
91   *
92   * @param way Via a leer.
93   * @param index Indice del groupo.
94   * @param offset Offset del bloque.
95   * @return char Dato.
96   */
97  static char _read_cache(int way, int index, int
       offset) {
98    cache.entries[way][index].is_new = 1;
99    cache.entries[!way][index].is_new = 0;
100   char value = cache.entries[way][index].data[offset
        ];
101
102   LOG("RH[%d][%d][%d] -> ", way, index, offset);
103   return value;
104 }
105
106 /**
107  * @brief Inicializa el cache.
108  *
109  */
110 void cache_init() {
111   memset(&cache, 0, sizeof(cache_t));
112 }
113
114 /**
115  * @brief Lee un byte del cache.
116  *
117  * @param address Dirección del byte a leer.
118  * @param memoria Memoria.
119  * @return unsigned char Byte leído.
120  */
121 char cache_read_byte(int address, const char *memory)
       {
122   int index = INDEX(address);
123   int offset = OFFSET(address);
```

```
124
125   if (IS_VALID(0, index) && TAG(address) == cache.
          entries[0][index].tag) {
126     cache.hit_count++;
127     return _read_cache(0, index, offset);
128   } else if (IS_VALID(1, index) && TAG(address) ==
          cache.entries[1][index].tag) {
129     cache.hit_count++;
130     return _read_cache(1, index, offset);
131   } else {
132     cache.miss_count++;
133     _load_block(index, TAG(address), memory);
134     return -1;
135   }
136 }
137
138 /**
139  * @brief Escribe en la memoria y si en necesario ne
          el cache.
140  *
141  * @param address Dirección de memoria en la que se
          escribe.
142  * @param value Valor a escribir.
143  * @param memory Memoria principal.
144  * @return int 0 si fue un hit, -1 si fue un miss.
145  */
146 int cache_write_byte(int address, unsigned char value
       , char *memory) {
147   int index = INDEX(address);
148   int offset = OFFSET(address);
149
150   int rv = 0;
151
152   if (IS_VALID(0, index) && TAG(address) == cache.
          entries[0][index].tag) {
153     cache.hit_count++;
154     *(cache.entries[0][index].data + offset) = value;
155     cache.entries[0][index].is_new = 1;
156     cache.entries[1][index].is_new = 0;
157     LOG("WH[%d][%d][%d]=%d\n", 0, index, offset,
            value);
158   } else if (IS_VALID(1, index) && TAG(address) ==
          cache.entries[1][index].tag) {
159     cache.hit_count++;
160     *(cache.entries[1][index].data + offset) = value;
```

```
161     cache.entries[1][index].is_new = 1;
162     cache.entries[0][index].is_new = 0;
163     LOG("WH[%d][%d][%d]=%d\n", 1, index, offset,
            value);
164   } else {
165     cache.miss_count++;
166     LOG("WM\n");
167     rv = -1;
168   }
169
170   /* WT: siempre escribe a memoria */
171   *(memory + address) = value;
172   return rv;
173 }
174
175 /**
176  * @brief Devuelve el miss rate redondeado al entero
        mas cercano.
177  *
178  * @return uint64_t miss rate.
179  */
180 int cache_get_miss_rate() {
181   return round((cache.miss_count * 100) / (cache.
        miss_count + cache.hit_count));
182 }
```

Listing 15: makefile