Organización de Computadoras - TP 1

Martín Cohen, Padron: 100812 martincohen98@gmail.com

1er Cuatrimestre 2020

Resumen

En este trabajo practico se programó una simulación de una memoria cache en C, con política de reemplazo LRU (Least Recently Used) y política de escritura WT/ ¬ WA (Write Through, No Write Allocate).

1 Introducción

El trabajo practico tiene como objetivo entender el funcionamiento de la memoria cache de forma practica, y poder medir el miss rate de una memoria cache para un programa. Para esto fue necesario crear una simulación de una memoria cache.

El cache pedido debe cumplir con las siguientes características:

- Ser un cache de un solo nivel.
- Tener una política de reemplazo LRU.
- Tener una política de escritura WT/ ¬ WA.
- Ser asociativa por conjunto de cuatro vías.
- Tener 4kB de capacidad.
- Tener bloques de memoria de 128 bytes.

Para esto también fue necesario simular una memoria principal que tuviera 64kB de capacidad para utilizar con la memoria cache. El programa de la simulación fue programado en su totalidad en el lenguaje de programación C.

2 Diseño e Implementación

2.1 Análisis de la Linea de Comandos

En la linea de comandos los parámetros deben ser enviados de la siguiente manera:

```
./cachecheck inputfile
```

Siendo inputfile el archivo donde se tienen la lista de escrituras y lecturas a realizar en el formato detallado en la consigna.

De no existir el archivo indicado o se ingresen parámetros incorrectos, se notifica por linea de comandos.

2.2 Desarrollo del Código Fuente

Todas las cantidades indicadas en el enunciado fueron definidas como constantes para evitar el hardcodeo de los números. Debido a esto las cantidades no van a ser utilizadas en esta explicación, ya que son en su mayoría independientes a la implementación.

Debido a la simpleza del problema, las variables de la memoria, la memoria cache, la cuenta, y los misses son globales para simplificar el código.

En la consigna se pedía implementar las siguientes funciones para el cache. El funcionamiento de estas funciones esta detallado en la consigna.

```
void init()
unsigned int get_offset (unsigned int address)
unsigned int find_set(unsigned int address)
unsigned int select_oldest(unsigned int setnum)
int compare_tag (unsigned int tag, unsigned int set)
void read_tocache(unsigned int blocknum, unsigned int way,
unsigned int set)
void write_tocache(unsigned int address, unsigned char value)
unsigned char read_byte(unsigned int address)
void write_byte(unsigned int address, unsigned char value)
float get_miss_rate()
```

A estas funciones agregue algunas funciones extra para evitar hacer la misma operacion varias veces y modularizar.

```
unsigned int get_tag(unsigned int address);
void sumar_tiempo();
```

La función get_tag devuelve el tag que utiliza el cache para identificar bloques del bloque que contiene a la dirección dada. Lo único que hace sumar_tiempo es aumentar el contador de todos los bloques en el cache. Esto es necesario para implementar la política de reemplazo.

La memoria principal es representada simplemente por un arreglo de tipo unsigned char (ya que ocupan un byte cada uno) del tamaño indicado de 64kB.

Para la implementación de la memoria cache tuve que crear estructuras de datos más especializadas.

Cada bloque de cache fue representado por una instancia de la struct BloqueCache. Este esta compuesto por un bit de validez, un tag para identificar el bloque, el tiempo desde la ultima lectura del bloque, y un arreglo que contiene una copia del bloque en si. El tiempo desde la ultima lectura se representa con un contador que se le suma uno cada vez que se lee un dato pero que vuelve a 0 cada vez que se lee un dato en el bloque.

Cada una de las vías también es una estructura de datos que contiene la cantidad de bloques de cache indicados. Luego el cache en si es un arreglo de las vías del cache.

Cada vez que se realiza una lectura se incrementa la variable cuenta, y si la lectura es un miss también se incrementa la variable misses. En las escrituras no se modifican ninguna de estas, ya que debido a la política de escritura siempre se accede a la memoria principal. Para calcular el porcentaje de misses se divide la cantidad de misses por la cantidad de lecturas.

Si se intenta escribir un dato mayor a lo que entra en un byte, se escribe 0 en lugar del dato indicado.

3 Proceso de Compilación

El proceso de compilación es simple. Lo único que se debe hacer es llamar a gcc con el archivo main.c de la siguiente manera:

```
gcc main.c -o cachecheck
```

Con esto ya se tiene el ejecutable listo para correr el programa.

Es importante notar que para ser debuggeado el programa debe ser compilado de una manera distinta. Se le deben agregar los flags siguientes a ambas versiones.

```
gcc main.c -g -O0 -o cachecheck
```

Estos flags mantienen la información de debuggeo al compilar y me aseguran que no se va a optimizar el código de ninguna manera respectivamente.

4 Casos de Prueba

Las ejecuciones de las pruebas pedidas en el enunciado con los archivos indicados:

4.1 Prueba 1

```
W 0, 255
W 1024, 254
W 2048, 248
W 4096, 096
W 8192, 192
R 0             Byte leido en la direccion 0: 255
R 1024          Byte leido en la direccion 1024: 254
R 2048          Byte leido en la direccion 2048: 248
R 8192          Byte leido en la direccion 8192: 192
MR              Miss rate: 1.000000
```

4.2 Prueba 2

```
R 0
R 127
W 128, 10       Byte leido en la direccion 0: 0
R 128           Byte leido en la direccion 127: 0
W 128, 20       Byte leido en la direccion 128: 10
R 128           Byte leido en la direccion 128: 20
MR              Miss rate: 0.500000
```

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
```

```
Byte leido en la direccion 1152: 0
Byte leido en la direccion 2176: 0
Byte leido en la direccion 3200: 0
Byte leido en la direccion 4224: 0
Byte leido en la direccion 128: 1
Byte leido en la direccion 129: 2
Byte leido en la direccion 130: 3
Byte leido en la direccion 131: 4
Miss rate: 0.625000
```

4.4 Prueba 4

```
W 0, 255
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 2048
R 1024
R 0
R 1
R 2
R 3
R 4
MR
```

```
Byte leido en la direccion 0: 255
Byte leido en la direccion 1: 2
Byte leido en la direccion 2: 3
Byte leido en la direccion 3: 4
Byte leido en la direccion 4: 5
Byte leido en la direccion 4096: 0
Byte leido en la direccion 8192: 0
Byte leido en la direccion 2048: 0
Byte leido en la direccion 1024: 0
Byte leido en la direccion 0: 255
Byte leido en la direccion 1: 2
Byte leido en la direccion 2: 3
Byte leido en la direccion 3: 4
Byte leido en la direccion 4: 5
Miss rate: 0.357143
```

4.5 Prueba 5

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

```
Byte leido en la direccion 4096: 0
Byte leido en la direccion 8192: 0
Byte leido en la direccion 4096: 0
Byte leido en la direccion 0: 0
Byte leido en la direccion 4096: 0
Miss rate: 0.600000
```

4.6 Prueba 6

```
W 0, 1
W 1024, 2
W 2048, 3
W 4096, 4
W 8192, 256
R 0
R 1024
R 2048
R 4096
R 8192
MR
```

```
Byte leido en la direccion 0: 1
Byte leido en la direccion 1024: 2
Byte leido en la direccion 2048: 3
Byte leido en la direccion 4096: 4
Byte leido en la direccion 8192: 0
Miss rate: 1.000000
```

5 Conclusiones

El trabajo practico tuvo más efecto que solo familiarizarse con el Assembly de MIPS. También tuvo una función de acercarse más a la programación de bajo nivel y la fuerte dependencia de esta al hardware. Se puede notar que el lenguaje C tiene casi una traducción directa a Assembly, con la diferencia que el lenguaje C se puede escribir de la misma manera sin importar la arquitectura de la computadora en la que va a ser ejecutado.

El trabajo me llevo a descubrir que gcc en una computadora con una arquitectura no solo traduce el código C de manera distinta que en una computadora con otra arquitectura (cosa que es trivial si se sabe el el código Assembly es completamente distinto), sino que no **puede** compilar el código para otra arquitectura. Esto significa que ambas versiones de gcc tienen código de traducción que la otra no tiene. Esto significa que son programas diferentes con el mismo nombre y propósito.

Una característica del ABI de MIPS que me pareció extraña es que las funciones deben guardar sus propios argumentos en memoria cuando se prepara el stack frame. No encuentro el propósito ya que los argumentos solo son útiles para la función misma y otras funciones no los necesitan. La única razón que se me ocurre es a la hora de hacer un backtrace durante un debuggeo o un crash-dump se pueda ver los parámetros con los que se llamo la función problemática.

6 Codigo

Codigo del archivo main.c:

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include <stdlib.h>
#define MEMSIZE 65536 //2 a la 16
#define CACHESIZE 32 //32 bloques en el cache
#define BLOCKSIZE 128
#define SETSNUM 4
#define SETSIZE CACHESIZE / SETSNUM
#define BUFLEN 25

void init();
unsigned int get_offset(unsigned int address);
unsigned int find_set(unsigned int address);
unsigned int select_oldest(unsigned int setnum);
int compare_tag (unsigned int tag, unsigned int set);
void read_tocache(unsigned int blocknum, unsigned int way,
unsigned int set);
void write_tocache(unsigned int address, unsigned char value);
unsigned char read_byte(unsigned int address);
void write_byte(unsigned int address, unsigned char value);
float get_miss_rate();

unsigned int get_tag(unsigned int address);
void sumar_tiempo();
void interpretar_archivo(FILE* archivo);


struct BloqueCache {
bool valido;
unsigned char bloque[BLOCKSIZE];
unsigned int tiempoDesdeLectura;
unsigned int tag;
};
```

```
struct Conjunto {
struct BloqueCache via[SETSIZE];
};

unsigned char memoria[MEMSIZE];
struct Conjunto cache[SETSNUM];
unsigned int misses;
unsigned int cuenta;

int main(int argc, char** argv) {
bool argsOk = false;
int retorno = 0;

    if(argc == 2) {
     argsOk = true;
    }

    if(!argsOk) {
     fprintf(stderr, "Error en entrada de parametros,
     ingresar un archivo.\n");
     retorno = -1;
    }

FILE* archivo;
archivo = fopen(argv[1], "r");
if (archivo == NULL) {
fprintf(stderr, "Error al arbir archivo, checkear nombre
de archivo\n");
return -2;
}

interpretar_archivo(archivo);

fclose(archivo);
    printf("\n");
    return retorno;
}


void interpretar_archivo(FILE* archivo) {
init();

int resultadoLectura;
char buffer[BUFLEN];
resultadoLectura = fscanf(archivo, "%s", buffer);
```

```
while (resultadoLectura != EOF) {
if (strcmp(buffer, "FLUSH") == 0)
init();
if (strcmp(buffer, "MR") == 0)
printf("Miss rate: %f", get_miss_rate());
if (strcmp(buffer, "R") == 0) {
unsigned int address;
resultadoLectura = fscanf(archivo, "%u", &address);
if (address < MEMSIZE)
printf("Byte leido en la direccion %u: %u\n",
address, read_byte(address));
}
if (strcmp(buffer, "W") == 0) {
unsigned int address;
unsigned char value;
resultadoLectura = fscanf(archivo, "%u, %hhu", &address, &value);
if (address < MEMSIZE)
write_byte(address, value);
}
resultadoLectura = fscanf(archivo, "%s", buffer);
}
}


void init() {
memset(memoria, 0, sizeof(memoria));
for (int i = 0; i < SETSNUM; i++) {
for (int j = 0; j < (SETSIZE); j++) {
cache[i].via[j].valido = false;
}
}
misses = 0;
}


unsigned int get_offset(unsigned int address) {
unsigned int offset = address % BLOCKSIZE;
return offset;
}


unsigned int find_set(unsigned int address) {
unsigned int bloque = address / BLOCKSIZE;
unsigned int set = bloque % SETSNUM;
return set;
}
```

```
unsigned int select_oldest(unsigned int setnum) {
unsigned int oldest = 0;
for (unsigned int i = 0; i < SETSIZE; i++) {
if (!cache[setnum].via[i].valido) {
oldest = i;
} else {
if (cache[setnum].via[oldest].valido &&
(cache[setnum].via[i].tiempoDesdeLectura >
cache[setnum].via[oldest].tiempoDesdeLectura)) {
oldest = i;
}
}
}
return oldest;
}


int compare_tag(unsigned int tag, unsigned int set) {
int match = -1;
for (unsigned int i = 0; i < SETSIZE; i++) {
if ((cache[set].via[i].tag == tag) && (cache[set].via[i].valido))
match = i;
}
return match;
}


void read_tocache(unsigned int blocknum, unsigned int way,
unsigned int set) {
unsigned int direccion_bloque = blocknum * BLOCKSIZE;
memcpy(cache[set].via[way].bloque ,
&memoria[direccion_bloque], BLOCKSIZE);
cache[set].via[way].tag = blocknum / SETSNUM;
cache[set].via[way].tiempoDesdeLectura = 0;
cache[set].via[way].valido = true;
}


void write_tocache(unsigned int address, unsigned char value) {
unsigned int set = find_set(address);
unsigned int tag = get_tag(address);
for (int i = 0; i < SETSIZE; i++) {
if (cache[set].via[i].valido && (cache[set].via[i].tag == tag)) {
unsigned int offset = get_offset(address);
```

```
cache[set].via[i].bloque[offset] = value;
}
}
}


unsigned char read_byte(unsigned int address) {
unsigned int offset = get_offset(address);
unsigned int set = find_set(address);
unsigned int tag = get_tag(address);
int way = compare_tag(tag, set);
if (way < 0) {
way = select_oldest(set);
read_tocache(address / BLOCKSIZE, way, set);
misses++;
}
sumar_tiempo();
cache[set].via[way].tiempoDesdeLectura = 0;
return cache[set].via[way].bloque[offset];
}


void write_byte(unsigned int address, unsigned char value) {
write_tocache(address, value);
memoria[address] = value;
}


float get_miss_rate() {
return ((float) misses / (float) cuenta);
}


void sumar_tiempo() {
for (int i = 0; i < SETSNUM; i++) {
for (int j = 0; j < SETSIZE; j++) {
cache[i].via[j].tiempoDesdeLectura++;
}
}
cuenta++;
}


unsigned int get_tag(unsigned int address) {
return (address / BLOCKSIZE) / SETSNUM;
}
```