Cache Implementation Technical report

2271064 Han-Sarang

Code Analysis

1. main.c

#include <stdio.h>

#include "cache\_impl.h"

*// Variables to track cache and memory access metrics*

int num\_cache\_hits = 0; *// Number of cache hits*

int num\_cache\_misses = 0; *// Number of cache misses*

int num\_bytes = 0; *// Number of accessed bytes*

int num\_access\_cycles = 0; *// Number of clock cycles*

int global\_timestamp = 0; *// Number of data access trials*

cache\_entry\_t cache\_array[CACHE\_SET\_SIZE][DEFAULT\_CACHE\_ASSOC]; *// Cache array*

*// Function to retrieve data from cache or memory based on the address and data type*

int retrieve\_data(void \**addr*, char *data\_type*) {

int value\_returned = -1; *// Accessed data*

int result = check\_cache\_data\_hit(*addr*, *data\_type*); *// Check if data exists in cache*

if (result == -1) {

*// Cache miss: Retrieve data from memory*

value\_returned = access\_memory(*addr*, *data\_type*);

} else {

*// Cache hit: Return the data from the cache*

cache\_entry\_t \*cache\_entry = &cache\_array[result / DEFAULT\_CACHE\_ASSOC][result % DEFAULT\_CACHE\_ASSOC];

*// Update based on the data type*

switch (*data\_type*) {

case 'b':

value\_returned = cache\_entry->data[((int)*addr*) % DEFAULT\_CACHE\_BLOCK\_SIZE\_BYTE];

break;

case 'h':

value\_returned = (cache\_entry->data[((int)*addr*) % DEFAULT\_CACHE\_BLOCK\_SIZE\_BYTE + 1] << 8) |

cache\_entry->data[((int)*addr*) % DEFAULT\_CACHE\_BLOCK\_SIZE\_BYTE];

break;

case 'w':

value\_returned = (cache\_entry->data[((int)*addr*) % DEFAULT\_CACHE\_BLOCK\_SIZE\_BYTE + 3] << 24) |

(cache\_entry->data[((int)*addr*) % DEFAULT\_CACHE\_BLOCK\_SIZE\_BYTE + 2] << 16) |

(cache\_entry->data[((int)*addr*) % DEFAULT\_CACHE\_BLOCK\_SIZE\_BYTE + 1] << 8) |

cache\_entry->data[((int)*addr*) % DEFAULT\_CACHE\_BLOCK\_SIZE\_BYTE];

break;

default:

break;

}

}

*// Increase the number of accessed bytes*

num\_bytes += (*data\_type* == 'b') ? 1 : (*data\_type* == 'h') ? 2 : 4;

return value\_returned;

}

*// Main function*

int main(void) {

FILE \*ifp = NULL, \*ofp = NULL;

unsigned long int access\_addr; *// Byte address from "access\_input.txt"*

char access\_type; *// 'b'(byte), 'h'(halfword), or 'w'(word) from "access\_input.txt"*

*// Initialize memory and cache content*

init\_memory\_content();

init\_cache\_content();

*// File handling*

ifp = fopen("access\_input.txt", "r");

if (ifp == NULL) {

printf("Can't open input file\n");

return -1;

}

ofp = fopen("access\_output.txt", "w");

if (ofp == NULL) {

printf("Can't open output file\n");

fclose(ifp);

return -1;

}

*/\* Read each line and retrieve data based on the (address, type) pair \*/*

fprintf(ofp, "[Access Data] \n");

while (fscanf(ifp, "%lu %c", &access\_addr, &access\_type) != EOF) {

int accessed\_data = retrieve\_data((void \*)access\_addr, access\_type); *// Search for data*

fprintf(ofp, "%lu %c %#x\n", access\_addr, access\_type, accessed\_data); *// Print the result*

}

*// Calculate hit ratio and bandwidth*

float hit\_ratio = (float)num\_cache\_hits / (num\_cache\_hits + num\_cache\_misses);

float bandwidth = (float)num\_bytes / num\_access\_cycles;

fprintf(ofp, "----------------------------------------------\n");

*// Print cache performance based on association size*

switch (DEFAULT\_CACHE\_ASSOC) {

case 1:

fprintf(ofp, "[Direct mapped cache performance]\n");

break;

case 2:

fprintf(ofp, "[2-way set associative cache performance]\n");

break;

case 4:

fprintf(ofp, "[Fully associative cache performance]\n");

break;

default:

fprintf(ofp, "[Unknown cache performance]\n");

break;

}

*// Output hit ratio and bandwidth*

fprintf(ofp, "Hit ratio = %.2f (%d/%d)\n", hit\_ratio, num\_cache\_hits, num\_cache\_hits + num\_cache\_misses);

fprintf(ofp, "Bandwidth = %.2f (%d/%d)\n", bandwidth, num\_bytes, num\_access\_cycles);

fclose(ifp);

fclose(ofp);

*// Print final cache entries*

print\_cache\_entries();

return 0;

}

This C code constitutes a cache simulator that reads input from a file ("access\_input.txt"), processes memory accesses, and records the results in an output file ("access\_output.txt"). It initializes memory and a cache, then iterates through the input file, retrieving data from the cache or memory based on the given address and data type.

The `retrieve\_data()` function orchestrates the data retrieval process. It first checks the cache for the requested data. Upon a cache miss, it accesses memory using the `access\_memory()` function, copying the retrieved data into the cache. The function returns the requested data, updating the number of accessed bytes as per the data type.

The `main()` function serves as the program's entry point. It handles file input/output operations, reads the input file line by line, and uses

`retrieve\_data()` to fetch data based on the address and type. The retrieved data and associated details are then written to the output file ("access\_output.txt").

After processing all accesses, the code computes the cache hit ratio and bandwidth. It calculates the hit ratio by dividing the number of cache hits by the total number of cache accesses. Bandwidth is determined by dividing the total accessed bytes by the number of access cycles.

The code concludes by printing performance metrics specific to the cache's associative size, such as hit ratio and bandwidth, to the output file. It then closes the file streams and displays the final state of the cache using the

`print\_cache\_entries()` function.

Overall, this program simulates a cache system, reading memory access patterns from a file, and computing performance metrics based on cache hits, misses, and data retrieval.

1. cache.c

#include <stdio.h>

#include <string.h>

#include "cache\_impl.h" *// Including header file for cache implementation*

*// External declarations for variables used in other files*

extern int num\_cache\_hits;

extern int num\_cache\_misses;

extern int num\_bytes;

extern int num\_access\_cycles;

extern int global\_timestamp;

*// Arrays to store cache and memory data*

cache\_entry\_t cache\_array[CACHE\_SET\_SIZE][DEFAULT\_CACHE\_ASSOC];

int memory\_array[DEFAULT\_MEMORY\_SIZE\_WORD];

*/\* Initialize memory content with predefined sample data \*/*

void init\_memory\_content() {

*// Predefined sample data for memory initialization*

unsigned char sample\_upward[16] = {0x001, 0x012, 0x023, 0x034, 0x045, 0x056, 0x067, 0x078, 0x089, 0x09a, 0x0ab, 0x0bc, 0x0cd, 0x0de, 0x0ef};

unsigned char sample\_downward[16] = {0x0fe, 0x0ed, 0x0dc, 0x0cb, 0x0ba, 0x0a9, 0x098, 0x087, 0x076, 0x065, 0x054, 0x043, 0x032, 0x021, 0x010};

int index, i = 0, j = 1, gap = 1;

*// Initializing memory\_array using predefined sample data*

for (index = 0; index < DEFAULT\_MEMORY\_SIZE\_WORD; index++) {

memory\_array[index] = (sample\_upward[i] << 24) | (sample\_upward[j] << 16) | (sample\_downward[i] << 8) | (sample\_downward[j]);

if (++i >= 16) i = 0; *// Cycle for sample\_upward array*

if (++j >= 16) j = 0; *// Cycle for sample\_downward array*

*// Printing memory content for debugging*

printf("mem[%d] = %#x\n", index, memory\_array[index]);

}

}

*/\* Initialize cache content by setting default values \*/*

void init\_cache\_content() {

int i, j;

*// Loop to initialize cache\_array with default values*

for (i = 0; i < CACHE\_SET\_SIZE; i++) {

for (j = 0; j < DEFAULT\_CACHE\_ASSOC; j++) {

cache\_entry\_t \*pEntry = &cache\_array[i][j];

pEntry->valid = 0; *// Marking entry as invalid*

pEntry->tag = -1; *// No tag assigned initially*

pEntry->timestamp = 0; *// No access trial initially*

}

}

}

*/\* Utility function to print all cache entries (for debugging) \*/*

void print\_cache\_entries() {

int i, j, k;

printf("ENTRY >>\n");

*// Loop through each set in cache\_array*

for (i = 0; i < CACHE\_SET\_SIZE; i++) {

printf("[Set %d] ", i);

*// Loop through each entry in a set*

for (j = 0; j < DEFAULT\_CACHE\_ASSOC; j++) {

cache\_entry\_t \*pEntry = &cache\_array[i][j];

printf("Valid: %d Tag: %#x Time: %d Data: ", pEntry->valid, pEntry->tag, pEntry->timestamp);

*// Loop through each block in an entry*

for (k = 0; k < DEFAULT\_CACHE\_BLOCK\_SIZE\_BYTE; k++) {

printf("(%d)%#x ", k, pEntry->data[k]);

}

printf("\t");

}

printf("\n");

}

}

*// Function to check if data exists in cache*

int check\_cache\_data\_hit(void \**addr*, char *type*) {

num\_access\_cycles += CACHE\_ACCESS\_CYCLE; *// Adding cache access cycles*

int block\_addr = ((int)*addr* / DEFAULT\_CACHE\_BLOCK\_SIZE\_BYTE); *// Calculating block address*

int cache\_index = block\_addr % CACHE\_SET\_SIZE;

int tag = block\_addr / CACHE\_SET\_SIZE;

int byte\_offset = ((int)*addr*) % DEFAULT\_CACHE\_BLOCK\_SIZE\_BYTE; *// Calculating byte offset*

printf("Cache>> block\_addr = %d, byte\_offset = %d, cache\_index = %d, tag = %d\n", block\_addr, byte\_offset, cache\_index, tag);

for (int i = 0; i < DEFAULT\_CACHE\_ASSOC; i++) {

cache\_entry\_t \*entry = &cache\_array[cache\_index][i];

if (entry->valid && entry->tag == tag) {

num\_cache\_hits++; *// Cache hit*

entry->timestamp = global\_timestamp++; *// Update timestamp*

printf("cache hit!\n");

return cache\_index \* DEFAULT\_CACHE\_ASSOC + i; *// Return index of data*

}

}

*// Data not found in cache (cache miss)*

num\_cache\_misses++;

printf("cache miss!\n");

return -1;

}

*// Function to find an entry index within a cache set*

int find\_entry\_index\_in\_set(int *cache\_index*) {

int set\_index = *cache\_index* % CACHE\_SET\_SIZE;

int empty\_entry\_index = -1;

int lru\_index = 0;

int lru\_timestamp = cache\_array[set\_index][0].timestamp;

*// For Direct-mapped cache*

if (DEFAULT\_CACHE\_ASSOC == 1) {

cache\_entry\_t \*entry = &cache\_array[set\_index][0];

if (!entry->valid) {

return 0; *// Return first index if cache is empty*

}

return 0; *// Return first index if cache is full*

}

if (empty\_entry\_index != -1) {

return empty\_entry\_index; *// Return index of empty entry if found*

} else {

return lru\_index; *// Return Least Recently Used (LRU) index*

}

}

*// Function to access memory and update cache*

int access\_memory(void \**addr*, char *type*) {

num\_access\_cycles += MEMORY\_ACCESS\_CYCLE; *// Adding memory access cycles*

int memory\_block = ((int)*addr* / DEFAULT\_CACHE\_BLOCK\_SIZE\_BYTE); *// Calculating memory block*

int word\_index = ((int)*addr*) % DEFAULT\_CACHE\_BLOCK\_SIZE\_BYTE; *// Calculating word index*

int cache\_set\_index = memory\_block % CACHE\_SET\_SIZE; *// Calculating cache set index*

int cache\_entry\_index = find\_entry\_index\_in\_set(cache\_set\_index); *// Finding index to store in cache*

int tag = memory\_block / CACHE\_SET\_SIZE; *// Calculating tag*

cache\_entry\_t \*entry = &cache\_array[cache\_set\_index][cache\_entry\_index];

entry->valid = 1;

entry->tag = tag;

entry->timestamp = global\_timestamp++;

*// Reading data from memory to copy to cache*

int memory\_index = memory\_block \* (DEFAULT\_CACHE\_BLOCK\_SIZE\_BYTE / WORD\_SIZE\_BYTE);

for (int i = 0; i < DEFAULT\_CACHE\_BLOCK\_SIZE\_BYTE; i++) {

entry->data[i] = (char)((memory\_array[memory\_index + i / WORD\_SIZE\_BYTE] >> ((i % WORD\_SIZE\_BYTE) \* 8)) & 0xFF);

}

printf("access\_memory: data in cache after copy:\n"); *// Debugging: printing copied data in cache*

for (int i = 0; i < DEFAULT\_CACHE\_BLOCK\_SIZE\_BYTE; i++) {

printf("(%d)%#x ", i, entry->data[i]);

}

printf("\n");

*// Finding and returning data from cache based on type*

switch (*type*) {

case 'b': *// Byte*

return entry->data[word\_index];

case 'h': *// Half-word*

return ((entry->data[word\_index + 1] << 8) | (entry->data[word\_index] & 0xFF));

case 'w': *// Word*

return ((entry->data[word\_index + 3] << 24) | (entry->data[word\_index + 2] << 16) |

(entry->data[word\_index + 1] << 8) | (entry->data[word\_index] & 0xFF));

default:

return -1; *// Unknown type*

}

}

This code implements a simple cache simulator in C. It consists of functions to initialize memory and cache, check for data in the cache, find entry indices, and access memory while updating the cache.

The code starts by including necessary headers, defining external variables, and declaring arrays for the cache and memory. The `init\_memory\_content()` function initializes the memory with predefined sample data. It cycles through two arrays (`sample\_upward` and `sample\_downward`) and fills the memory array accordingly, printing the content for debugging purposes.

The `init\_cache\_content()` function initializes the cache array with default values - marking entries as invalid, setting tags to -1, and initializing timestamps.

The `print\_cache\_entries()` function is a debugging utility to print the content of the cache array. It iterates through each set and entry, displaying the validity, tag, timestamp, and data for each cache entry.

The `check\_cache\_data\_hit()` function checks if the data exists in the cache. It calculates the block address, cache index, tag, and byte offset, then searches for the data in the cache. If found, it updates the timestamp and returns the index; otherwise, it registers a cache miss.

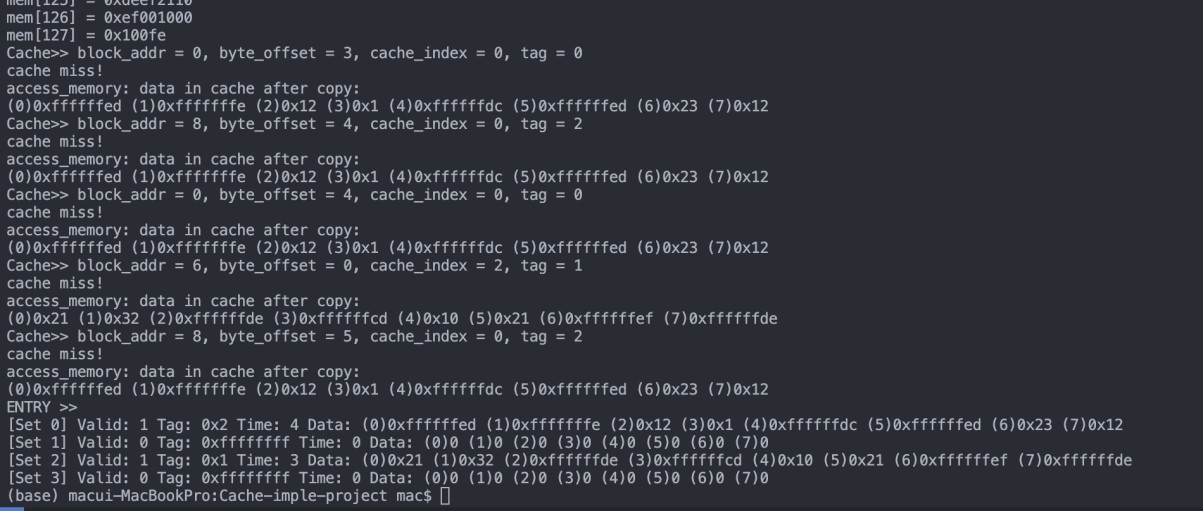
The `find\_entry\_index\_in\_set()` function is used to find an entry index within a cache set. It considers direct-mapped caches and searches for empty entries or the Least Recently Used (LRU) entry within a set.

The `access\_memory()` function is responsible for accessing memory and updating the cache. It calculates memory and cache indices, finds the entry index, and reads data from memory into the cache. Based on the access type ('b' for byte, 'h' for half-word, 'w' for word), it retrieves the appropriate data from the cache and returns it.

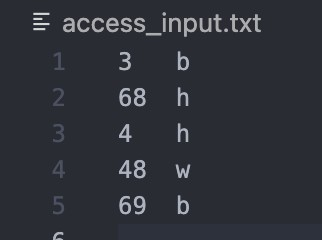
Overall, this code simulates a basic cache system in C, handling memory access, cache hits/misses, and updating cache entries based on access patterns.

Debugging print statements are included throughout the code to aid in understanding and tracing the execution flow.

1. console (Direct mapped cache EX)



1. Input & output



텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명