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.

2. cache.c

```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
        }
    }
}
```

```c
/* 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
```

```c
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.

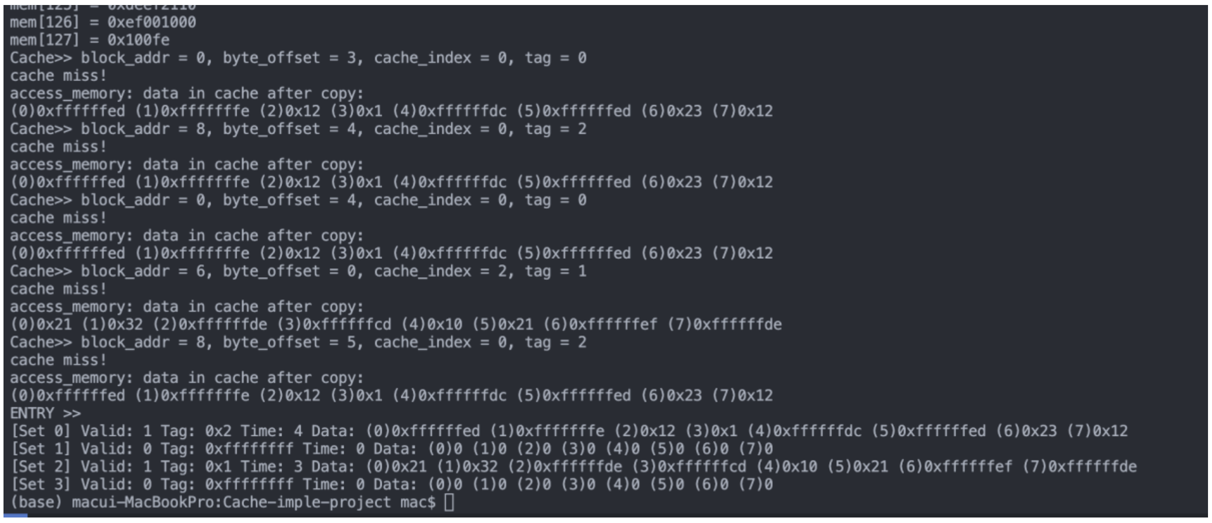
The `check_cache_data_hit()` function checks if the data exists in the cache. It calculates the block address, cache index, tag, and byte offset, then searches for the data in the cache. If found, it updates the timestamp and returns the index; otherwise, it registers a cache miss.

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.

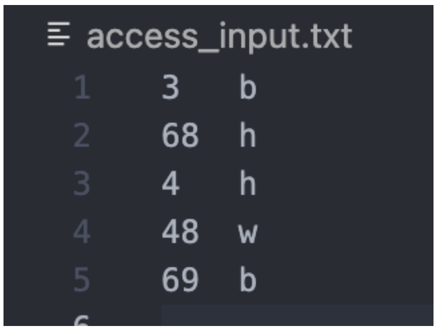
Overall, this code simulates a basic cache system in C, handling memory access, cache hits/misses, and updating cache entries based on access patterns. Debugging print statements are included throughout the code to aid in understanding and tracing the execution flow.

3. console (Direct mapped cache EX)

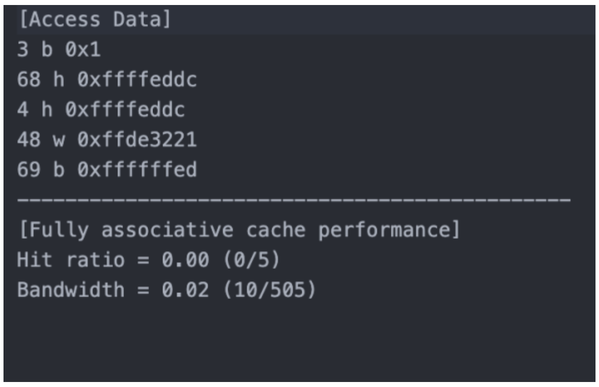


```
mem[125] = 0xdeef2110
mem[126] = 0xef001000
mem[127] = 0x100fe
Cache>> block_addr = 0, byte_offset = 3, cache_index = 0, tag = 0
cache miss!
access_memory: data in cache after copy:
(0)0xffffffed (1)0xfffffffe (2)0x12 (3)0x1 (4)0xffffffdc (5)0xffffffed (6)0x23 (7)0x12
Cache>> block_addr = 8, byte_offset = 4, cache_index = 0, tag = 2
cache miss!
access_memory: data in cache after copy:
(0)0xffffffed (1)0xfffffffe (2)0x12 (3)0x1 (4)0xffffffdc (5)0xffffffed (6)0x23 (7)0x12
Cache>> block_addr = 0, byte_offset = 4, cache_index = 0, tag = 0
cache miss!
access_memory: data in cache after copy:
(0)0xffffffed (1)0xfffffffe (2)0x12 (3)0x1 (4)0xffffffdc (5)0xffffffed (6)0x23 (7)0x12
Cache>> block_addr = 6, byte_offset = 0, cache_index = 2, tag = 1
cache miss!
access_memory: data in cache after copy:
(0)0x21 (1)0x32 (2)0xffffffde (3)0xffffffcd (4)0x10 (5)0x21 (6)0xffffffef (7)0xffffffde
Cache>> block_addr = 8, byte_offset = 5, cache_index = 0, tag = 2
cache miss!
access_memory: data in cache after copy:
(0)0xffffffed (1)0xfffffffe (2)0x12 (3)0x1 (4)0xffffffdc (5)0xffffffed (6)0x23 (7)0x12
ENTRY >>
[Set 0] Valid: 1 Tag: 0x2 Time: 4 Data: (0)0xffffffed (1)0xfffffffe (2)0x12 (3)0x1 (4)0xffffffdc (5)0xffffffed (6)0x23 (7)0x12
[Set 1] Valid: 0 Tag: 0xffffffff Time: 0 Data: (0)0 (1)0 (2)0 (3)0 (4)0 (5)0 (6)0 (7)0
[Set 2] Valid: 1 Tag: 0x1 Time: 3 Data: (0)0x21 (1)0x32 (2)0xffffffde (3)0xffffffcd (4)0x10 (5)0x21 (6)0xffffffef (7)0xffffffde
[Set 3] Valid: 0 Tag: 0xffffffff Time: 0 Data: (0)0 (1)0 (2)0 (3)0 (4)0 (5)0 (6)0 (7)0
(base) macui-MacBookPro:Cache-imple-project mac$ 
```

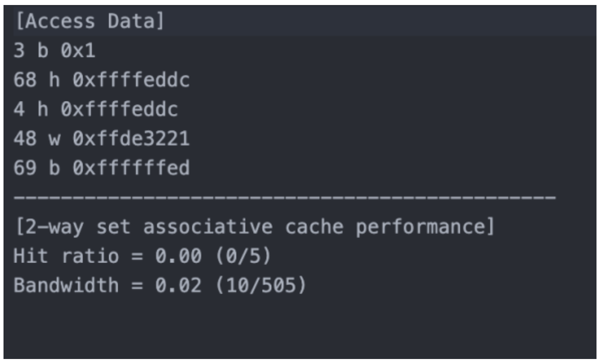
4. Input & output



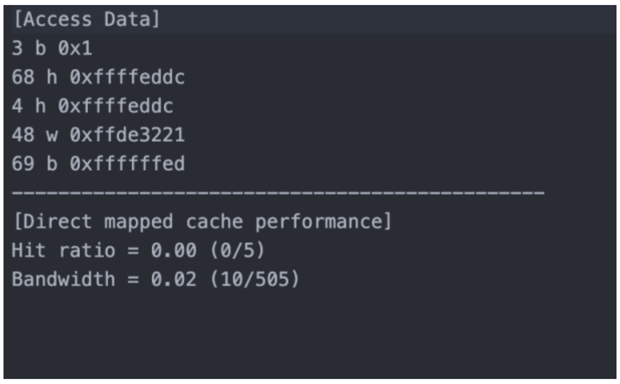
```
access_input.txt
3    b
68   h
4    h
48   w
69   b
```



```
[Access Data]
3 b 0x1
68 h 0xffffeddc
4 h 0xffffeddc
48 w 0xffde3221
69 b 0xffffffed
-----------------------------------------------
[Fully associative cache performance]
Hit ratio = 0.00 (0/5)
Bandwidth = 0.02 (10/505)
```



```
[Access Data]
3 b 0x1
68 h 0xffffeddc
4 h 0xffffeddc
48 w 0xffde3221
69 b 0xffffffed
-----------------------------------------------
[2-way set associative cache performance]
Hit ratio = 0.00 (0/5)
Bandwidth = 0.02 (10/505)
```



```
[Access Data]
3 b 0x1
68 h 0xffffeddc
4 h 0xffffeddc
48 w 0xffde3221
69 b 0xffffffed
---------------------------------------------
[Direct mapped cache performance]
Hit ratio = 0.00 (0/5)
Bandwidth = 0.02 (10/505)
```