# Assignment 4: Memory and Caching

## Jasraj Singh Johal 400434346

## Task 1: Theory

**a)** Requirements for a component to qualify as a Memory Cache:

1. Small Size: To maintain economic efficiency, a cache's storage is significantly smaller than the total data storage. Typically, cache sizes are less than 10% of the main storage size, often holding less than 1% of the data store.
2. Active Mechanism: A cache incorporates an active mechanism that analyzes each request and determines the appropriate response. This includes checking if the requested item is available in the cache, fetching a copy from the data store if not, and making decisions on which items to retain in the cache.
3. Transparency: A cache is considered transparent, meaning it can be introduced without modifying the requester or data store. The interface presented by the cache to the requester mirrors the data store's interface, and vice versa.
4. Automatic Operation: Typically, a cache operates without explicit instructions. Instead, it employs algorithms to examine the sequence of requests, using this information to autonomously manage the cache storage. The cache does not rely on external guidance for storing or retrieving data items.

Cache performance is evaluated based on its ability to enhance data retrieval speed compared to the data store. The cost of accessing the cache (Ch) when a requested item is found, and the cost of accessing the data store (Cm) in case of a cache miss, are crucial factors.

The hit ratio (r), representing the percentage of requests satisfied from the cache, is a key metric. Cache designers aim to increase the hit ratio or reduce the cost of a cache hit to enhance overall performance.

$$r = \mathrm{Nhit}/(\mathrm{Nhit} + \mathrm{Nmiss})$$

$$Miss\ Ratio = 1 - Hit\ Ratio$$

$$Cost\ per\ Request = r \times Cost\ of\ Hit + (1 - r) \times Cost\ of\ Miss$$

Furthermore, to address higher startup costs, preloading the cache with anticipated values is suggested, allowing for improved initial hit ratios and efficient system operation over time. Automated methods like periodic storage snapshots or reference-based prefetching contribute

to optimizing cache performance, particularly relevant for web pages with multiple referenced elements.

The benefit of using a cache, on average, compared to not using a cache (where the cost is Cm without cache) is given by:

$$Benefit = Cm - Ccache = r \times (Cm - Ch) \geq 0$$

Increasing the hit ratio or decreasing the cost of a hit can improve cache performance.

b) In a Direct-Mapped Memory Cache implemented on a 16-bit architecture with 8 blocks of memory, each block being 32 bytes long, the number of bits necessary to represent the tag of each block is calculated. The block size (B) is 32 bytes, requiring 5 bits for the block offset within a block. With 8 blocks in the cache, the number of index bits (I) is 3 bits. Using the formula

$$T = Address\ Size - (No.of\ Index\ bits + Block\ offset\ bits)$$

It is determined that 8 bits are needed to represent the tag of each block.

Cache Configuration:

1. Total Blocks in Cache: 8

2. Block ID Bits Needed: $\log_2(8) = 3$ bits.

3. Block Size: 32 bytes

4. Offset Bits Needed: $\log_2(32) = 5$ bits.

5. Address Size: 16 bits (for a 16-bit architecture)

6. Tag Bits Calculation: Total Address Bits - (Block ID Bits + Offset Bits)

   16 - (3 + 5) = 8 bits for the tag.

A byte address in memory is divided into three parts: Tag, Block ID, and Offset. Powers of two are employed to simplify the process of parsing a byte address and locating the correct data stored in the cache. Specifically, block division, cache line designation, and bit extraction for the block number are facilitated by powers of two, streamlining hardware implementation and avoiding the need for modulo arithmetic.

The cache functions as a high-speed memory subsystem, storing and retrieving frequently accessed data efficiently. It utilizes a direct-mapped technique, associating each block with a specific cache slot based on its block number. Tags are employed to uniquely identify entries in a slot, enhancing efficiency. The cache lookup algorithm operates conceptually as an array, utilizing the block number as an index. Powers of two simplify the hardware implementation, making block size, cache size, and address space choices strategically. This avoids the need for

complex arithmetic operations during cache lookup, contributing to the overall efficiency of the system.

c) In a 32-bit architecture with 256 MB of RAM utilizing demand paging, the operating system reserves 128 MB of memory for privileged use, leaving the remaining memory for a page table consisting of 1024 entries and a corresponding set of 1024 frames. To determine the physical memory size of each page, we observe that 128 MB is 2^7 × 2^20 , and dividing it by 1024 gives

$$\frac{2\text{^}7\times 2\text{^}20}{2\text{^}10} = 2\text{^}7 \times 2\text{^}10 = 128\,KB \text{ per page.}$$

Address Translation Process by MMU:

1. Page Lookup: In the typical operating sequence, the Memory Management Unit (MMU) loads an address 'a' on page 'P' and checks the page table 'Pt[P]' to determine if the presence bit is set.
2. Presence Bit Check:

   If the presence bit is not set:

   - MMU raises a page fault.
   - Operating system (OS) handles the page fault by loading page 'P' from the hard drive and updating the page table 'Pt'.
   - If no space is available in memory, the OS swaps a page, prioritizing one with a use bit set to 0.
   - If the modified bit is set, indicating changes, the page needs to be saved; otherwise, it is discarded.

   If the presence bit is set:

   - MMU looks up the address, updating the use bit and modified bit in the page table.
3. Efficiency Enhancement with Powers of 2:

   For powers of 2, the page table lookup is optimized as

   $$P = pagetable[high\_order\_bits(V)] + low\_order\_bits(V)$$

   making it more efficient.

4. The Translation Lookaside Buffer (TLB) is a cache that stores recent translations of virtual memory to physical memory addresses, significantly speeding up the address translation process. During the initial translation, the MMU copies the page table entry to the TLB. Subsequent lookups involve parallel operations: the standard address translation steps

and a high-speed TLB search. If the requested information is found in the TLB, the MMU uses the TLB data, bypassing the standard translation. If not, the standard translation proceeds.

5. A TLB enhances performance by optimizing successive lookups, avoiding the need to index into the page table for every translation. This performance improvement is particularly significant for architectures storing page tables in memory, as TLBs eliminate the need for repeated memory accesses during translation, resulting in faster and more efficient address resolution.

## Task 2: Array Storage

```c
// My assumption: i, j are 0-indexed and m, n are not
// My assumption: -1234 is a sentinel value that is used as an error code for
two_d_int.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Node {
    int payload;
    struct Node *next;
} node;

// Function to allocate a 2D array of given dimensions and element size
char* two_d_alloc(size_t N, size_t M, size_t sz) {
    if (sz <= 0) return NULL; // Check for invalid element size
    char *buff = malloc((N * M) * sz); // Allocate memory for the array
    return buff;
}

// Function to deallocate memory allocated for the 2D array
void two_d_dealloc(char* array) {
    free(array);
}

// Function to store an integer in the 2D array at the specified position
int two_d_store_int(int arg, char* array, size_t i, size_t j, size_t M, size_t N)
{
```

```
    if (array == NULL || i >= N || j >= M) return -1; // Check for invalid
arguments
    array[i * M + j] = arg; // Store the integer in the array
    return 1;
}

// Function to fetch an integer from the 2D array at the specified position
int two_d_fetch_int(char* array, size_t i, size_t j, size_t M, size_t N) {
    if (array == NULL || i >= N || j >= M) {
        printf("\nInvalid fetch :(");
        return -1234; // Return sentinel value for invalid fetch
    }
    return array[i * M + j]; // Return the fetched integer
}

// Function to store data of arbitrary type in the 2D array at the specified
position
int two_d_store(void* arg, char* array, size_t i, size_t j, size_t M, size_t N,
size_t sz) {
    if (array == NULL || i >= N || j >= M) return -1; // Check for invalid
arguments
    memcpy(&array[(i * M + j) * sz], arg, sz); // Copy the data to the array
    return 1;
}

// Function to fetch data of arbitrary type from the 2D array at the specified
position
void* two_d_fetch(char* array, size_t i, size_t j, size_t M, size_t N, size_t sz)
{
    if (array == NULL || i >= N || j >= M) {
        printf("\nInvalid fetch with garbage values");
    }
    return &array[(i * M + j) * sz]; // Return a pointer to the fetched data
}

// Function to print a 2D array of nodes, displaying their addresses and
hexadecimal values
void print_node_arr(char* array, size_t M, size_t N, size_t sz) {
    for (size_t i = 0; i < N; ++i) {
        for (size_t j = 0; j < M; ++j) {
            // Print the address of the current element
            printf("%p\t", (void*)&array[(i * M + j) * sz]);

            // Print the hexadecimal values of each byte in the element
            for (size_t k = 0; k < sz; ++k) {
```

```
                printf("%02X ", array[(i * M + j) * sz + k]);
            }
            printf("\t");
        }
        printf("\n");
    }
    printf("\n");
}


int main() {
    // Allocate a 2D array for integers
    char* bf = two_d_alloc(2, 3, sizeof(int));
    if (bf == NULL) printf("Failed due to invalid size");

    // Test cases for two_d_store_int
    printf("\n------------------------------------------------------------------------
------------------------");
    printf("\ntwo_d_store_int Test Cases:\n");
    int result1 = two_d_store_int(42, bf, 0, 0, 3, 2);
    printf("\nTest Case 1 -> Result: %s, Expected: 1", (result1 == 1) ? "Passed
:)" : "Fail :(");

    int result2 = two_d_store_int(99, bf, 1, 2, 3, 2);
    printf("\nTest Case 2 -> Result: %s, Expected: 1", (result2 == 1) ? "Passed
:)" : "Fail :(");

    int result3 = two_d_store_int(77, bf, 0, 1, 3, 2);
    printf("\nTest Case 3 -> Result: %s, Expected: 1", (result3 == 1) ? "Passed
:)" : "Fail :(");

    int result4 = two_d_store_int(123, bf, 4, 7, 3, 2);
    printf("\nTest Case 4 -> Result: %s, Expected: -1", (result4 == -1) ? "Passed
:)" : "Fail :(");

    // Test cases for two_d_fetch_int
    printf("\n------------------------------------------------------------------------
------------------------");
    printf("\ntwo_d_fetch_int Test Cases:\n");

    int result5 = two_d_fetch_int(bf, 0, 0, 3, 2);
    printf("\nTest Case 1 -> Result: %s, Expected: 42", (result5 == 42) ? "Passed
:)" : "Fail :(");

    int result6 = two_d_fetch_int(bf, 1, 2, 3, 2);
```

```
    printf("\nTest Case 2 -> Result: %s, Expected: 99", (result6 == 99) ? "Passed
:)" : "Fail :(");

    int result7 = two_d_fetch_int(bf, 0, 1, 3, 2);
    printf("\nTest Case 3 -> Result: %s, Expected: 77", (result7 == 77) ? "Passed
:)" : "Fail :(");

    int result8 = two_d_fetch_int(bf, 4, 7, 3, 2);
    printf("\nTest Case 4 -> Result: %s, Expected: -1234", (result8 == -1234) ?
"Passed :)" : "Fail :(");

    two_d_dealloc(bf);

    // Allocate a 2D array for floats
    char* bf2 = two_d_alloc(3, 3, sizeof(float));
    if (bf2 == NULL) printf("Failed Allocation");

    // Test cases for two_d_store
    printf("\n-------------------------------------------------------------------
-----------------------");
    printf("\ntwo_d_store Test Cases:\n");

    float floatValue1 = 1.111;
    int result9 = two_d_store((void*)&floatValue1, bf2, 1, 1, 3, 2,
sizeof(float));
    printf("\nTest Case 1 -> Result: %s, Expected: 1", (result9 == 1) ? "Passed
:)" : "Fail :(");

    float floatValue2 = 2.222;
    int result10 = two_d_store((void*)&floatValue2, bf2, 0, 0, 3, 2,
sizeof(float));
    printf("\nTest Case 2 -> Result: %s, Expected: 1", (result10 == 1) ? "Passed
:)" : "Fail :(");

    float floatValue3 = 3.333;
    int result11 = two_d_store((void*)&floatValue3, bf2, 2, 2, 3, 2,
sizeof(float));
    printf("\nTest Case 3 -> Result: %s, Expected: -1", (result11 == -1) ?
"Passed :)" : "Fail :(");

    float floatValue4 = 4.444;
    int result12 = two_d_store((void*)&floatValue4, bf2, 6, 6, 3, 2,
sizeof(float));
    printf("\nTest Case 4 -> Result: %s, Expected: -1", (result12 == -1) ?
"Passed :)" : "Fail :(");
```

```
    // Test cases for two_d_fetch
    printf("\n-------------------------------------------------------------------------
------------------------");
    printf("\ntwo_d_fetch Test Cases:\n");

    float result13 = *(float*)(two_d_fetch(bf2, 1, 1, 3, 2, sizeof(float)));
    printf("\nTest Case 1 -> Result: %s, Expected: 1.111", (result13 ==
(float)1.111) ? "Passed :)" : "Fail :(");

    float result14 = *(float*)(two_d_fetch(bf2, 0, 0, 3, 2, sizeof(float)));
    printf("\nTest Case 2 -> Result: %s, Expected: 2.222", (result14 ==
(float)2.222) ? "Passed :)" : "Fail :(");

    float result15 = *(float*)(two_d_fetch(bf2, 2, 2, 3, 2, sizeof(float)));
    printf("\nTest Case 3 -> Result: %s, Expected: 0.0", (result15 == 0.0) ?
"Passed :)" : "Fail :(");

    float result16 = *(float*)(two_d_fetch(bf2, 6, 6, 3, 2, sizeof(float)));
    printf("\nTest Case 4 -> Result: %s, Expected: 0.0", (result16 == 0.0) ?
"Passed :)" : "Fail :(");

    two_d_dealloc(bf2);
    printf("\n-------------------------------------------------------------------------
------------------------");
    printf("\nAll test cases complete ;)");


    // Bonus
    // Test two_d_alloc
    // Set the size parameters for the 2D array and the size of each element
    size_t M = 4, N = 4, sz = sizeof(int);
    char* bf3 = two_d_alloc(3, 3, sizeof(float));
    // Test Node storage and fetch

    // Display the state of the 2D array before storing nodes
    printf("Before:\n");
    print_node_arr(bf3, M, N, sz);

    // Create and store a node with payload -1 and NULL next pointer at position
(0, 0)
    node last;
    last.payload = -1;
    last.next = NULL;
    two_d_store((void *)&last, bf3, 0, 0, M, N, sz);
```

```
    // Fetch the stored node at position (0, 0)
    node *lst = (node *)two_d_fetch(bf3, 0, 0, M, N, sz);

    // Create and store a node with payload 100 and a next pointer pointing to
the previously stored node at (0, 0)
    node d1;
    d1.payload = 100;
    d1.next = lst;
    two_d_store((void *)&d1, bf3, 0, 1, M, N, sz);

    // Fetch the stored node at position (0, 1)
    lst = (node *)two_d_fetch(bf3, 0, 1, M, N, sz);

    // Create and store a node with payload 200 and a next pointer pointing to
the previously stored node at (0, 1)
    d1.payload = 200;
    d1.next = lst;
    two_d_store((void *)&d1, bf3, 1, 1, M, N, sz);

    // Display the state of the 2D array after storing nodes
    printf("After:\n");
    print_node_arr(bf3, M, N, sz);

    // Deallocate memory for the 2D array
    two_d_dealloc(bf3);

    return 0;
}
```