Direct Caching Simulator

A paper submitted to:

Prof. Ronald M. Pascual

College of Computer Studies

Department of Computer Technology

Term 3, AY 2023-2024

In partial fulfillment of the requirements for

CSARCH2: Introduction to Computer Organization and Architecture 2

Submitted by:

Chua, Judy P.

Ha, Eun Ji

Telosa, Arwyn Gabrielle A.

Uy, Jasmine Louise

July 25, 2024

Table of Contents

I. Documentation

A. Description of the project

Cache memory is important to learn and understand, considering how memory is closely related to working with the CPU in storing data and instructions, especially when frequently used. It plays a crucial role inside the computer since it affects the system's overall performance. Therefore, a computer scientist must be significantly familiar with the topic in order to optimize and design efficient computer systems. Understanding how they work can further improve current processing speeds, refine resource management, and evolve computer systems.

With that in mind, this project aims to show our understanding of direct cache mapping by creating a web-based application that asks users to input their preferred block size, main memory size, main memory access time, cache memory size, cache memory access time, and the example program flow to be used for showing the output. For the main and cache memory sizes, the user is limited to inputting positive numbers with either a word or block as the unit. The time requested is automatically set to nanoseconds. For the output, the program will compute the number of cache hits and cache misses in the memory, the miss penalty, average memory access time, the total memory access time, and the final memory table after execution. It also has the option to be exported as a text file, so the user can view the results offline. After which, the user can either edit their current input to produce another output once enter is clicked or reset and remove all current inputs.

B. Timeline

To achieve the goals of this project, the following has been set to show the progress done:

Task	Date	Done by:
Initial draft of the program (in HTML and CSS)	Jul 1, 2024 to Jul 12, 2024	Uy
Set up “back-end” of the program (JavaScript)	Jul 12, 2024 to Jul 15, 2024	Telosa
Fix and debug code	Jul 15, 2024 to Jul 20, 2024	Chua
Design the final output	Jul 20, 2024 to Jul 24, 2024	Ha
Readme, Documentation, Analysis Write-up, and Demonstration	Jul 24, 2024 to Jul 25, 2024	Everyone

II. Demonstrations

A. Example 1: Program flow by block

An example run is shown in Figure 1 below. The main memory size is given in words, the cache memory size is given in blocks, and the program flow is given in blocks.

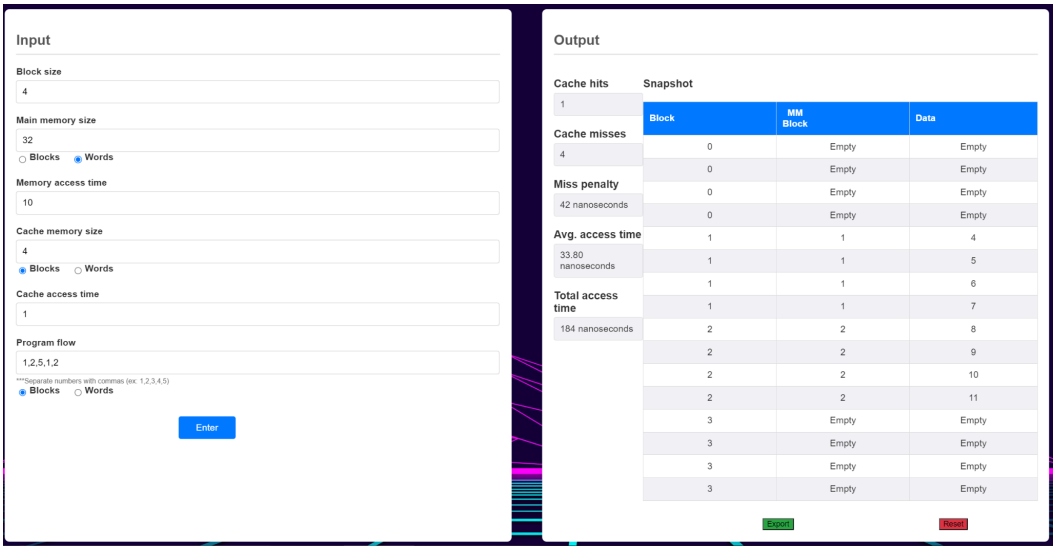

Figure 1: *First Simulation*

1. Inputs

Given the inputs shown in Figure 1 above, we can see that the block size is 4 words, which means the main memory (MM) has 8 blocks, its access time is 10ns, the cache memory has 4 blocks, and its access time is 1 ns. The sequence of the blocks to be fetched is 1, 2, 5, then 1 and 2 again.

2. Processing

When the CPU requests for MM's block 1 from the cache memory, it will be a cache miss because the cache memory is initially empty. This extra time it takes to probe will add 1ns to the total access time. Since the cache does not have the requested block, it will fetch the respective block from the main memory. Because it takes 10ns to move words from memory and there are 4 words to fetch, 40ns is added to the total time. MM's block 1 will go to the cache memory's block 1 because 1 % 4 $(MM\ block\ no.\ \%\ cache\ memory\ size)$ is 1. Now that block 1 is

in the cache, it can now move to the CPU. The cache access time is 1ns and we are copying 4 words, so 4ns will be added to the total time, making the running total 45ns.

The same process will occur when the CPU requests MM's block 2 since it is also initially absent. This brings the total to 90ns.

The same is also true for MM's block 5. This time, MM's block 5 will go to the cache's block 1 (replacing the earlier block 1) because $5 \% 4$ is 1. The total running time is now 135ns.

The next request from the CPU is 1 again. It will be a cache miss because it was overwritten by block 5; thus, it will go through the entire process again. This makes the total access time 180ns.

Next, 2 is requested again, but it is already in the cache, so the only thing to do is to send it to the CPU. Moving the 4 words to the CPU will take 4ns, so the final total access time is 184ns.

3. Outputs

There is only 1 cache hit, which is the final request for block 2. The other 4 requests are cache misses. The miss penalty is the time it takes if there is a 1-word miss. This is given by $cache\ access\ time\ +\ block\ size\ *\ memory\ access\ time\ +\ cache\ access\ time$. Plugging in the example inputs, $1 + 4 * 10 + 1$ yields 42ns. The average access time is given by

$hit\ ratio\ *\ cache\ access\ time\ +\ miss\ ratio\ *\ miss\ penalty$. Plugging in the example inputs, $\frac{1}{5} * 1 + \frac{4}{5} * 42$ gives 33.8ns. Finally, as mentioned earlier, the total access time is 184ns.

B. Example 2: Program flow by word

Another example run is shown in Figure 2 below. This time, the main memory size is given in blocks, the cache memory size is given in words, and the program flow is also given in words.

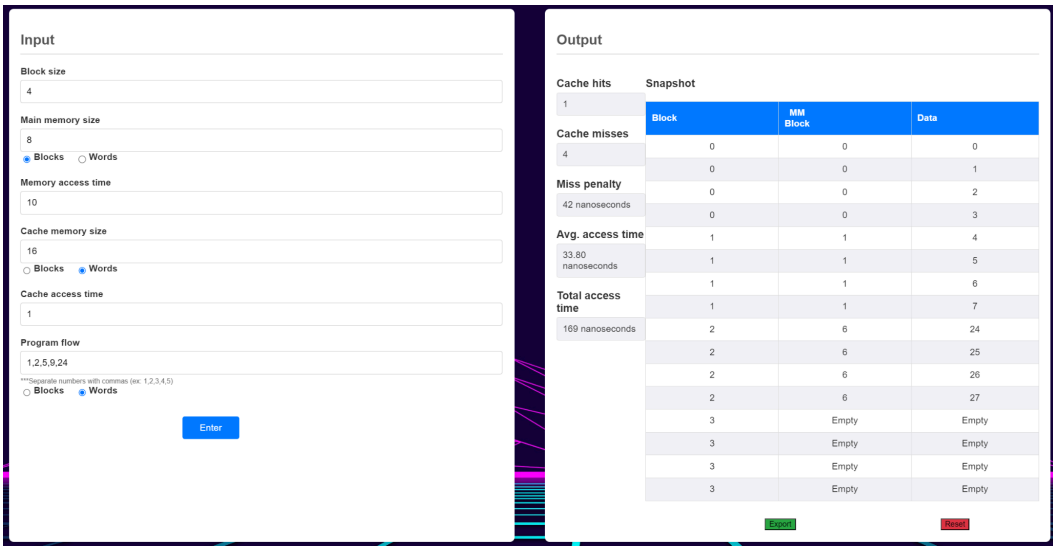

Figure 2: *Second Simulation*

1. Inputs

Given the inputs shown in Figure 2 above, we can see that the block size is 4 words, the main memory (MM) has 8 blocks, and its access time is 10ns. The cache memory has 16 words, which means it has 4 blocks, and its access time is 1ns. The sequence of the words to be fetched is 1, 2, 5, 9, and 24.

2. Processing

When the CPU requests for MM's word 1 from the cache memory, it will be a cache miss because the cache memory is initially empty. This extra time it takes to probe will add 1ns to the total access time. Since the cache did not have the requested word, it will fetch the block that MM word 1 belongs to from the main memory (because cache memory only handles blocks). MM's word 1 belongs to MM block 0 because $\lfloor \frac{1}{4} \rfloor$ ($\lfloor \frac{word\ no.}{block\ size} \rfloor$) is 0. Because it takes 10ns to move words from memory and there are 4 words to fetch, 40ns is added to the total time. MM's block 0 will go to the cache memory's block 0 because $0 \% 4$ ($MM\ block\ no.\ \%\ cache\ memory\ size$) is 1. Now that MM word 1 is in the cache, it can now move to the CPU. The cache access time is 1ns and we are copying 1 word so 1ns will be added to the total time, making the running total 42ns.

The next request from the CPU is MM word 2. It was fetched to the cache together with word 1, so it is a cache hit. The cache access time it takes to move word 2 from cache to the CPU is 1ns, so the total time is now 43ns.

The next thing to be fetched is MM's word 5. It is not yet in the cache memory and this miss will cost 1ns. Now, the block containing word 5 should be moved from the MM to the cache. Word 5 is in MM block 1 because $\lfloor \frac{5}{4} \rfloor$ is 1. Moving MM's block 1 to cache will take 40ns. MM block 1 will go to cache block 1 because $1 \% 4$ is 1. Moving word 5 from the cache to CPU will take 1ns. This makes the running total access time 85ns.

Next, 9 is requested. Because it is also not yet in the cache, the same process as the fetching of word 5 will occur. This brings the total time to 127ns.

The last thing to fetch is word 28. It is not yet in cache at this point so it will also follow the same process as 5 and 9. It belongs to MM block 6 (as given by $\lfloor \frac{24}{4} \rfloor$). Because 6 % 4 is 2, MM block 6 will go to cache's block 2, replacing the earlier MM block 2. The word will then move from the cache memory to the CPU. After this process, the total access time is now 169ns.

3. Outputs

There is only 1 cache hit, which occurred when fetching word 2 since it came together with word 1. The other 4 requests are cache misses. The miss penalty is the time it takes if there is a 1-word miss. This is given by $cache\ access\ time\ +\ block\ size\ *\ memory\ access\ time\ +\ cache\ access\ time$. Plugging in the example inputs, $1 + 4 * 10 + 1$ yields 42ns. Finally, the average access time is given by $hit\ ratio\ *\ cache\ access\ time\ +\ miss\ ratio\ *\ miss\ penalty$. Plugging in the example inputs, $\frac{1}{5} * 1 + \frac{4}{5} * 42$ gives 33.8ns.

III. Code Structure Analysis

A. HTML (index.html)

The HTML file structures the user interface into two main sections: input and output. It includes forms for user inputs and areas to display the simulation results.

- **Input Section:**
    - Block size
    - Main memory (MM) size (blocks / words)
    - Memory access time (in ns)
    - Cache memory size (blocks / words)
    - Cache access time (in ns)
    - Program flow (blocks / words)
- **Output Section:**
    - Number of cache hits
    - Number of cache misses
    - Miss penalty
    - Average access time
    - Total access time
    - Snapshot of the cache memory

B. JavaScript (script.js)

The JavaScript file handles the logic for the cache simulation, input validation, result calculation, and dynamic DOM manipulation.

- **Event Listeners:**
    - DOMContentLoaded: Sets up event listeners for form submission, export button, and reset button once the DOM content is loaded.
    - Form Submission Event: Prevents default form submission, checks input, processes the cache simulation, and updates the DOM with results if inputs are valid.

    - Export Button Event: Creates and downloads a text file with the simulation results.
    - Reset Button Event: Clears all inputs and hides the output section.

- **Functions:**
    - formatNum(num): Formats numbers to two decimal places if necessary
    - checkInputs(): Validates user inputs, ensuring that all required fields are filled and values are positive numbers.
    - postiveonly(userInput): Ensures that only positive numbers can be entered in the input fields.

- **Simulation Logic**
    - Input Handling: Retrieves and processes user inputs from the form, including converting program flow to block addresses if necessary/
    - Cache Initialization: Initializes a cache array based on the specified cache memory size.
    - Execution: Iterates over the program flow addresses to simulate cache access, counting hits and misses.
    - **Metrics Calculation:**
        - Hit Ratio: Proportion of cache hits.
        - Miss Ratio: Proportion of cache misses.
        - Miss Penalty: Calculated based on cache access time and memory access time.

        - Average Access Time: Weighted average of access times considering hits and misses.
        - Total Access Time: Total time taken for all memory accesses in the program flow.

- **DOM Manipulation**
    - Display Results: Updates the HTML elements with calculated metrics and populates the cache snapshot table.
    - Export Results: Compiles results into a text format and generates a download.
    - Reset Inputs: Clears all input fields and hides the output section.

C. CSS (style.css)

The CSS file styles the user interface, ensuring a visually appealing and user-friendly experience.

- **General Styles:**
    - Body: Sets a background image, font family, and basic styling for the entire page.
    - Headings: Styles for the main heading and subheadings, including text alignment, color, and animation for color change effects.
- **Layout and Flexbox**
    - flex-parent: Uses Flexbox to arrange the input and output sections side by side.
    - big-box: Styles for the input and output sections, including background color, border radius, box shadow, padding, and width.

- **Form Elements**
    - Text Inputs: Styles for number and text inputs, including width, padding, border, border radius, font size, and focus effects.
    - Radio Inputs: Styles for radio buttons and their labels.
    - Submit Button: Styles for the submit button, including background color, text color, padding, border radius, and hover effects.
    - relative-button: Positioning for the submit button to align it correctly within the form.
    - Note Message: Styling for informational messages.
    - Incomplete Message: Styling for error messages.
- **Output Section**
    - big-box-right: Initially hidden; displayed only when the form is submitted successfully.
    - output-flex: Uses Flexbox to arrange output elements.
    - Text Blocks: Styles for text blocks displaying results, including background color, padding, border, and margin.
- **Table Styles**
    - Table: Styles for snapshot table, including width, border collapse, and margin.
    - Table Cells: Styles for table header and cells, including padding, border, text alignment, and background color for alternating rows.
- **Export and Reset Buttons**

- Export and Reset Buttons: Styles for export and reset buttons, including background color, margin, and hover effects.
- radio-group: Styles for the radio button groups to ensure alignment.