Cache Simulator Project

This project dives deep into how computer memory works, focusing on building a "cache simulator" using Python. Imagine a cache as a super-fast, small memory that acts as a bridge between your computer's main brain (CPU) and its slower, larger main memory (RAM). The big idea here is to speed up data access by keeping frequently used information super close to the CPU.

Our simulator brings a realistic memory system to life with two levels of cache:

* **L1 Caches:** These are the fastest and smallest caches. They're split into two specialized areas: one for "instructions" (think of these as the CPU's to-do list) and another for "data" (the actual information the CPU needs to work with).
* **L2 Cache:** This cache is a bit bigger and a touch slower than L1, but still way faster than main memory. It's like a common waiting room for both instructions and data that the CPU might need soon.

We've designed the simulator to be super flexible, letting you tweak various settings:

* **Block Sizes:** This determines how much information gets moved at once between memory levels.
* **Associativity:** This fancy term refers to how many different spots a piece of data can call home within the cache, which impacts how cleverly it can find things.
* **Write-Back Policy:** This is a clever strategy for handling changes to data in the cache. Instead of immediately writing changes back to the slower memory, it waits until absolutely necessary, saving a lot of time.

To make sure everything runs smoothly and correctly, we've also built a comprehensive set of "unit tests." Think of these as tiny, targeted experiments that rigorously check every single part of the cache and memory system to ensure it behaves exactly as it should.

# Overview

This project implements a cache simulator in Python, mimicking a memory hierarchy that includes L1 instruction/data caches and a unified L2 cache. The simulator allows you to configure block sizes, associativity, and a write-back policy. We've also included extensive unit tests to confirm the accuracy of the cache and memory hierarchy's behavior.

# Implementation Details

## 1. memory.py (Cache Simulator)

### Features Implemented

* **L1 Caches:** We've included separate instruction and data caches, each with a 1KB capacity.
* **L2 Cache:** A unified cache with a 16KB capacity.
* **Configurable Block Size:** You can set the L1 block size to 4, 8, 16, or 32 bytes, and the L2 block size to 16, 32, or 64 bytes.
* **Associativity:** L1 caches can be direct-mapped or fully associative, while L2 is direct-mapped.
* **Write-Back Policy:** Any "dirty" blocks (those that have been modified) are written back to the next memory level when they're evicted from the cache.
* **LRU Eviction:** We use the "Least Recently Used" policy for set-associative and fully associative caches, meaning the data that hasn't been used in the longest time gets removed first.
* **Statistics:** The simulator keeps track of important metrics like hits, misses, and writebacks, and also calculates hit/miss rates.

### Key Classes

* Block: This class represents a single cache block, storing information about its validity, tag, writeback status, and when it was last used.
* Cache: This class models a cache, allowing you to configure its size, block size, and associativity. It handles data access, LRU eviction, and statistics tracking.
* MemSimulator: This is the main class that orchestrates the entire memory hierarchy, managing the interactions between the L1 and L2 caches.

### Changes Made

* We've added checks for the parameters related to block size and associativity.
* The LRU eviction logic has been fully implemented.
* We've included write-back handling and the tracking of various statistics.
* Methods for accessing the cache and retrieving statistics have been provided.

## 2. test\_memory.py (Unit Tests)

### Testing Approach

* **Fixtures:** We've used pytest fixtures to set up both direct-mapped and fully associative cache instances specifically for testing.
* **Block Tests:** These tests verify that cache blocks are initialized correctly by default.
* **Cache Tests:** We've thoroughly checked correct initialization, hit/miss behavior, LRU eviction, and the write-back logic of the cache.
* **MemSimulator Tests:** These tests validate the correct behavior of the entire memory hierarchy, including how L1 and L2 caches interact and how the overall cost is calculated.

### Example Tests

* **Initialization:** These tests ensure that caches are created with the precise parameters we expect.
* **Hit/Miss:** We confirm that repeated accesses to data result in the expected hits and misses.
* **LRU Eviction:** These tests involve filling up the cache and then checking that the least recently used block is indeed the one that gets evicted.
* **Write-Back:** We verify that dirty blocks are correctly written back to the next level when they're evicted from the cache.
* **Cost Calculation:** These tests ensure that the simulator accurately computes the cost based on the number of misses and writebacks.

### Changes Made

* We've added detailed tests to cover all the main behaviors of both the cache and the memory simulator.
* We've used pytest for efficient test discovery and execution.
* A \_\_main\_\_ block has been added, allowing you to run tests simply by executing python test\_memory.py. This block invokes pytest and displays a concise summary of the test results.
* The test runner has been adjusted to only show the summary line (e.g., 9 passed in 0.12s) by removing verbose and print options.

## How to Run the Tests

1. **Install pytest** (if you haven't already):pip install pytest
2. **Run the tests** from your project directory:python test\_memory.py

You'll then see a summary like:9 passed in 0.12s

## Summary

* memory.py is where we've implemented a flexible, write-back cache simulator that includes LRU eviction and gathers statistics.
* test\_memory.py provides a thorough set of unit tests using pytest, making sure all the cache behaviors are correct.
* Overall, this project is easy to test and can be readily expanded for even more experiments with cache and memory hierarchies.