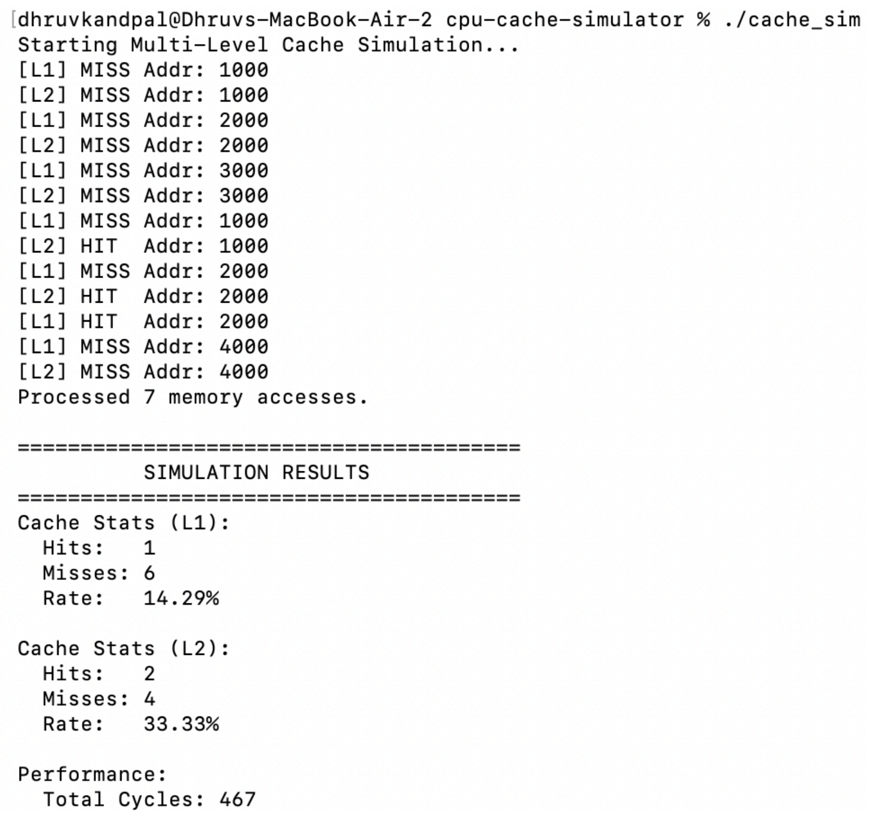
Multilevel CPU Cache Simulator

1. System Overview

The Multilevel Cache Simulator is a software-based simulation of a two-level CPU cache hierarchy (L1 and L2). The system mimics the behavior of hardware memory subsystems, processing memory access requests to calculate performance metrics such as hit rates, miss counts, and simulated latency. The primary objective is to demonstrate the principles of spatial and temporal locality, as well as the impact of cache capacity and associativity on system performance.



```
dhruvkandpal@Dhruvs-MacBook-Air-2 cpu-cache-simulator % ./cache_sim
Starting Multi-Level Cache Simulation...
[L1] MISS Addr: 1000
[L2] MISS Addr: 1000
[L1] MISS Addr: 2000
[L2] MISS Addr: 2000
[L1] MISS Addr: 3000
[L2] MISS Addr: 3000
[L1] MISS Addr: 1000
[L2] HIT  Addr: 1000
[L1] MISS Addr: 2000
[L2] HIT  Addr: 2000
[L1] HIT  Addr: 2000
[L1] MISS Addr: 4000
[L2] MISS Addr: 4000
Processed 7 memory accesses.

========================================
          SIMULATION RESULTS
========================================
Cache Stats (L1):
  Hits:   1
  Misses: 6
  Rate:   14.29%

Cache Stats (L2):
  Hits:   2
  Misses: 4
  Rate:   33.33%

Performance:
  Total Cycles: 467
```

Fig 1 : Output on Running the simulator (Memory Addresses are hardcoded)

2. System Architecture

2.1 Hierarchy Structure

The simulation implements a strictly ordered memory hierarchy:

1. **Level 1 (L1) Cache:** The smallest and fastest storage level, positioned closest to the CPU. It is configured with low latency and limited capacity.
2. **Level 2 (L2) Cache:** A larger, slower storage level that acts as a backstop for L1 misses.
3. **Main Memory:** Simulated as an infinite storage resource with high latency.

2.2 Cache Parameters

Each cache level is defined by the following configurable parameters:

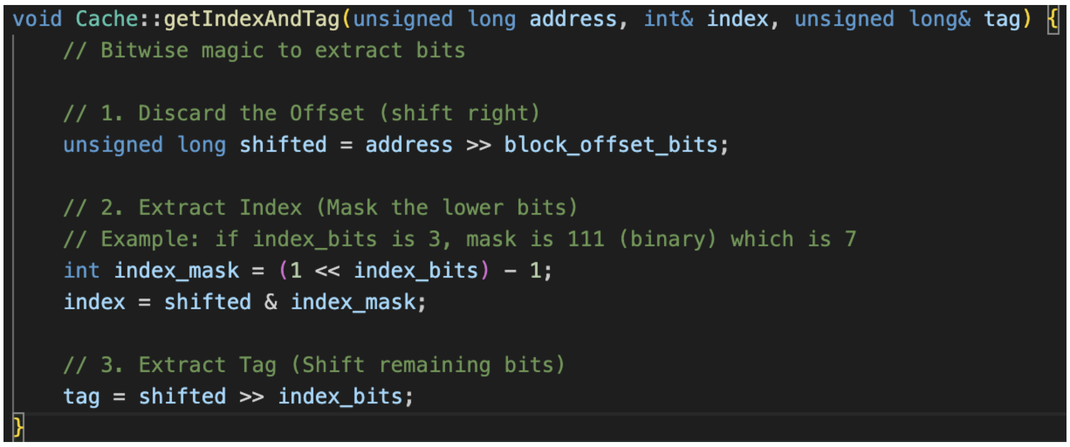
- **Total Size:** The total capacity of the cache in bytes (e.g., 32 KB).
- **Block Size:** The size of a single data unit (cache line) in bytes (e.g., 64 bytes).
- **Associativity:** The number of cache lines per set (e.g., 2-way or 4-way set associative).

3. Detailed Component Design

3.1 Addressing Scheme

The system utilizes a standard N-way Set Associative addressing model. A 32-bit memory address is bit-masked and shifted to extract three distinct components:

- **Tag:** Uniquely identifies a memory block within a specific set to distinguish between aliases.
- **Index:** Determines which specific Set (row) of the cache the address maps to.
- **Offset:** (Simulated) Represents the specific byte within a cache block.

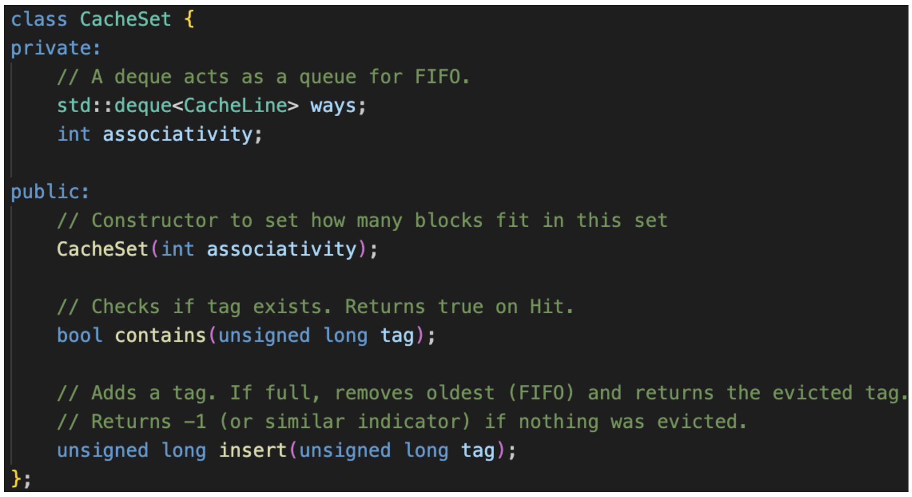


```cpp
void Cache::getIndexAndTag(unsigned long address, int& index, unsigned long& tag) {
    // Bitwise magic to extract bits

    // 1. Discard the Offset (shift right)
    unsigned long shifted = address >> block_offset_bits;

    // 2. Extract Index (Mask the lower bits)
    // Example: if index_bits is 3, mask is 111 (binary) which is 7
    int index_mask = (1 << index_bits) - 1;
    index = shifted & index_mask;

    // 3. Extract Tag (Shift remaining bits)
    tag = shifted >> index_bits;
}
```

Fig 2 : The `getIndexAndTag` function in `src/cache/cache.cpp`

3.2 Storage Structure

- **Cache Line:** The fundamental unit of storage, containing the Tag and a Valid Bit. Actual data payload is abstracted as the simulation focuses on hit/miss logic.
- **Cache Set:** A collection of $N$ cache lines (where $N$ is the associativity).
- **Container Implementation:** Each Cache Set is implemented using a Double-Ended Queue (std::deque) to efficiently manage the order of elements for the replacement policy.



```cpp
class CacheSet {
private:
    // A deque acts as a queue for FIFO.
    std::deque<CacheLine> ways;
    int associativity;

public:
    // Constructor to set how many blocks fit in this set
    CacheSet(int associativity);

    // Checks if tag exists. Returns true on Hit.
    bool contains(unsigned long tag);

    // Adds a tag. If full, removes oldest (FIFO) and returns the evicted tag.
    // Returns -1 (or similar indicator) if nothing was evicted.
    unsigned long insert(unsigned long tag);
};
```

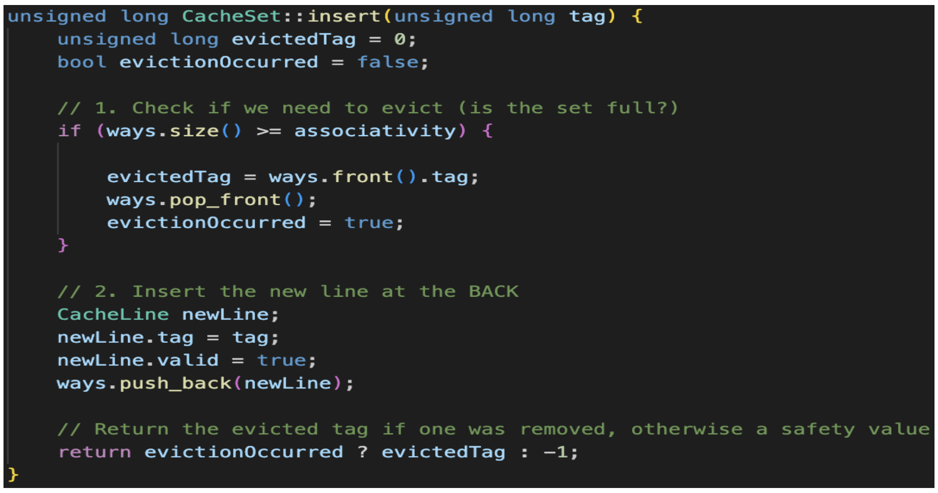
Fig 3 : The `CacheSet` class definition in `src/cache/cache_set.h`

4. Algorithms and Policies

4.1 Replacement Policy: First-In, First-Out (FIFO)

The simulator employs a FIFO replacement policy to manage cache evictions when a set reaches full capacity.

- **Insertion:** New memory blocks are appended to the back of the queue.
- **Eviction:** When the set is full, the block at the front of the queue (the oldest resident) is removed.
- **Access Update:** Unlike Least Recently Used (LRU) policies, accessing an existing block does not alter its position in the queue.



```cpp
unsigned long CacheSet::insert(unsigned long tag) {
    unsigned long evictedTag = 0;
    bool evictionOccurred = false;

    // 1. Check if we need to evict (is the set full?)
    if (ways.size() >= associativity) {

        evictedTag = ways.front().tag;
        ways.pop_front();
        evictionOccurred = true;
    }

    // 2. Insert the new line at the BACK
    CacheLine newLine;
    newLine.tag = tag;
    newLine.valid = true;
    ways.push_back(newLine);

    // Return the evicted tag if one was removed, otherwise a safety value
    return evictionOccurred ? evictedTag : -1;
}
```

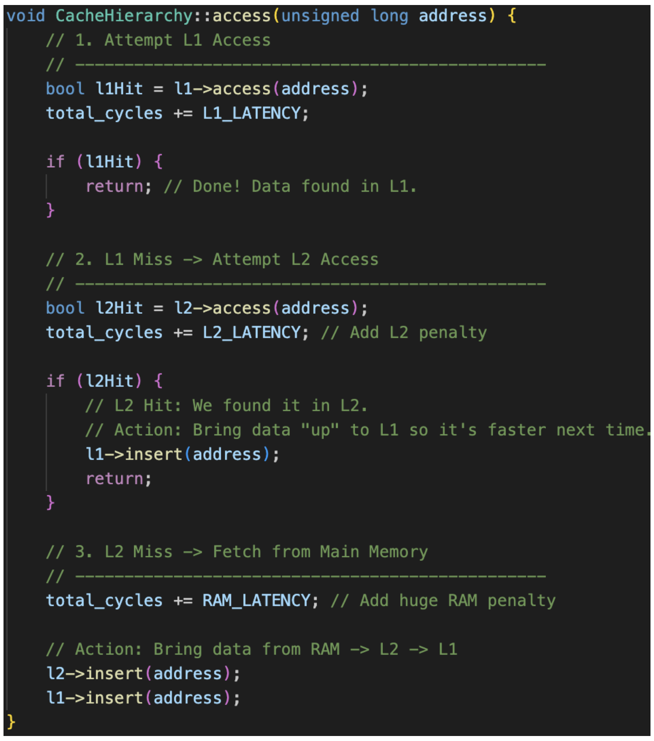
Fig 4 : The `insert` function in `src/cache/cache_set.cpp`

4.2 Read Logic and Data Flow

The data retrieval process follows a strict "Look Aside" flow:

1. **L1 Access:** The controller queries the L1 cache.

    - *Hit:* Request satisfied; metrics updated.
    - *Miss:* Request propagates to L2.
2. **L2 Access:** The controller queries the L2 cache.
    - *Hit:* Data is found in L2. The block is then inserted into L1 (Write-Allocate/Backfill) to ensure future L1 hits.
    - *Miss:* Request propagates to Main Memory.
3. **Memory Fetch:** A high-latency penalty is incurred. The data block is fetched and inserted into **both** L2 and L1, populating the hierarchy for subsequent accesses.



```cpp
void CacheHierarchy::access(unsigned long address) {
    // 1. Attempt L1 Access
    // ------------------------------------------------
    bool l1Hit = l1->access(address);
    total_cycles += L1_LATENCY;

    if (l1Hit) {
        return; // Done! Data found in L1.
    }

    // 2. L1 Miss -> Attempt L2 Access
    // ------------------------------------------------
    bool l2Hit = l2->access(address);
    total_cycles += L2_LATENCY; // Add L2 penalty

    if (l2Hit) {
        // L2 Hit: We found it in L2.
        // Action: Bring data "up" to L1 so it's faster next time.
        l1->insert(address);
        return;
    }

    // 3. L2 Miss -> Fetch from Main Memory
    // ------------------------------------------------
    total_cycles += RAM_LATENCY; // Add huge RAM penalty

    // Action: Bring data from RAM -> L2 -> L1
    l2->insert(address);
    l1->insert(address);
}
```

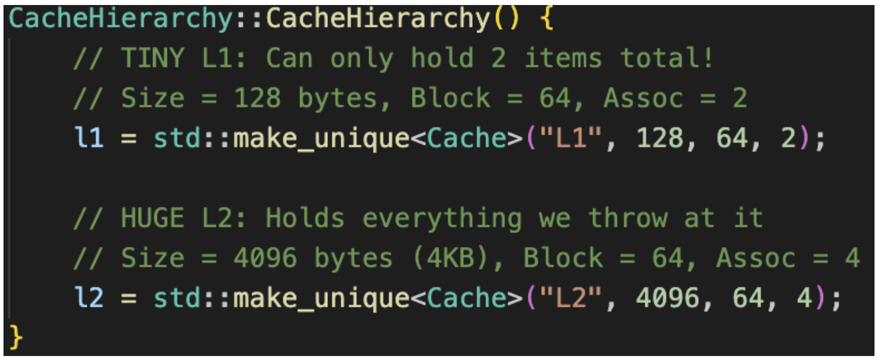
Fig 5 : The `access` function in `src/hierarchy/cache_hierarchy.cpp`

5. Configuration and Validation

5.1 Hierarchy Test Configuration

To rigorously validate the eviction logic and multilevel data propagation, the simulator was configured with a specific "stress test" topology designed to force frequent cache misses:

- **L1 Cache (Constrained):** Configured with minimal capacity (128 bytes, 2 cache lines). This forces immediate contention and ensures that new data frequently displaces older data.
- **L2 Cache (Backstop):** Configured with larger capacity (4 KB). This ensures that data evicted from L1 remains resident in L2, allowing the system to demonstrate the "L1 Miss / L2 Hit" recovery scenario.



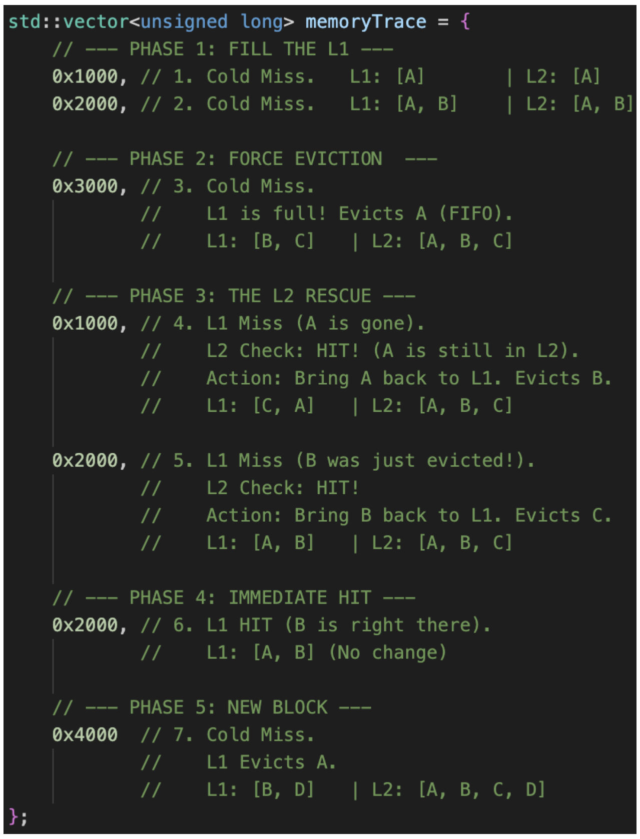
```cpp
CacheHierarchy::CacheHierarchy() {
    // TINY L1: Can only hold 2 items total!
    // Size = 128 bytes, Block = 64, Assoc = 2
    l1 = std::make_unique<Cache>("L1", 128, 64, 2);

    // HUGE L2: Holds everything we throw at it
    // Size = 4096 bytes (4KB), Block = 64, Assoc = 4
    l2 = std::make_unique<Cache>("L2", 4096, 64, 4);
}
```

Fig 6 : The `CacheHierarchy` constructor in `src/hierarchy/cache_hierarchy.cpp`

5.2 Trace-Driven Verification

The system functionality was verified using a deterministic memory trace designed to induce **Thrashing** (resource contention). The validation sequence involves accessing three competing memory addresses (A, B, and C) within a cache set capable of holding only two blocks:

1. **Saturation:** The L1 cache is filled to capacity with initial addresses.
2. **Forced Eviction:** A third unique address is accessed, triggering the FIFO policy to evict the oldest resident block.
3. **Hierarchy Recovery:** The evicted address is immediately re-accessed. The system correctly reports an L1 Miss and an L2 Hit, proving that data was successfully retained in the lower hierarchy level and correctly "backfilled" to L1 upon re-access.



```cpp
std::vector<unsigned long> memoryTrace = {
    // --- PHASE 1: FILL THE L1 ---
    0x1000, // 1. Cold Miss.   L1: [A]       | L2: [A]
    0x2000, // 2. Cold Miss.   L1: [A, B]    | L2: [A, B]

    // --- PHASE 2: FORCE EVICTION  ---
    0x3000, // 3. Cold Miss.
            //    L1 is full! Evicts A (FIFO).
            //    L1: [B, C]   | L2: [A, B, C]

    // --- PHASE 3: THE L2 RESCUE ---
    0x1000, // 4. L1 Miss (A is gone).
            //    L2 Check: HIT! (A is still in L2).
            //    Action: Bring A back to L1. Evicts B.
            //    L1: [C, A]   | L2: [A, B, C]

    0x2000, // 5. L1 Miss (B was just evicted!).
            //    L2 Check: HIT!
            //    Action: Bring B back to L1. Evicts C.
            //    L1: [A, B]   | L2: [A, B, C]

    // --- PHASE 4: IMMEDIATE HIT ---
    0x2000, // 6. L1 HIT (B is right there).
            //    L1: [A, B] (No change)

    // --- PHASE 5: NEW BLOCK ---
    0x4000  // 7. Cold Miss.
            //    L1 Evicts A.
            //    L1: [B, D]   | L2: [A, B, C, D]
};
```

Fig 7 : The `memoryTrace` vector initialization in `src/main.cpp`

6. Conclusion

The Multilevel Cache Simulator successfully models the fundamental behavior of a hierarchical CPU memory subsystem. By implementing a modular architecture separating storage (`CacheSet`), addressing logic (`Cache`), and hierarchy orchestration (`CacheHierarchy`), the system provides a robust platform for analyzing memory access patterns and latency penalties.

While the current implementation abstracts away data payloads and write policies to focus on read-access behavior, the object-oriented design allows for seamless future extensibility. Potential enhancements include the integration of Least Recently Used (LRU) replacement algorithms, the implementation of Write-Back policies with dirty-bit tracking, and the support for larger, complex trace files to simulate real-world computing workloads. The project fulfills its primary objective of demonstrating the critical role of cache hierarchies in bridging the speed gap between high-performance CPUs and main memory.