Implementation of A Memory Management Simulator

Shubham Kataria, 23114092

1. Introduction

This report describes the design, implementation, and main functionalities of a custom memory allocator combined with a configurable multi-level cache simulator. The system exposes a text-based command-line interface (CLI) that allows users to experiment with allocation strategies, fragmentation, and cache hierarchy performance in a controlled environment.
The project replaces the standard heap allocator with a fixed-size simulated heap and models how memory accesses interact with a cache hierarchy. It is intended as an educational tool and a playground for studying memory management and cache behaviour.

2. System Overview

The system consists of three main components:

1. Heap Allocator
    - Manages a fixed 64 KB heap.
    - Supports multiple placement strategies: first-fit, best-fit, worst-fit.
    - Maintains a linked list of heap blocks, each tagged with metadata.

2. Multi-Level Cache Simulator
    - Models an arbitrary number of cache levels (L1, L2, …).
    - Each level has configurable capacity, block size, associativity, and latency.
    - Uses an inclusive cache model and an LFU+LRU replacement policy.
    - Tracks detailed per-level and global statistics.

3. CLI Front-End
    - Provides user commands to allocate/free memory, read/write data, configure the cache, and inspect statistics.
    - Exposes memory-safety checks and cache behaviour interactively.

3. Heap Allocator Design

3.1 Heap Model

- The allocator manages a single contiguous heap region of 64 KB in user space.
- Memory is organized as a linked list of blocks, each consisting of:
    - A header with metadata (size, allocation status, links to neighbours, and optionally bookkeeping data).
    - A payload region visible to the user..

3.2 Block Metadata and Alignment

- Each block header stores:
    - Block size (excluding header).
    - Allocation flag (allocated/free).
    - Pointers to the next/previous blocks.
- Allocations are aligned to std::max_align_t, ensuring the payload is suitable for any primitive type.
- When a free block is larger than needed, it may be split into:
    - An allocated block of the requested size.
    - A remainder block marked free.

3.3 Allocation Strategies

The allocator supports three fit strategies, selectable per allocation request:

1. First-Fit
    - Traverses blocks from the heap start.
    - Selects the first free block whose size is at least the requested size.
    - Typically fast, but can leave small gaps and increase external fragmentation over time.
2. Best-Fit
    - Examines all free blocks and chooses the smallest one that still satisfies the request.
    - Aims to minimize leftover space in the chosen block.
    - Can reduce external fragmentation but may be slower due to the need to inspect more blocks.
3. Worst-Fit
    - Selects the largest free block available.
    - Attempts to preserve fewer but larger contiguous free regions, which can be beneficial for future large allocations.
    - Can increase internal fragmentation due to oversized allocations.

The CLI malloc command accepts a size and an optional strategy specifier, mapping to these internal policies.

3.4 Freeing and Coalescing

When a block is freed:

- Its allocation flag is cleared.
- The allocator checks neighbouring blocks:
    - If adjacent blocks are free, they are coalesced into a single larger free block.
- Coalescing reduces external fragmentation and keeps the free list compact.

4. Safety and Debugging Features

The allocator includes pattern-based safety checks to detect misuse:

- On allocation, payload bytes are filled with an “uninitialized” pattern.
- On free, payload bytes are overwritten with a distinct “freed” pattern.
- During reads:
    - If any byte in the requested range still matches an uninitialized or freed pattern, the read is rejected.
- During writes:
    - Writes are only allowed to valid, allocated regions.
    - Each written byte is treated as a cache access to update cache statistics.

These patterns help simulate and detect:

- Use-after-free errors.
- Reads from uninitialized memory.
- Accidental overlap and misuse of freed blocks.

The read command prints the accessed data both as ASCII (with non-printables replaced by .) and as hexadecimal bytes for easier inspection.

5. Multi-Level Cache Simulator Design

5.1 Cache Hierarchy Model

The cache simulator models a hierarchy of levels (L1, L2, …) in front of the heap:

- Each level caches blocks (cache lines) from the underlying level or main memory.
- The hierarchy is inclusive:
    - Data stored in an upper level must also reside in all lower levels that participated in the fetch.

- Main memory is assumed to have a configurable fixed access latency.

5.2 Per-Level Configuration

Each cache level is defined by:

- Total size in bytes.
- Block size (line size) in bytes.
- Associativity:
    - From direct-mapped (1-way) to fully associative.
- Access latency in cycles.

These parameters determine the number of sets and ways per set.

5.3 Replacement Policy: LFU + LRU

Within each cache level:

- When a set is full and a new line must be inserted, the replacement policy is:
    - LFU (Least Frequently Used) as the primary criterion.
    - LRU (Least Recently Used) for tie-breaking between lines with equal frequency.
- Each line maintains:
    - An access count (frequency).
    - A recency indicator (e.g., last access timestamp or counter).

This hybrid policy balances persistent hot lines against recently used ones.

5.4 Access Flow

Every allocator read and write is translated into a cache access:

- The address and size are mapped to one or more cache blocks.
- For each block:
    - The access starts at L1 and proceeds down the hierarchy until:
        - A hit is found, or
        - All cache levels miss and main memory is accessed.
- On a miss:
    - The block is fetched from the level (or memory) that contains it.
    - The fetched line is filled into all intermediate levels (inclusive policy).
- Latencies from each level and main memory are accumulated to compute the total penalty for each access.

6. Statistics and Instrumentation

The system maintains detailed statistics for both the allocator and the cache hierarchy.

6.1 Allocator Statistics

The stats command reports:

- Total used bytes and free bytes.
- Number of allocated and free blocks.
- Internal fragmentation (wasted space inside allocated blocks due to alignment or over-allocation).
- External fragmentation (degree to which free memory is split into small pieces).
- Size of the largest free block.
- Number of allocation requests, both successful and failed.
- Overall heap utilization percentage.

These metrics help analyze the impact of different fit strategies on fragmentation and efficiency.

6.2 Cache Statistics

For each cache level, the simulator tracks:

- Number of accesses.
- Number of hits and misses.
- Hit ratio per level.
- Average miss penalty propagated to lower levels or main memory.
- Configuration summary:
    - Size, block size, associativity, latency, and number of sets.

Global statistics across the entire hierarchy include:

- Total accesses, hits, and misses.
- Overall hit ratio.
- Average access penalty in cycles per access.

These statistics are printed when:

- The stats command is issued in the allocator prompt.
- The “Dump cache statistics” option is selected in the cache menu.

7. Command-Line Interface

The CLI is the main entry point for interacting with the system. At startup, users see an allocator> prompt, with commands such as:

- malloc <size> [strategy] – Allocate a block with chosen strategy.
- free <id> – Free a previously allocated block.
- dump – Show a detailed view of heap block layout, offsets, sizes, and states.

- stats – Display allocator and cache statistics.
- read <id> <off> <size> – Safely read bytes from a block (with pattern checks).
- write <id> <off> <data...> – Write an ASCII string into a block (with cache accesses).
- cache – Enter the cache configuration submenu.
- help – List available commands.
- exit / quit – Terminate the program.

7.1 Cache Configuration Menu

Within the allocator>cache> submenu, users can:

- Initialize a default multi-level cache (e.g., small fast L1, larger slower L2).
- Reset the cache to no levels, clearing all stats.
- Add a new cache level by specifying size, block size, associativity, and latency.
- Reconfigure an existing cache level by index.
- Dump cache statistics.

This menu allows users to quickly experiment with different cache hierarchies and immediately observe their impact on performance.

8. Typical Usage Workflow

- Start the program and optionally configure the cache (either via defaults or manual setup).
- Allocate one or more blocks using malloc <size> [strategy].
- Use write and read to generate realistic access patterns and trigger cache activity.
- Use dump to inspect the internal heap structure and block layout.
- Use stats to evaluate:
    - Heap utilization and fragmentation under different strategies.
    - Cache hit/miss behaviour and average access penalties across cache levels.

This workflow enables systematic exploration of how allocator design and cache configuration interact to influence performance and memory efficiency.

9. Conclusion

The custom memory allocator and multi-level cache simulator provide an integrated environment for studying dynamic memory management and cache behaviour. By

decoupling the allocator from the system heap and exposing a configurable cache hierarchy, the project makes it possible to:

- Compare allocation strategies (first-fit, best-fit, worst-fit).
- Observe internal and external fragmentation.
- Explore how different cache configurations affect hit ratios and access latency.
- Understand the practical impact of memory-safety violations via pattern-based checks.

The modular design and CLI interface make the system suitable both as an educational tool and as a foundation for further experimentation with more advanced allocation policies or cache models.