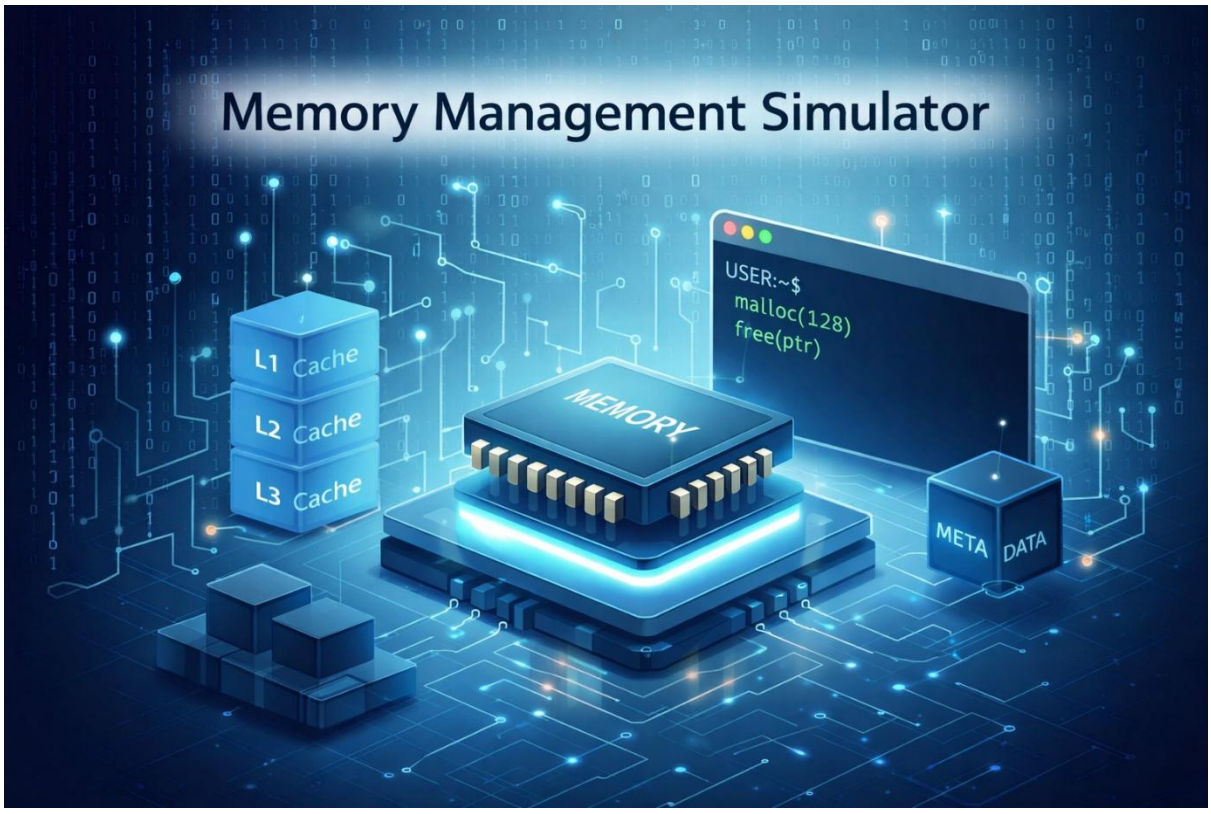
ACM OPEN PROJECT



Shivam Kumar

EE 2Y

24115135

1. Introduction

Memory management is a fundamental responsibility of an operating system, involving the allocation and deallocation of memory while minimizing fragmentation and ensuring efficient access. Understanding how different allocation strategies and cache mechanisms work is essential for studying system-level performance and design trade-offs.

This project presents the design and implementation of a **Memory Management Simulator** that models **physical memory allocation** and **CPU cache behavior** in a controlled, user-space environment. The simulator implements multiple allocation strategies, including **First Fit, Best Fit, Worst Fit**, and a **Buddy Allocation System**, allowing comparative analysis of their behavior and fragmentation characteristics.

In addition to memory allocation, the simulator includes a **three-level cache hierarchy (L1, L2, and L3)** with configurable associativity, replacement policies, and access latencies. A command-line interface (CLI) is provided to interactively allocate memory, free memory, perform read and write operations, and observe cache performance and memory statistics.

The simulator operates strictly on **physical memory addresses**. Advanced concepts such as virtual memory, paging, page tables, and address translation are **not implemented** in this project and are considered future extensions.

---

2. System Architecture Overview

2.1 High-Level Architecture

The simulator follows a modular architecture that mirrors the logical flow of memory operations in real systems. User commands are processed by a CLI, which coordinates memory allocation, memory access, and statistics collection.

A key design principle of the simulator is the **clear separation between memory allocation and memory access**. Allocation requests reserve space in the heap, while cache behaviour is triggered **only during read and write operations**, accurately reflecting real CPU behaviour.

The overall flow is:

1. User issues a command through the CLI.
2. The allocator reserves or frees physical memory.
3. Read/write commands resolve allocation IDs to physical addresses.

4. Cache levels (L1 → L2 → L3) are accessed in order.
5. Statistics are updated based on the operation.

2.2 Module Responsibilities

- **CLI Module:** Handles user input, command parsing, and validation of allocation IDs.
- **Allocator Module:** Implements FIT-based strategies and the Buddy allocator for managing physical memory.
- **Cache Module:** Simulates multilevel caching with hit/miss tracking, replacement policies, and latency modeling.
- **Statistics Module:** Computes memory utilization, fragmentation, allocation success rate, and cache metrics.
- **Memory Dump Module:** Provides a detailed visualization of free and allocated memory blocks.

This modular separation improves clarity and allows individual components to be modified or extended independently.

---

3. Physical Memory Model

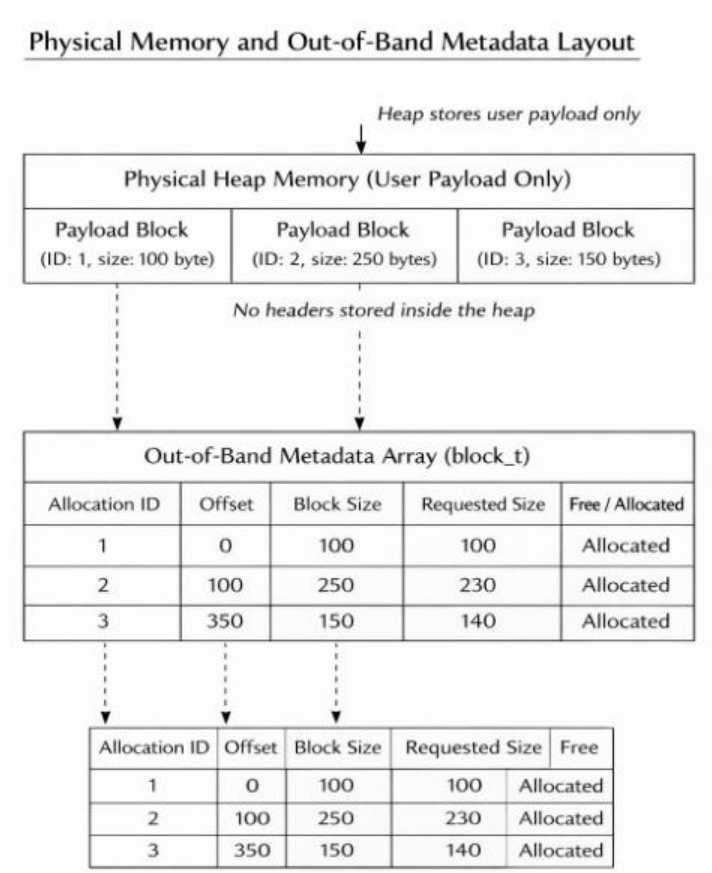
3.1 Heap Design

The simulator models physical memory as a **contiguous, byte-addressable heap** allocated at runtime. The heap size is specified by the user during initialization. The heap represents raw physical memory and does not store actual application data; instead, it is used to simulate allocation behaviour.

The system assumes:

- A single process
- A single address space
- No concurrency
- No virtual memory abstraction

3.2 Out-of-Band Metadata



For First Fit, Best Fit, and Worst Fit allocation strategies, the simulator uses **out-of-band metadata**. This means that no headers or control information are stored inside the heap itself. Instead, each memory block is described by a separate metadata structure containing:

- Offset within the heap
- Block size
- Requested size
- Allocation status
- Allocation ID

This design ensures that the heap contains only user payload, simplifies fragmentation analysis, and enables accurate memory dumps.

3.3 ID-Based Allocation

Each successful allocation returns a **unique allocation ID** instead of a raw memory pointer. During read or write operations, this ID is resolved internally to a physical address using allocator metadata.

This abstraction improves safety, prevents invalid memory access, and allows the simulator to consistently track and validate all memory operations across different allocation strategies.

---

4. Memory Allocation Strategies (First Fit, Best Fit, Worst Fit)

The simulator implements three classical **FIT-based dynamic memory allocation strategies**: First Fit, Best Fit, and Worst Fit. All three strategies operate on the same physical memory model and use **out-of-band metadata** to track memory blocks.

Each memory block is represented using a metadata structure that stores its offset in the heap, size, allocation status, and allocation ID. When a memory request is made, the allocator searches this metadata to find a suitable free block.

4.1 First Fit Allocation

The **First Fit** strategy scans the list of memory blocks from the beginning and selects the first free block that is large enough to satisfy the request. This approach is fast because it stops searching as soon as a suitable block is found.

However, First Fit may lead to **external fragmentation**, as smaller free blocks tend to accumulate near the beginning of the heap over time.

4.2 Best Fit Allocation

The **Best Fit** strategy searches the entire list of free blocks and selects the smallest block that can accommodate the requested size. This minimizes wasted space within each allocation.

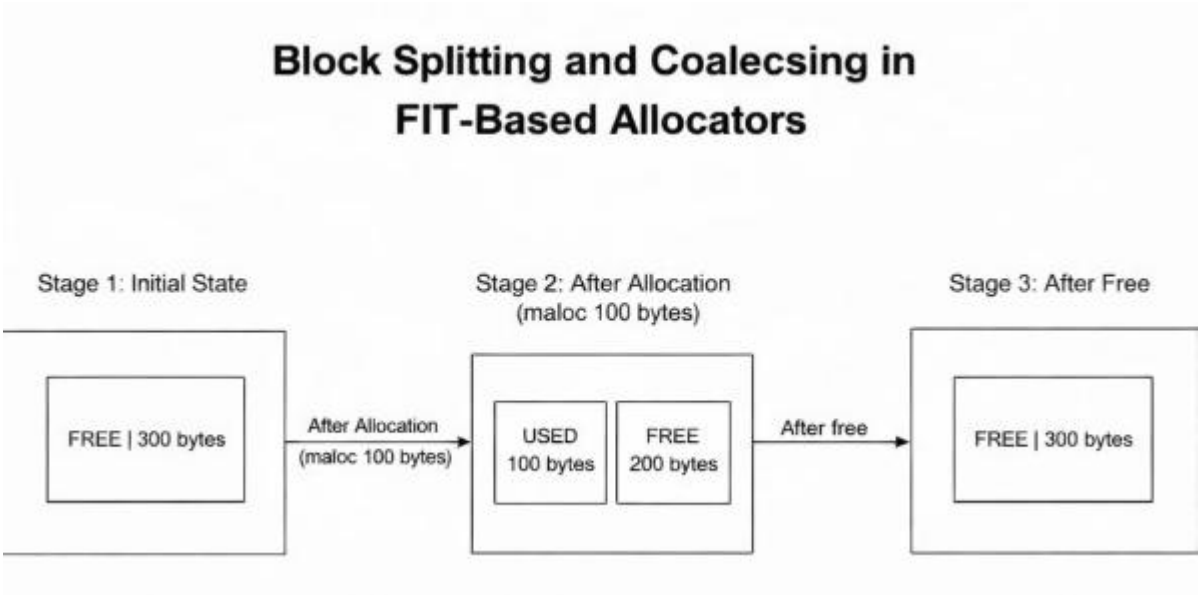
Although Best Fit can reduce unused space within allocated blocks, it requires scanning the full metadata list, making it slower than First Fit. It can also increase external fragmentation by creating many small free blocks.

4.3 Worst Fit Allocation

The **Worst Fit** strategy allocates memory from the largest available free block. The idea is to leave large free blocks available for future allocations.

In practice, Worst Fit often performs poorly, as it breaks large blocks into smaller fragments, which may not be efficiently reused later.

4.4 Block Splitting and Coalescing



When a selected free block is larger than the requested size, the allocator **splits** the block into:

- An allocated block of the requested size
- A remaining free block

When memory is freed, the allocator performs **coalescing** by merging adjacent free blocks. Both forward and backward coalescing are implemented to reduce external fragmentation.

Because all metadata is stored out-of-band and block sizes match requested sizes exactly, **internal fragmentation is zero** for FIT-based allocators.

---

5. Buddy Allocation System

To address the limitations of FIT-based allocators, the simulator also implements a **Buddy Allocation System**, which manages memory using power-of-two block sizes.

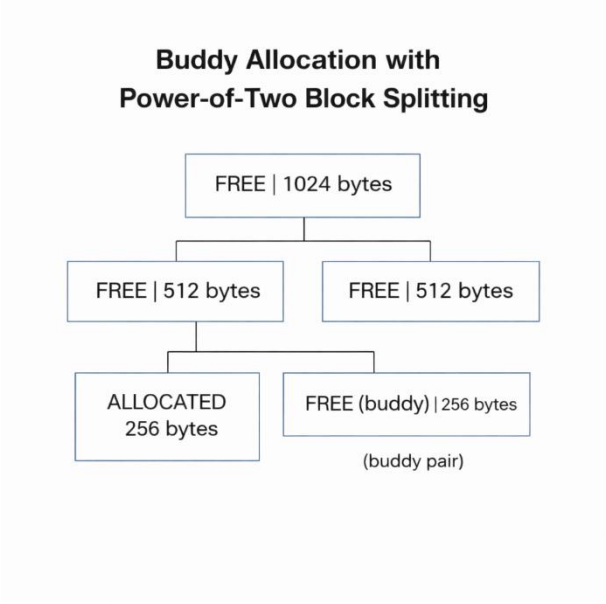
5.1 Design Overview

In the Buddy system, memory is divided into blocks whose sizes are powers of two. When a memory request is made, the requested size is rounded up to the nearest power of two. This approach enables fast allocation and predictable block management.

Unlike FIT-based allocators, the Buddy system stores a **small header at the start of every block**, which tracks:

- Allocation ID
- Block order ($\log_2$ of block size)
- Requested size

5.2 Allocation Mechanism



If a free block of the required size is not available, a larger block is recursively **split into two equal-sized "buddy" blocks** until the desired block size is obtained. Free blocks are managed using separate free lists for each block order.

This process allows memory to be allocated efficiently, even when exact-sized blocks are not initially available.

5.3 Deallocation and Fragmentation

When a block is freed, it is returned to the appropriate free list. The simulator tracks **internal fragmentation**, which occurs because block sizes are rounded up to powers of two.

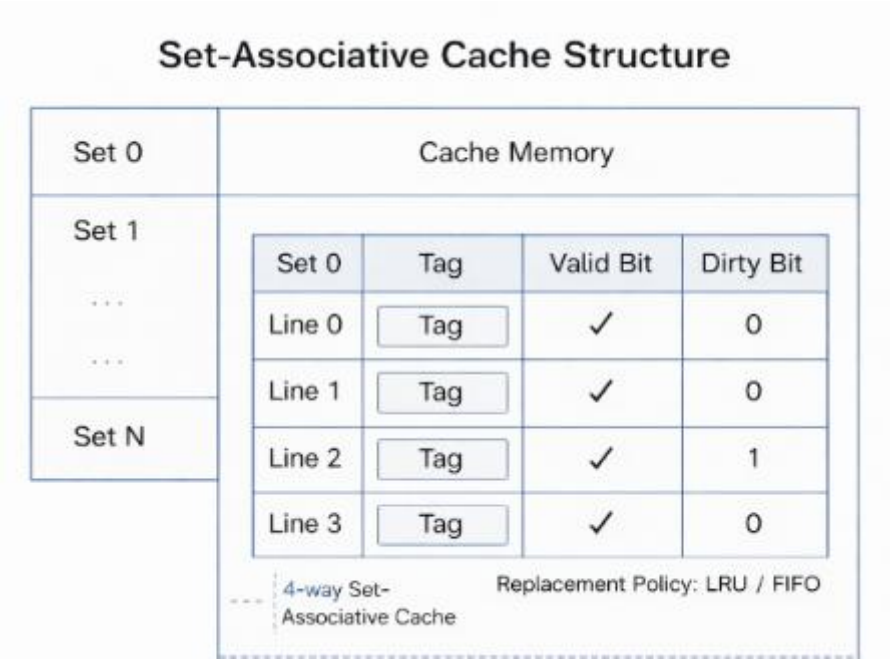
While the Buddy system reduces external fragmentation and offers fast allocation and deallocation, it trades this advantage for **internal fragmentation**, especially for allocation sizes that are not close to a power of two.

---

6. Cache Simulation

To model realistic memory access behaviour, the simulator includes a **multilevel CPU cache simulation**. The cache subsystem is independent of the memory allocator and is accessed **only during read and write operations**, not during memory allocation.

6.1 Cache Architecture



The simulator implements a **three-level unified cache hierarchy** consisting of L1, L2, and L3 caches. Each cache level is modelled as a **set-associative cache** with configurable parameters such as cache size, block size, associativity, and replacement policy.

The configuration used in the simulator is:

- **L1 Cache:** 1 KB, 2-way set associative, LRU replacement
- **L2 Cache:** 4 KB, 4-way set associative, LRU replacement
- **L3 Cache:** 16 KB, 8-way set associative, FIFO replacement
- **Block size:** 64 bytes (common across all levels)

Each cache level consists of multiple sets, and each set contains a fixed number of cache lines based on the associativity.

6.2 Address Mapping

For a given physical memory address, the cache divides the address into three components:

- **Block Offset:** Identifies the byte within a cache block
- **Set Index:** Determines which cache set is accessed
- **Tag:** Used to identify whether the desired block is present in the cache

The set index and tag are computed using the block size and the number of sets in the cache. This mapping allows efficient lookup and replacement within each cache level.

6.3 Replacement Policies

The simulator supports two cache replacement policies:

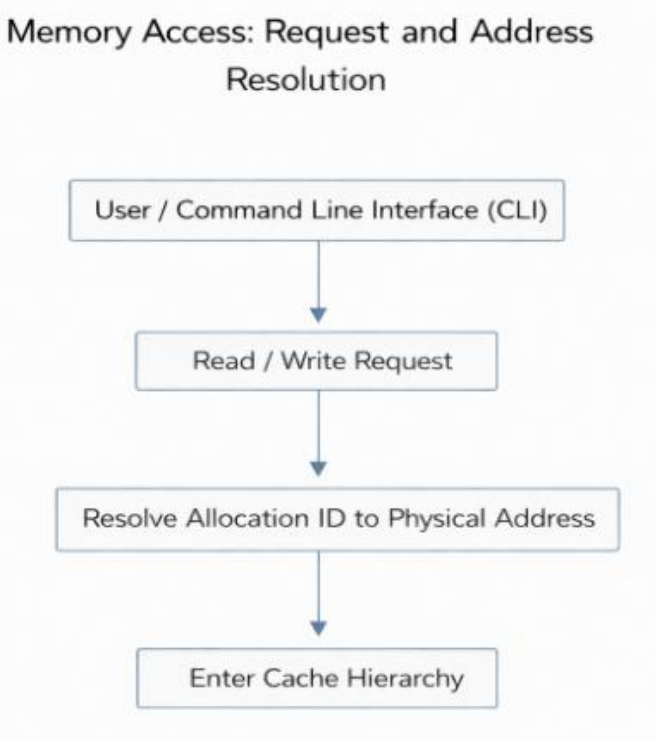
- **Least Recently Used (LRU):** The cache line that has not been accessed for the longest time is replaced.
- **First-In-First-Out (FIFO):** The cache line that was inserted earliest is replaced.

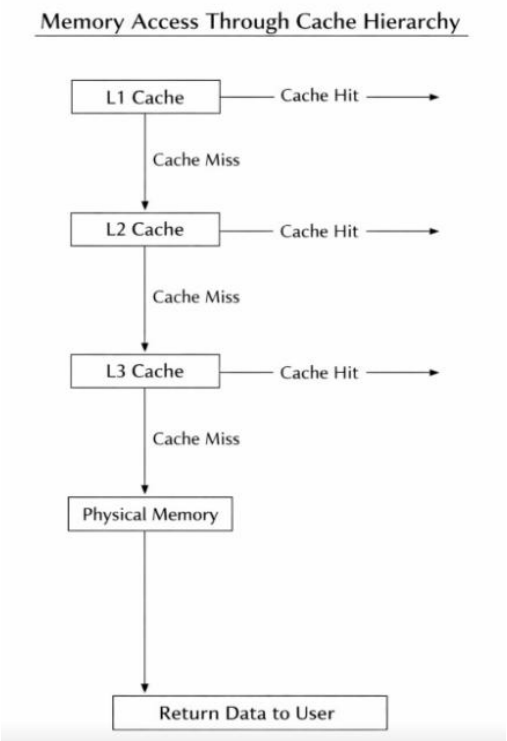
LRU is used for L1 and L2 caches to prioritize frequently accessed data, while FIFO is used for L3 to reduce management overhead.

6.4 Write Policy

The cache follows a **write-back policy**. On a write hit, the cache line is updated and marked dirty without immediately updating main memory. When a dirty line is evicted, the write-back to main memory is simulated. This approach reduces memory traffic and reflects the behavior of modern processors.

6.5 Memory Access Flow and Latency Modelling





During a read or write operation, the cache is accessed in the following order:

1. L1 Cache
2. L2 Cache
3. L3 Cache
4. Main Memory (on a cache miss at all levels)

Each cache level has an associated access latency:

- L1: 1 cycle
- L2: 5 cycles
- L3: 20 cycles
- Main Memory: 100 cycles

The simulator accumulates these latencies to compute the **total access cost** and tracks the **Average Memory Access Time (AMAT)** across all requests.

7. Memory Access Flow and Command-Line Interface

The simulator provides an interactive **command-line interface (CLI)** that allows users to perform memory allocation, deallocation, and memory access operations. The CLI acts as the primary interface between the user and the internal memory management and cache subsystems.

7.1 Allocation and Deallocation via CLI

Memory allocation is performed using the malloc <size> command. Upon successful allocation, the allocator returns a **unique allocation ID**, which is stored by the CLI for validation. This ID is later used for freeing memory or performing read/write operations.

Deallocation is handled using the free <id> command. Before freeing a block, the CLI verifies that the given ID corresponds to an active allocation, preventing invalid or duplicate free operations.

7.2 ID-to-Address Resolution

For memory access operations, the CLI internally resolves the allocation ID to a **physical memory address**. The resolution mechanism depends on the currently selected allocation strategy:

- For FIT-based allocators, the address is computed using the heap base and the block offset stored in metadata.
- For the Buddy allocator, a helper function is used to locate the block and return the payload address.

This abstraction ensures that users do not directly interact with raw memory addresses.

7.3 Read and Write Operations

The simulator supports memory access using the read <id> <offset> and write <id> <offset> commands. These commands simulate CPU memory accesses by computing a physical address from the allocation ID and offset.

Once the physical address is determined, it is passed to the cache subsystem. The cache hierarchy (L1 → L2 → L3) is accessed sequentially, and the corresponding hit or

miss behavior is recorded. The total access latency is computed based on the cache levels involved.

Importantly, **cache access occurs only during read and write operations**. Memory allocation and deallocation do not trigger cache activity.

7.4 Observability and Output

The CLI provides several commands for observing the internal state of the simulator:

- dump: Displays the current memory layout, showing free and allocated blocks.
- stats: Prints memory allocation statistics and fragmentation metrics.
- cache_stats: Displays cache hit/miss counts, total cycles, and AMAT.

These features allow users to analyze the behavior of different allocation strategies and observe the impact of caching on memory access performance.

---

8. Statistics and Performance Metrics

To evaluate the behaviour of different allocation strategies and cache configurations, the simulator maintains detailed **runtime statistics**. These metrics provide insight into memory utilization, fragmentation, allocation efficiency, and cache performance.

8.1 Memory Allocation Metrics

The simulator tracks the total number of allocation requests, successful allocations, failed allocations, and deallocation operations. Based on these values, an allocation success rate is computed.

For FIT-based allocators, memory usage statistics are derived directly from the out-of-band metadata. The simulator reports:

- Total heap size
- Used memory
- Free memory
- Number of used and free blocks
- Memory utilization percentage

Since block sizes exactly match requested sizes in FIT allocators, **internal fragmentation is zero**. External fragmentation is calculated using the ratio between the largest free block and the total free memory.

8.2 Buddy Allocator Statistics

For the Buddy allocation system, statistics are computed by traversing the heap using block headers. In addition to memory usage and allocation counts, the simulator tracks **internal fragmentation**, which arises due to rounding allocation sizes to the nearest power of two.

The following metrics are reported:

- Internal fragmentation (in bytes)
- External fragmentation
- Memory utilization
- Used and free block counts

This enables a direct comparison between FIT-based allocators and the Buddy allocator in terms of fragmentation trade-offs.

8.3 Cache Performance Metrics

The cache subsystem records hits and misses at each cache level (L1, L2, and L3). Using these values, the simulator computes hit rates for individual cache levels.

Additionally, the simulator tracks the total number of memory access requests and the total number of cycles consumed. From these values, the **Average Memory Access Time (AMAT)** is calculated, providing a quantitative measure of cache effectiveness.

8.4 Observability and Analysis

Statistics can be viewed at any time using CLI commands such as stats and cache_stats. These outputs allow users to observe how different allocation strategies and access patterns affect memory fragmentation and cache performance.

---

9. Limitations and Future Work

While the memory management simulator successfully models physical memory allocation and cache behaviour, it has certain limitations due to scope and design choices.

9.1 Limitations

- **No Virtual Memory Support:**
  The simulator operates only on physical memory addresses. Concepts such as paging, page tables, address translation, and page faults are not implemented.
- **Single Process Model:**
  The system assumes a single process and does not support multiple address spaces or context switching.
- **Symbolic Data Handling:**
  The simulator does not store or manipulate actual application data. Memory accesses are simulated symbolically to study allocation and cache behavior.
- **Simplified Buddy Deallocation:**
  While the Buddy allocator supports block splitting, recursive buddy merging during deallocation is not fully implemented.
- **Unified Cache Model:**
  The cache is modeled as a unified cache rather than separate instruction and data caches.

9.2 Future Work

Several extensions can be added to enhance the simulator:

- Implementation of **virtual memory** using paging and page tables
- Support for **page replacement policies** and page fault handling
- Recursive buddy merging to further reduce fragmentation
- Simulation of **multiple processes** and context switching
- Addition of a **TLB (Translation Lookaside Buffer)** for faster address translation