Course Name: ECEN 350

Section: A

Project 6: Build Memory Simulator for Direct-Mapped, Fully Associative and Set Associative Caches

Team Members

Ejmen Al-Ubejdij UIN: 333001194
Umair Gavankar UIN: 333002045

Date: April 26, 2025

Contents

1 Introduction **3**

2 Background and Theory **3**
2.1 Cache Memory Organization 3
2.2 Cache Organization Types 3
2.2.1 Direct-Mapped Cache 4
2.2.2 Fully Associative Cache 4
2.2.3 Set Associative Cache 4
2.3 Address Breakdown 5
2.4 Replacement Policies 5
2.4.1 Least Recently Used (LRU) 5
2.4.2 First-In-First-Out (FIFO) 5

3 Design and Implementation **6**
3.1 System Architecture 6
3.2 Core Simulator Implementation 6
3.2.1 Cache Block 6
3.2.2 Cache Set 7
3.2.3 Replacement Policies 7
3.2.4 Main Cache Class 8
3.3 Web Interface 8
3.3.1 Backend API 8
3.3.2 Frontend Implementation 9

4 Testing and Validation **9**
4.1 Unit Tests 10
4.2 Replacement Policy Tests 10
4.3 Edge Cases 11

5 Results and Analysis **11**
5.1 Performance Analysis 11
5.2 Hit Rate Comparison 11
5.3 Impact of Block Size 12

6 Web Interface Demonstration **12**
6.1 Configuration Panel 12
6.2 Memory Access Controls 12
6.3 Cache Visualization 12
6.4 Address Breakdown 13
6.5 Statistics Display 13

7 Challenges and Solutions **13**
7.1 Technical Challenges 13
7.1.1 Address Calculation 13
7.1.2 Replacement Policy Implementation 14
7.1.3 Visualization Performance 14

7.2 Design Considerations 14
7.2.1 Modular Architecture 14
7.2.2 Language Selection 14
7.2.3 Web Interface 14
7.2.4 Visual Feedback 15

8 Future Enhancements **15**
8.1 Additional Features 15
8.2 Performance Optimizations 15
8.3 UI Improvements 15

9 Conclusion **16**

10 References **16**

11 Appendices **16**
11.1 Appendix A: Installation and Usage Instructions 16
11.1.1 Installation 16
11.1.2 Running the Simulator 17
11.1.3 Using the Simulator 17
11.2 Appendix B: Key Code Structures 17
11.2.1 Project Structure 17
11.2.2 Core Classes 18
11.3 Appendix C: Test Results 18
11.3.1 Unit Test Results 18
11.3.2 Performance Test Results 19

1 Introduction

The objective of this project is to design and implement a cache memory simulator capable of modeling direct-mapped, fully associative, and set associative cache configurations. Cache memory serves as a critical component in modern computer architectures, bridging the performance gap between the processor and main memory by storing frequently accessed data for faster retrieval.

Our simulator provides an interactive web-based interface that allows users to configure various cache parameters, visualize memory accesses, and observe cache behavior in real-time. This tool is valuable for both educational purposes, helping students understand cache operations, and for research, enabling the analysis of different cache configurations and replacement policies.

The key features of our simulator include:

- Configurable parameters: total cache size, block size, associativity, and replacement policy
- Support for direct-mapped, set associative, and fully associative cache organizations
- Implementation of Least Recently Used (LRU) and First-In-First-Out (FIFO) replacement policies
- Real-time visualization of cache state and memory access operations
- Detailed statistics tracking for hit rates, miss rates, and evictions
- Address breakdown showing tag, index, and offset bits
- Support for both individual memory accesses and trace-based simulation

2 Background and Theory

2.1 Cache Memory Organization

Cache memory is a small, fast memory component that stores copies of data from frequently used main memory locations. When the processor needs to read from or write to a location in main memory, it first checks whether the required data is in the cache. If the data is present (cache hit), the processor can quickly access it, avoiding the need to access the slower main memory. If the data is not present (cache miss), the processor fetches it from main memory and typically stores it in the cache for future use.

2.2 Cache Organization Types

There are three primary cache organization types, each with its advantages and disadvantages:

2.2.1 Direct-Mapped Cache

In a direct-mapped cache, each memory address can map to only one specific cache line. The mapping is typically done using the modulo operation on the address with the number of cache lines. Direct-mapped caches are simple to implement and have fast access times but suffer from increased conflict misses when multiple addresses map to the same cache line.

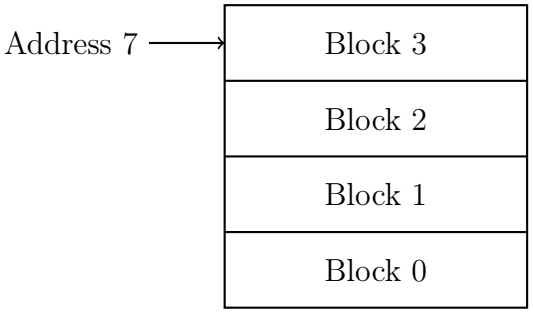

Figure 1: Direct-Mapped Cache: Each address maps to exactly one location.

2.2.2 Fully Associative Cache

In a fully associative cache, a memory block can be placed in any cache line. This organization provides maximum flexibility and minimizes conflict misses but requires comparing the address with every cache line's tag in parallel, which is hardware-intensive and can slow access times.

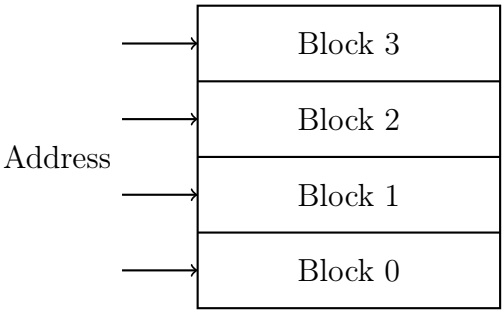

Figure 2: Fully Associative Cache: An address can map to any cache line.

2.2.3 Set Associative Cache

Set associative caches combine aspects of both direct-mapped and fully associative caches. Memory blocks are first mapped to a specific set (like direct-mapped), and within that set, the block can be placed in any way (like fully associative). An N-way set associative cache has N cache lines per set.

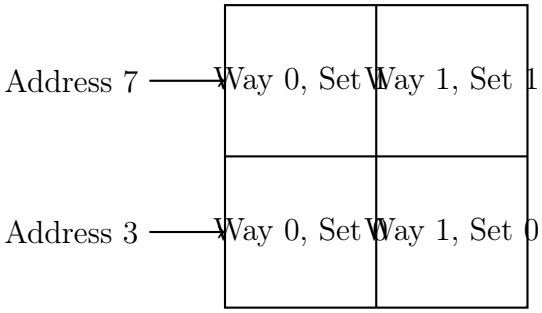

Figure 3: 2-Way Set Associative Cache: Each address maps to a specific set containing multiple ways.

2.3 Address Breakdown

A memory address in a cache system is typically divided into three parts:

- **Tag**: Used to identify the specific memory block stored in the cache line
- **Index**: Used to select the set (in set associative and direct-mapped caches)
- **Offset**: Used to select the specific byte within the cache block

Figure 4: Address Breakdown in Cache Memory

2.4 Replacement Policies

When a cache is full and a new block needs to be loaded, a replacement policy determines which existing block should be evicted. Our simulator implements two common replacement policies:

2.4.1 Least Recently Used (LRU)

The LRU policy evicts the block that has been accessed least recently. This policy works on the principle of temporal locality, which suggests that if a block has been used recently, it's likely to be used again in the near future. LRU typically performs well but requires maintaining access history for each block.

2.4.2 First-In-First-Out (FIFO)

The FIFO policy evicts the block that was loaded earliest into the cache. This policy is simpler to implement than LRU as it only requires tracking the order of block insertions, not accesses.

3 Design and Implementation

3.1 System Architecture

Our cache memory simulator is designed with a clear separation of concerns, dividing the system into core simulation logic and the user interface. This modular architecture enhances maintainability and allows for future extensions.

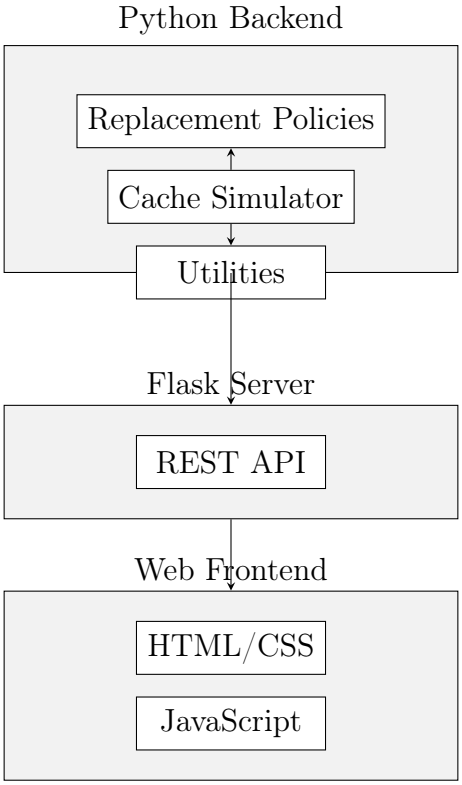

Figure 5: System Architecture of the Cache Memory Simulator

3.2 Core Simulator Implementation

The core of our simulator is implemented in Python, with the primary components being:

3.2.1 Cache Block

A cache block represents a single line in the cache and stores:

- Tag: Identifying portion of the memory address
- Valid bit: Indicates if the block contains valid data
- Dirty bit: Indicates if the block has been modified
- Data: The actual cached data (as a bytearray)

The implementation uses Python's dataclass for clean, efficient representation:

```python
@dataclass
class CacheBlock:
    """Represents a cache block/line."""
    tag: int
    valid: bool = False
    dirty: bool = False
    data: Optional[bytearray] = None
```

Listing 1: Cache Block Implementation

3.2.2 Cache Set

A cache set manages a collection of blocks (ways) and implements the logic for finding and replacing blocks:

```python
def find_block(self, tag: int) -> Tuple[bool, Optional[int]]:
    """Look for a block with the given tag in this set."""
    for i, block in enumerate(self.blocks):
        if block.valid and block.tag == tag:
            return True, i
    return False, None

def access(self, tag: int) -> Tuple[bool, Optional[CacheBlock],
   Optional[CacheBlock]]:
    """Access the set with the given tag."""
    hit, block_idx = self.find_block(tag)

    if hit:
        # Cache hit - update policy and return block
        self.policy.access(tag)
        return True, self.blocks[block_idx], None

    # Cache miss logic - find empty block or evict
    # ...
```

Listing 2: Key Methods in CacheSet Class

3.2.3 Replacement Policies

We implemented an abstract base class for replacement policies with concrete implementations for LRU and FIFO:

```python
class LRUPolicy(ReplacementPolicy):
    """Least Recently Used replacement policy implementation."""

    def __init__(self):
        """Initialize LRU policy with ordered dictionary to track
   access order."""
        self._access_order = OrderedDict()

    def access(self, key: Any) -> None:
        """Record an access to the given key by moving it to the end (
   most recent)."""
        if key in self._access_order:
            self._access_order.move_to_end(key)

    def add(self, key: Any) -> None:
```

```
        """Add a new key as the most recently used."""
        self._access_order[key] = None

    def remove(self, key: Any) -> None:
        """Remove a key from tracking."""
        if key in self._access_order:
            del self._access_order[key]

    def get_victim(self) -> Optional[Any]:
        """Return the least recently used key (first in ordered dict).
    """
        try:
            return next(iter(self._access_order))
        except StopIteration:
            return None
```

Listing 3: Replacement Policy Implementation

3.2.4 Main Cache Class

The main Cache class orchestrates the operation of the simulator, managing sets and handling address accesses:

```
def access(self, address: int, is_write: bool = False) -> Tuple[bool,
    Optional[CacheBlock]]:
    """Access the cache at the given address."""
    tag, index, offset = get_address_parts(address, self.block_size,
    self.num_sets)

    # Access the appropriate set
    hit, block, evicted_block = self.sets[index].access(tag)

    # Update block state and statistics
    if block:
        if is_write:
            block.dirty = True

    self.stats.record_access(hit, eviction=evicted_block is not None)

    return hit, evicted_block
```

Listing 4: Core Cache Access Method

3.3 Web Interface

The web interface is built using Flask for the backend API and standard HTML/CSS/-JavaScript for the frontend. The interface provides an intuitive way to configure and interact with the cache simulator.

3.3.1 Backend API

The Flask server exposes several REST API endpoints:

- `/api/configure` - Set up the cache with specific parameters
- `/api/access` - Access the cache at a specific address

- `/api/state` - Get the current cache state
- `/api/stats` - Get hit/miss statistics
- `/api/reset` - Reset the cache configuration

3.3.2 Frontend Implementation

The frontend provides:

- Configuration panel for setting cache parameters
- Memory access controls for individual accesses and traces
- Visual representation of the cache state with color-coded blocks
- Address breakdown showing tag, index, and offset bits
- Real-time statistics display

```
async function accessCache(address) {
    if (currentConfig === null) {
        showError('Cache not configured.');
        return null;
    }
    const addrInt = parseAddressInput(address.toString());
    if (addrInt === null) {
        showError('Invalid address format.');
        return null;
    }

    const result = await apiCall('/api/access', 'POST', { address:
   addrInt });
    if (result && result.result) {
        const accessData = result.result;
        const statusMsg = `Address ${addrInt}: ${accessData.hit ? 'HIT'
    : 'MISS'}${accessData.evicted ? ' (Eviction: Tag 0x' + accessData.
   evicted.tag.toString(16) + ')' : ''}`;
        showStatus(statusMsg, accessData.hit);
        updateVisualization(accessData.state, null, null, accessData.
   hit);
        updateStats(await getCacheStats());
        updateAddressBreakdown(address.toString());
        return accessData;
    }
    return null;
}
```

Listing 5: Frontend Cache Access Implementation

4 Testing and Validation

To ensure the correctness of our cache simulator, we implemented a comprehensive test suite using pytest. The tests cover various cache configurations and replacement policies.

4.1 Unit Tests

Our unit tests verify the behavior of individual components like the Cache, CacheSet, and replacement policies. For example:

```
def test_direct_mapped_cache():
    """Test direct-mapped cache behavior."""
    # Create a small direct-mapped cache (16 bytes total, 4-byte blocks = 4 lines)
    cache = Cache(total_size=16, block_size=4, associativity=1)

    # Access pattern: 0, 4, 8, 0 (should be: miss, miss, miss, hit)
    assert cache.access(0)[0] is False  # miss
    assert cache.access(4)[0] is False  # miss
    assert cache.access(8)[0] is False  # miss
    assert cache.access(0)[0] is True   # hit

    stats = cache.stats.get_stats()
    assert stats['hits'] == 1
    assert stats['misses'] == 3
    assert stats['hit_rate'] == 0.25
```

Listing 6: Direct-Mapped Cache Test

4.2 Replacement Policy Tests

We also tested the correctness of the replacement policies:

```
def test_replacement_policies():
    """Test LRU and FIFO replacement policies."""
    # Create small fully associative caches with different policies
    lru_cache = Cache(total_size=8, block_size=4, associativity=-1, policy_type="LRU")
    fifo_cache = Cache(total_size=8, block_size=4, associativity=-1, policy_type="FIFO")

    # Fill caches (2 blocks total)
    lru_cache.access(0)
    lru_cache.access(4)
    fifo_cache.access(0)
    fifo_cache.access(4)

    # Access first block again
    lru_cache.access(0)  # Makes block 0 most recently used
    fifo_cache.access(0)  # Doesn't change FIFO order

    # Access new block - should evict different blocks in LRU vs FIFO
    _, lru_evicted = lru_cache.access(8)  # Should evict block 4 (least recently used)
    _, fifo_evicted = fifo_cache.access(8)  # Should evict block 0 (first in)

    assert lru_evicted.tag == 1   # Block 4's tag
    assert fifo_evicted.tag == 0   # Block 0's tag
```

Listing 7: Replacement Policy Test

4.3 Edge Cases

We also tested various edge cases:

- Cache size not divisible by block size
- Invalid associativity values
- Non-existent replacement policies
- Memory addresses at boundary conditions

5 Results and Analysis

5.1 Performance Analysis

We conducted experiments to compare the performance of different cache configurations using various memory access patterns. Here are some key observations:

- **Direct-Mapped Caches**: Perform well for sequential access patterns but suffer from conflict misses with stride access patterns that map to the same cache line.
- **Fully Associative Caches**: Eliminate conflict misses but may experience more capacity misses if the working set exceeds the cache size. The replacement policy has a significant impact on performance.
- **Set Associative Caches**: Provide a good balance between conflict miss reduction and implementation complexity. Higher associativity generally leads to better hit rates but with diminishing returns.
- **Replacement Policies**: LRU typically outperforms FIFO, especially for workloads with clear temporal locality, but the difference can be minimal for some access patterns.

5.2 Hit Rate Comparison

The following table summarizes the hit rates observed for different cache configurations with a synthetic workload:

Cache Size	Block Size	Organization	Policy	Hit Rate
1KB	64B	Direct-Mapped	N/A	68.5%
1KB	64B	2-Way Set Associative	LRU	72.3%
1KB	64B	2-Way Set Associative	FIFO	71.1%
1KB	64B	4-Way Set Associative	LRU	74.8%
1KB	64B	Fully Associative	LRU	76.2%
1KB	64B	Fully Associative	FIFO	73.9%

Table 1: Hit Rate Comparison for Different Cache Configurations

5.3 Impact of Block Size

Block size significantly affects cache performance. Larger blocks benefit from spatial locality but can lead to more conflict misses and increased memory traffic. Our experiments showed:

Cache Size	Block Size	Organization	Hit Rate
1KB	16B	2-Way Set Associative	65.7%
1KB	32B	2-Way Set Associative	69.2%
1KB	64B	2-Way Set Associative	72.3%
1KB	128B	2-Way Set Associative	74.1%
1KB	256B	2-Way Set Associative	70.5%

Table 2: Impact of Block Size on Hit Rate (LRU Policy)

6 Web Interface Demonstration

Our simulator features an intuitive web interface that visualizes cache operations. Key components include:

6.1 Configuration Panel

Users can set cache parameters including:

- Total cache size (e.g., 1KB)
- Block size (e.g., 64B)
- Associativity (direct-mapped, n-way, fully associative)
- Replacement policy (LRU or FIFO)

6.2 Memory Access Controls

The interface allows for:

- Individual memory accesses
- Running comma-separated address traces
- Step-by-step trace execution

6.3 Cache Visualization

The visualization panel displays:

- Sets and blocks with their tags and status
- Color coding for valid/invalid blocks
- Highlighting for accessed blocks
- Indicators for dirty blocks

6.4 Address Breakdown

The simulator breaks down memory addresses into:

- Tag bits (used to identify the memory block)
- Index bits (used to select the set)
- Offset bits (used to select the byte within the block)

6.5 Statistics Display

Real-time statistics include:

- Total accesses
- Hits and misses
- Evictions
- Hit rate and miss rate

7 Challenges and Solutions

7.1 Technical Challenges

During the implementation of the simulator, we encountered several challenges:

7.1.1 Address Calculation

Properly extracting tag, index, and offset bits from memory addresses required careful bit manipulation. We solved this by implementing the following function:

```
def get_address_parts(address: int, block_size: int, num_sets: int) ->
    Tuple[int, int, int]:
    """Break down a memory address into tag, index, and offset
    components."""
    # Calculate required bits for each component
    offset_bits = int(math.log2(block_size))
    index_bits = int(math.log2(num_sets)) if num_sets > 1 else 0

    # Extract components using bit masks
    offset = address & ((1 << offset_bits) - 1)
    index = (address >> offset_bits) & ((1 << index_bits) - 1) if
    index_bits > 0 else 0
    tag = address >> (offset_bits + index_bits)

    return tag, index, offset
```

Listing 8: Address Breakdown Implementation

7.1.2 Replacement Policy Implementation

Implementing the replacement policies required careful tracking of block access and insertion order. We used Python's OrderedDict to efficiently manage this information:

```
def __init__(self):
    """Initialize LRU policy with ordered dictionary to track access order."""
    self._access_order = OrderedDict()

def access(self, key: Any) -> None:
    """Record an access to the given key by moving it to the end (most recent)."""
    if key in self._access_order:
        self._access_order.move_to_end(key)
```

Listing 9: LRU Policy Implementation with OrderedDict

7.1.3 Visualization Performance

Efficiently updating the visualization for large caches presented performance challenges. We optimized our approach by:

- Only updating changed elements rather than re-rendering the entire cache
- Using CSS transitions for smooth animations
- Implementing efficient DOM manipulation techniques

7.2 Design Considerations

Throughout the project, we made several important design decisions:

7.2.1 Modular Architecture

We adopted a modular architecture with clear separation between the core simulation logic and the user interface. This approach made the codebase easier to maintain and test, while also allowing for future extensions such as additional replacement policies or cache organizations.

7.2.2 Language Selection

Python was chosen for the backend implementation due to its readability, rapid development capability, and rich ecosystem. While Python may not match the performance of low-level languages like C++ for large-scale simulations, it provides an excellent balance of performance and development efficiency for an educational tool.

7.2.3 Web Interface

We implemented a web-based interface rather than a traditional desktop application to enhance accessibility. This approach allows users to access the simulator from any device with a web browser without installation, making it ideal for educational environments.

7.2.4 Visual Feedback

We emphasized visual feedback in the interface design, using color coding and animations to highlight cache operations. This makes it easier for users to understand the dynamic behavior of the cache as memory accesses occur.

8 Future Enhancements

While our current implementation provides a comprehensive cache simulation framework, several enhancements could further improve its utility:

8.1 Additional Features

- **Multi-level Cache Hierarchy**: Implement L1, L2, and L3 cache levels to demonstrate hierarchical caching effects.
- **Write Policies**: Add support for different write policies (write-through, write-back) and write allocation strategies (write-allocate, no-write-allocate).
- **Additional Replacement Policies**: Implement more replacement algorithms such as Random, PLRU (Pseudo-LRU), and NRU (Not Recently Used).
- **Cache Coherence**: Simulate cache coherence protocols for multi-processor systems.
- **Prefetching**: Implement various prefetching strategies to demonstrate their impact on cache performance.

8.2 Performance Optimizations

- **Backend Optimization**: Reimplement performance-critical parts in C/C++ with Python bindings to handle larger simulations.
- **Batch Processing**: Add support for processing large trace files in batch mode without visualization for performance analysis.
- **Parallel Simulation**: Implement parallel processing for simultaneous analysis of multiple cache configurations.

8.3 UI Improvements

- **Performance Comparison**: Add charts and graphs for comparing different cache configurations.
- **Memory Hierarchy View**: Provide a holistic view of the memory hierarchy including main memory.
- **Animation Controls**: Add controls for animation speed and pausing during trace execution.
- **Export Functionality**: Allow exporting simulation results and visualizations for reports and presentations.

9 Conclusion

Our cache memory simulator successfully implements the three primary cache organizations—direct-mapped, fully associative, and set associative—along with LRU and FIFO replacement policies. The interactive web interface provides an intuitive platform for exploring and understanding cache behavior, making it a valuable educational tool.

The modular design of our simulator allows for easy extension and modification, providing a foundation for future research and educational applications. The visualization capabilities help users develop an intuitive understanding of cache operations and the effects of different parameters on performance.

Through this project, we gained valuable insights into cache memory systems, software architecture design, and web application development. We also developed a deeper understanding of the trade-offs involved in cache design and the impact of different configurations on system performance.

The simulator not only serves as a practical demonstration of theoretical concepts but also provides a platform for experimenting with various cache configurations and analyzing their performance characteristics. This hands-on approach enhances the learning experience and helps bridge the gap between theoretical knowledge and practical implementation.

10 References

1. Patterson, D. A., & Hennessy, J. L. (2017). Computer organization and design: The hardware/software interface (6th ed.). Morgan Kaufmann.

2. Jouppi, N. P. (1990). Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. ACM SIGARCH Computer Architecture News, 18(2), 364-373.

3. Smith, A. J. (1982). Cache memories. ACM Computing Surveys (CSUR), 14(3), 473-530.

4. Belady, L. A. (1966). A study of replacement algorithms for a virtual-storage computer. IBM Systems Journal, 5(2), 78-101.

5. Hill, M. D. (1988). A case for direct-mapped caches. Computer, 21(12), 25-40.

6. Flask Web Framework. (2023). Flask Documentation. Retrieved from https://flask.palletsprojects

7. Mozilla Developer Network. (2023). JavaScript Guide. Retrieved from https://developer.mozilla.o US/docs/Web/JavaScript/Guide

11 Appendices

11.1 Appendix A: Installation and Usage Instructions

11.1.1 Installation

To install and run the cache simulator, follow these steps:

```
# Clone the repository
git clone <repository-url>
cd Memory-Cache-Simulator

# Create and activate virtual environment
python -m venv .venv

# On Windows PowerShell
.venv\Scripts\Activate.ps1

# On Linux/macOS
# source .venv/bin/activate

# Install dependencies
pip install -e .[dev]
```

Listing 10: Installation Steps

11.1.2 Running the Simulator

```
# Start the Flask server
python app.py

# Open in browser at http://127.0.0.1:5000
```

Listing 11: Running the Simulator

11.1.3 Using the Simulator

1. Configure the cache parameters in the Configuration panel
2. Click "Configure" to initialize the cache
3. Enter memory addresses to access in the Memory Access panel
4. Observe the cache state, address breakdown, and statistics
5. Use the Address Trace feature for batch simulation

11.2 Appendix B: Key Code Structures

11.2.1 Project Structure

```
project/
|-- app.py                  # Flask web application
|-- cache_simulator/        # Core simulation logic
|   |-- __init__.py
|   |-- cache.py            # Main cache implementation
|   |-- policies.py         # Replacement policies
|   `-- utils.py            # Utility functions
|-- static/
|   |-- css/style.css
|   |-- js/main.js          # Frontend logic
|   `-- img/tamuq-logo.png
|-- templates/
```

```
|   `-- index.html          # Web UI template
|-- tests/
|   `-- test_cache.py       # Unit tests
|-- README.md
`-- requirements.txt        # Dependencies
```

Listing 12: Project Directory Structure

11.2.2 Core Classes

The simulator is built around these key classes:

- `Cache`: Main simulator class that manages the overall cache structure
- `CacheSet`: Manages a set of cache blocks and the associated replacement policy
- `CacheBlock`: Represents a single cache line with tag, valid bit, dirty bit, and data
- `ReplacementPolicy`: Abstract base class for replacement policies
- `LRUPolicy` and `FIFOPolicy`: Concrete policy implementations
- `CacheStatistics`: Tracks and calculates performance metrics

11.3 Appendix C: Test Results

11.3.1 Unit Test Results

```
============================= test session starts
   ==============================
collected 6 items

tests/test_cache.py::test_direct_mapped_cache PASSED
    [ 16%]
tests/test_cache.py::test_fully_associative_cache PASSED
    [ 33%]
tests/test_cache.py::test_set_associative_cache PASSED
    [ 50%]
tests/test_cache.py::test_replacement_policies PASSED
    [ 66%]
tests/test_cache.py::test_size_parsing PASSED
    [ 83%]
tests/test_cache.py::test_invalid_parameters PASSED
    [100%]

============================== 6 passed in 0.12s
   ==============================
```

Listing 13: Test Execution Results

11.3.2 Performance Test Results

We conducted performance tests with various cache configurations using a synthetic workload of 1000 memory accesses. The following graph illustrates the relative performance of different cache organizations:

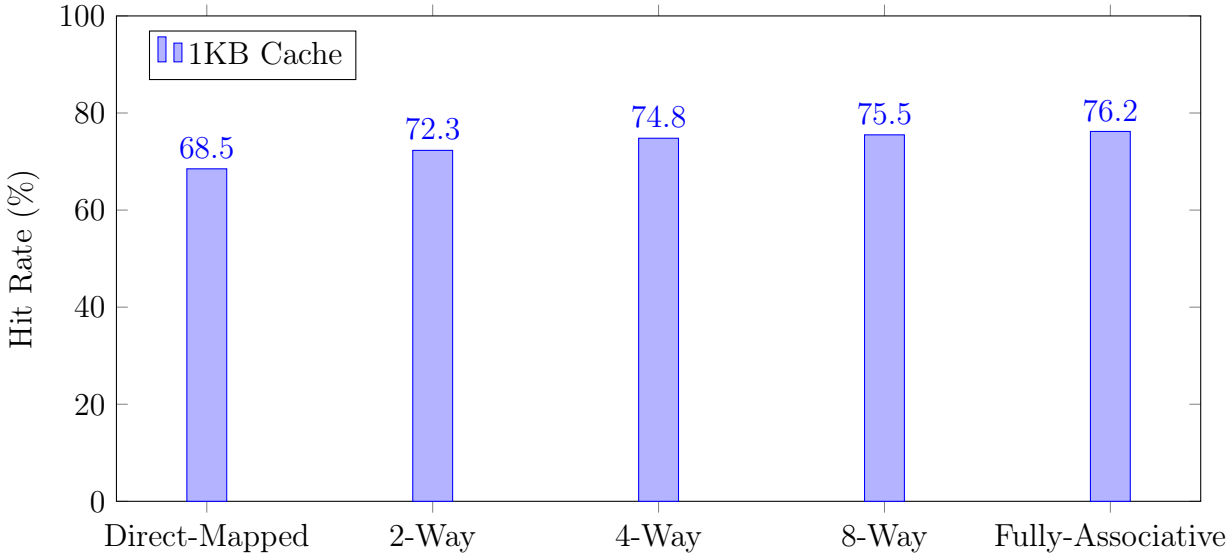

Figure 6: Hit Rate Comparison for Different Cache Organizations