Cache Simulator: Detailed Technical Documentation

Table of Contents

Introduction

The cache simulator is a Python-based tool designed to model and analyze CPU cache behavior under different configurations and memory access patterns. This detailed documentation provides a comprehensive exploration of the code, explaining the implementation details and the underlying concepts of cache memory.

This simulator's core functionality allows users to:

- Configure cache parameters (size, block size, associativity, replacement policy)
- Generate different memory access patterns
- Simulate cache behavior for these patterns
- Analyze and visualize the results
- Compare different cache configurations

Understanding Cache Memory

Before diving into the code, it's essential to understand how cache memory works at a fundamental level.

Memory Address Breakdown

In a cache system, a memory address is typically divided into three parts:

```
|------- Tag ------|-- Set Index --|-- Block Offset --|
```

- **Block Offset**: Identifies a specific byte within a cache block
- **Set Index**: Determines which cache set the address maps to
- **Tag**: Used to identify a specific block within a set

The bit widths of these fields depend on the cache parameters:

- Block offset: $\log_2$(block size) bits
- Set index: $\log_2$(number of sets) bits
- Tag: remaining address bits

For example, with a 32-bit address, 64-byte blocks (6 bits for offset), and 128 sets (7 bits for index), the tag would be 19 bits.

Cache Mapping Techniques

The simulator supports three cache mapping techniques:

1. **Direct-Mapped Cache (Associativity = 1)**:
    - Each memory block maps to exactly one cache location
    - Simple to implement but prone to conflict misses
    - Address mapping: `set_index = (address / block_size) % num_sets`
2. **Set-Associative Cache (Associativity = N)**:
    - Each memory block can go into any of N positions in a set
    - Balances flexibility and implementation complexity
    - Requires a replacement policy to decide which block to evict
3. **Fully Associative Cache (Associativity = -1 in the code)**:
    - A memory block can go anywhere in the cache
    - Maximum flexibility but requires searching the entire cache
    - Implemented as a single set containing all cache blocks

Replacement Policies

The simulator implements four replacement policies:

1. **LRU (Least Recently Used)**:

    - Evicts the block that hasn't been accessed for the longest time
    - Tracks access time for each block
    - Generally provides good performance for most workloads

2. **FIFO (First-In-First-Out)**:
    - Evicts the oldest block in the set, regardless of usage
    - Simpler than LRU but typically less effective
    - Tracks insertion time for each block

3. **LFU (Least Frequently Used)**:
    - Evicts the block that has been accessed the least number of times
    - Tracks access frequency for each block
    - Can be effective for certain workloads but may hold onto blocks too long

4. **Random**:
    - Randomly selects a block for replacement
    - Simple to implement and requires no metadata
    - Serves as a baseline for comparison

Code Implementation: Line-by-Line Analysis

Let's analyze the implementation details of the key components in the code.

CacheSimulator Class

The core class is `CacheSimulator`, which models a parameterized cache:

```
class CacheSimulator:
    def __init__(self, cache_size, block_size, associativity,
replacement_policy):
        """
        Initialize the cache simulator with given parameters
        """
        self.cache_size = cache_size
        self.block_size = block_size
        self.associativity = associativity
        self.replacement_policy = replacement_policy.upper()
```

These initial lines store the basic parameters of the cache:

- `cache_size`: Total size of the cache in bytes
- `block_size`: Size of each cache block in bytes
- `associativity`: Number of ways in a set-associative cache (1 for direct-mapped, -1 for fully associative)
- `replacement_policy`: Algorithm for block replacement (converted to uppercase for case-insensitivity)

```
# Calculate cache organization parameters
self.num_blocks = cache_size // block_size

if associativity == -1:  # Fully associative
    self.num_sets = 1
    self.set_size = self.num_blocks
else:
    self.set_size = associativity
    self.num_sets = self.num_blocks // self.set_size
```

This code calculates the cache structure based on the provided parameters:

- `num_blocks`: Total number of blocks in the cache (cache_size / block_size)
- For fully associative caches (associativity = -1):
  - Creates a single set containing all blocks
- For direct-mapped and set-associative caches:
  - `set_size`: Number of blocks per set (equals associativity)
  - `num_sets`: Number of sets in the cache (num_blocks / set_size)

```
# Initialize cache structure
self.cache = [[] for _ in range(self.num_sets)]
```

The cache is implemented as a list of sets, where each set is initially an empty list. As blocks are added to the cache, each set will contain tuples of `(tag, metadata)`, where the metadata depends on the replacement policy.

```
# Counters
self.hits = 0
self.misses = 0
self.accesses = 0
self.clock = 0  # Used for LRU and FIFO policies

# Statistics tracking
self.hit_addresses = []
self.miss_addresses = []
self.access_trace = []
```

These variables track cache performance metrics and maintain lists of hit and miss addresses for later visualization.

Address Processing Methods

The simulator provides two key methods to extract information from memory addresses:

```
def _get_set_index(self, address):
    """Calculate the set index for an address"""
    # Extract block number (remove offset bits)
    block_number = address // self.block_size

    if self.associativity == -1:  # Fully associative
        return 0
    else:
        # Extract set index bits
        return block_number % self.num_sets
```

This method calculates which cache set a memory address maps to:

1. First, it divides the address by the block size to get the block number (essentially removing the offset bits)
2. For fully associative caches, it always returns 0 (there's only one set)
3. For other cache types, it calculates the set index using modulo operation

```
def _get_tag(self, address):
    """Calculate the tag for an address"""
    # Extract block number (remove offset bits)
    block_number = address // self.block_size

    if self.associativity == -1:  # Fully associative
        return block_number
    else:
        # Extract tag bits
        return block_number // self.num_sets
```

This method extracts the tag portion of the address:

1. First, it divides the address by the block size to get the block number
2. For fully associative caches, the entire block number is the tag
3. For other cache types, it divides the block number by the number of sets to get the tag

The Access Method

The `access` method is the heart of the cache simulator, modeling how the cache responds to memory accesses:

```
def access(self, address):
    """
    Simulate a memory access to the given address

    Args:
        address (int): Memory address to access
```

```
    Returns:
        bool: True if hit, False if miss
    """
    self.accesses += 1
    self.clock += 1
    self.access_trace.append(address)
```

These initial lines:

1. Increment the total access counter
2. Increment the clock (used for tracking access/insertion time)
3. Add the address to the access trace for later visualization

```
tag = self._get_tag(address)
set_index = self._get_set_index(address)
cache_set = self.cache[set_index]
```

This code processes the address:

1. Extract the tag from the address
2. Calculate the set index for the address
3. Get a reference to the specific cache set

```
# Check if we have a hit
for i, (block_tag, metadata) in enumerate(cache_set):
    if block_tag == tag:
        # Hit: Update metadata based on replacement policy
        if self.replacement_policy == 'LRU':
            cache_set[i] = (tag, self.clock)
        elif self.replacement_policy == 'LFU':
            cache_set[i] = (tag, metadata + 1)
        # FIFO: No update needed on hit

        self.hits += 1
        self.hit_addresses.append(address)
        return True
```

This section checks if the requested block is already in the cache:

1. Iterate through all blocks in the set
2. If the tag matches, it's a cache hit
3. Update the metadata based on the replacement policy:
    - For LRU: Update the access time to the current clock
    - For LFU: Increment the access frequency
    - For FIFO: No update needed (insertion time remains unchanged)
4. Record the hit and return True

```
# Miss: Need to insert the block
self.misses += 1
self.miss_addresses.append(address)

# Initialize metadata based on replacement policy
if self.replacement_policy == 'LRU':
    metadata = self.clock
elif self.replacement_policy == 'FIFO':
    metadata = self.clock
elif self.replacement_policy == 'LFU':
    metadata = 1
else:  # RANDOM
    metadata = None
```

If the block is not in the cache, it's a miss:

1. Increment the miss counter and record the address
2. Initialize metadata for the new block based on the replacement policy:
    - For LRU/FIFO: Current clock value (time of insertion)
    - For LFU: Initial access count of 1
    - For RANDOM: No metadata needed

```
# If set isn't full, simply add the block
if len(cache_set) < self.set_size:
    cache_set.append((tag, metadata))
else:
    # Need to replace a block based on the policy
    if self.replacement_policy == 'LRU':
        # Replace the least recently used
        lru_index = min(range(len(cache_set)), key=lambda i: cache_set[i]
[1])
        cache_set[lru_index] = (tag, metadata)
    elif self.replacement_policy == 'FIFO':
        # Replace the first inserted
        fifo_index = min(range(len(cache_set)), key=lambda i: cache_set[i]
[1])
        cache_set[fifo_index] = (tag, metadata)
    elif self.replacement_policy == 'LFU':
        # Replace the least frequently used
        lfu_index = min(range(len(cache_set)), key=lambda i: cache_set[i]
[1])
        cache_set[lfu_index] = (tag, metadata)
    else:  # RANDOM
        # Replace a random entry
        random_index = random.randint(0, len(cache_set) - 1)
        cache_set[random_index] = (tag, metadata)
```

This code handles block insertion:

1. If the set has space, simply append the new block
2. If the set is full, replace a block according to the replacement policy:
    - For LRU: Replace the block with the lowest clock value (oldest access time)
    - For FIFO: Replace the block with the lowest clock value (earliest insertion time)
    - For LFU: Replace the block with the lowest access count
    - For RANDOM: Replace a randomly selected block

Metrics and Visualization Methods

The simulator provides methods to calculate performance metrics and visualize results:

```
def get_hit_rate(self):
    """Calculate the cache hit rate"""
    if self.accesses == 0:
        return 0
    return self.hits / self.accesses

def get_miss_rate(self):
    """Calculate the cache miss rate"""
    if self.accesses == 0:
        return 0
    return self.misses / self.accesses
```

These methods calculate the hit and miss rates, handling the edge case where no accesses have occurred.

```
def reset_stats(self):
    """Reset hit/miss statistics"""
    self.hits = 0
    self.misses = 0
    self.accesses = 0
    self.clock = 0
    self.hit_addresses = []
    self.miss_addresses = []
    self.access_trace = []

    # Reset cache
    self.cache = [[] for _ in range(self.num_sets)]
```

This method resets all statistics and clears the cache, useful for running multiple tests with the same cache configuration.

```
def print_stats(self):
    """Print cache statistics"""
    print("\nCache Statistics:")
    print(f"  Total Accesses: {self.accesses}")
    print(f"  Hits: {self.hits}")
    print(f"  Misses: {self.misses}")
```

```
        print(f"  Hit Rate: {self.get_hit_rate():.4f}")
        print(f"  Miss Rate: {self.get_miss_rate():.4f}")
```

This method prints a summary of cache performance metrics.

Memory Trace Generation: In-Depth

The `generate_memory_trace` function creates different memory access patterns for testing cache behavior:

Sequential Pattern Implementation

```
if pattern_type == 'sequential':
    # Sequential access pattern
    for i in range(num_accesses):
        trace.append(i % address_range)
```

This creates a sequential access pattern:

- Accesses addresses 0, 1, 2, ..., up to address_range-1
- When it reaches the end of the range, it wraps around to 0
- Creates perfect spatial locality and predictable access pattern

Random Pattern Implementation

```
elif pattern_type == 'random':
    # Random access pattern
    for _ in range(num_accesses):
        trace.append(random.randint(0, address_range - 1))
```

This creates a uniform random access pattern:

- Each address in the range has an equal probability of being accessed
- No inherent spatial or temporal locality
- Provides a worst-case scenario for most cache designs

Loop Pattern Implementation

```
elif pattern_type == 'loop':
    # Loop access pattern (repeatedly access a fixed range)
    loop_size = min(100, address_range)
    loop_start = random.randint(0, address_range - loop_size)
    loop_addresses = list(range(loop_start, loop_start + loop_size))

    for i in range(num_accesses):
```

```
            idx = i % loop_size
            trace.append(loop_addresses[idx])
```

This creates a loop access pattern:

1. Select a contiguous region of memory of size `loop_size` (up to 100 addresses)
2. Repeatedly iterate through this region in order
3. Creates strong temporal locality (same addresses accessed repeatedly)
4. Mimics common programming constructs like loops over arrays

Locality Pattern Implementation

```
elif pattern_type == 'locality':
    # Temporal and spatial locality
    # 80% of accesses in 20% of address space
    hot_region_size = int(0.2 * address_range)
    hot_region_start = random.randint(0, address_range - hot_region_size)

    for _ in range(num_accesses):
        if random.random() < 0.8:  # 80% probability of hot region
            addr = random.randint(hot_region_start, hot_region_start +
hot_region_size - 1)
        else:
            # Access outside hot region
            if random.random() < 0.5 and hot_region_start > 0:
                # Before hot region
                addr = random.randint(0, hot_region_start - 1)
            else:
                # After hot region
                addr = random.randint(hot_region_start + hot_region_size,
address_range - 1)
        trace.append(addr)
```

This creates a locality-based access pattern following the 80/20 principle:

1. Define a "hot" region consisting of 20% of the address space
2. 80% of all accesses go to this hot region
3. The remaining 20% of accesses are distributed across the rest of the address space
4. Creates a realistic mix of temporal and spatial locality

Comparative Analysis Functions

The simulator includes functions to compare different cache configurations:

Policy Comparison Implementation

```
def compare_policies(trace, cache_size, block_size, associativity):
    """
```

```
    Compare different replacement policies for the same trace and cache
configuration
    """
    policies = ['LRU', 'FIFO', 'LFU', 'RANDOM']
    results = {}

    for policy in policies:
        simulator = CacheSimulator(cache_size, block_size, associativity,
policy)
        for addr in trace:
            simulator.access(addr)

        results[policy] = {
            'hit_rate': simulator.get_hit_rate(),
            'miss_rate': simulator.get_miss_rate(),
            'hits': simulator.hits,
            'misses': simulator.misses
        }

        print(f"\n{policy} Policy Results:")
        simulator.print_stats()

    return results
```

This function:

1. Tests all four replacement policies with the same cache parameters and memory trace
2. Creates a new simulator for each policy
3. Runs the entire trace through each simulator
4. Collects and returns performance metrics for each policy

Associativity Comparison Implementation

```
def compare_associativities(trace, cache_size, block_size, policy):
    """
    Compare different associativity levels for the same trace, cache size,
and policy
    """
    # Test direct-mapped, 2-way, 4-way, 8-way, and fully associative
    associativities = [1, 2, 4, 8, -1]  # -1 for fully associative
    results = {}

    for assoc in associativities:
        if assoc == -1:
            assoc_name = "Fully Associative"
        elif assoc == 1:
            assoc_name = "Direct-Mapped"
        else:
            assoc_name = f"{assoc}-Way Set Associative"

        print(f"\nTesting {assoc_name}:")
```

```
        simulator = CacheSimulator(cache_size, block_size, assoc, policy)
        for addr in trace:
            simulator.access(addr)

        results[assoc_name] = {
            'hit_rate': simulator.get_hit_rate(),
            'miss_rate': simulator.get_miss_rate(),
            'hits': simulator.hits,
            'misses': simulator.misses
        }

        simulator.print_stats()

    return results
```

This function:

1. Tests five different associativity levels: direct-mapped, 2-way, 4-way, 8-way, and fully associative
2. Uses the same replacement policy, cache size, and block size for all tests
3. Creates descriptive names for each associativity level
4. Collects and returns performance metrics for each associativity level

Block Size Comparison Implementation

```
def compare_block_sizes(trace, cache_size, associativity, policy):
    """
    Compare different block sizes for the same trace, cache size, and
associativity
    """
    # Test different block sizes (in bytes)
    block_sizes = [16, 32, 64, 128, 256]
    results = {}

    for block_size in block_sizes:
        print(f"\nTesting Block Size {block_size} bytes:")
        simulator = CacheSimulator(cache_size, block_size, associativity,
policy)
        for addr in trace:
            simulator.access(addr)

        results[block_size] = {
            'hit_rate': simulator.get_hit_rate(),
            'miss_rate': simulator.get_miss_rate(),
            'hits': simulator.hits,
            'misses': simulator.misses
        }

        simulator.print_stats()

    return results
```

This function:

1. Tests five different block sizes: 16, 32, 64, 128, and 256 bytes
2. Uses the same replacement policy, cache size, and associativity for all tests
3. Collects and returns performance metrics for each block size

Visualization Methods Explained

The simulator includes visualization tools to help interpret the results:

Access Pattern Plotting

```
def plot_access_pattern(self, output_file=None):
    """
    Plot the access pattern with hits and misses
    """
    plt.figure(figsize=(12, 6))

    # Plot all accesses
    x = list(range(len(self.access_trace)))
    plt.scatter(x, self.access_trace, c='gray', s=10, alpha=0.3, label='All
Accesses')

    # Plot hits
    if self.hit_addresses:
        hit_indices = []
        hit_addresses = []
        for i, addr in enumerate(self.access_trace):
            if addr in self.hit_addresses and
self.hit_addresses.count(addr) > 0:
                hit_indices.append(i)
                hit_addresses.append(addr)
                self.hit_addresses.remove(addr)  # Remove to handle
duplicate addresses
        plt.scatter(hit_indices, hit_addresses, c='green', s=20,
label='Cache Hits')

    # Plot misses
    if self.miss_addresses:
        miss_indices = []
        miss_addresses = []
        for i, addr in enumerate(self.access_trace):
            if addr in self.miss_addresses and
self.miss_addresses.count(addr) > 0:
                miss_indices.append(i)
                miss_addresses.append(addr)
                self.miss_addresses.remove(addr)  # Remove to handle
duplicate addresses
        plt.scatter(miss_indices, miss_addresses, c='red', s=20,
label='Cache Misses')

    plt.title('Memory Access Pattern with Cache Hits and Misses')
```

```
    plt.xlabel('Access Number')
    plt.ylabel('Memory Address')
    plt.legend()
    plt.grid(True)

    if output_file:
        plt.savefig(output_file)
    else:
        plt.show()
```

This method visualizes the memory access pattern and cache behavior:

1. Creates a scatter plot of all memory accesses over time
2. Overlays cache hits in green
3. Overlays cache misses in red
4. The x-axis represents the sequence of accesses
5. The y-axis represents the memory addresses
6. The plot can be saved to a file or displayed directly

Note the special handling for duplicate addresses in the hit and miss lists, which ensures each address is properly plotted.

Comparative Results Plotting

```
def plot_comparison(results, title, xlabel, ylabel='Hit Rate'):
    """
    Plot comparison of results
    """
    plt.figure(figsize=(10, 6))

    categories = list(results.keys())
    hit_rates = [results[cat]['hit_rate'] for cat in categories]
    miss_rates = [results[cat]['miss_rate'] for cat in categories]

    x = np.arange(len(categories))
    width = 0.35

    plt.bar(x - width/2, hit_rates, width, label='Hit Rate')
    plt.bar(x + width/2, miss_rates, width, label='Miss Rate')

    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(title)
    plt.xticks(x, categories, rotation=45 if len(categories[0]) > 5 else 0)
    plt.legend()
    plt.grid(True, axis='y')
    plt.tight_layout()

    plt.show()
```

This function creates bar charts to compare different cache configurations:

1. Takes a dictionary of results from the comparison functions
2. Creates a grouped bar chart with hit rates and miss rates
3. Customizes labels, title, and formatting
4. Automatically rotates long labels to prevent overlap

Comprehensive Analysis

The `run_comprehensive_analysis` function performs a detailed investigation of cache configurations:

```
def run_comprehensive_analysis():
    """Run a comprehensive analysis with various cache configurations and
access patterns"""
    # Define parameters
    cache_sizes = [1024, 4096, 16384]  # 1KB, 4KB, 16KB
    block_sizes = [32, 64, 128]
    associativities = [1, 2, 4, -1]  # 1 = direct-mapped, -1 = fully
associative
    policies = ['LRU', 'FIFO', 'LFU', 'RANDOM']
    access_patterns = ['sequential', 'random', 'loop', 'locality']

    # Define number of addresses to access
    num_accesses = 10000
    address_range = 32768  # 32KB address space
```

This setup defines a comprehensive set of parameters to test, creating a multi-dimensional test space.

```
# Store results
all_results = {}

# Generate traces for each access pattern
traces = {}
for pattern in access_patterns:
    traces[pattern] = generate_memory_trace(pattern, num_accesses,
address_range)
```

This code generates memory traces for each access pattern once, to ensure consistent testing across configurations.

```
# Run simulations
print("Running Comprehensive Cache Analysis...")
for cache_size in cache_sizes:
    all_results[cache_size] = {}
    for block_size in block_sizes:
        all_results[cache_size][block_size] = {}
        for assoc in associativities:
            all_results[cache_size][block_size][assoc] = {}
```

```
            for policy in policies:
                all_results[cache_size][block_size][assoc][policy] = {}
                for pattern in access_patterns:
                    # Create simulator
                    simulator = CacheSimulator(cache_size, block_size,
assoc, policy)

                    # Run simulation
                    for addr in traces[pattern]:
                        simulator.access(addr)

                    # Store results
                    all_results[cache_size][block_size][assoc][policy]
[pattern] = {
                        'hit_rate': simulator.get_hit_rate(),
                        'miss_rate': simulator.get_miss_rate(),
                        'hits': simulator.hits,
                        'misses': simulator.misses
                    }
```

This nested loop structure:

1. Iterates through all combinations of cache parameters (720 total configurations)
2. Creates a simulator for each configuration
3. Runs each of the four memory traces through each configuration
4. Stores the results in a multi-level dictionary

```
# Print summary of best configurations for each access pattern
print("\nBest Cache Configurations for Each Access Pattern:")
for pattern in access_patterns:
    best_hit_rate = 0
    best_config = None

    for cache_size in cache_sizes:
        for block_size in block_sizes:
            for assoc in associativities:
                for policy in policies:
                    result = all_results[cache_size][block_size][assoc]
[policy][pattern]
                    if result['hit_rate'] > best_hit_rate:
                        best_hit_rate = result['hit_rate']
                        best_config = (cache_size, block_size, assoc,
policy)

    # Print best configuration
    cache_size, block_size, assoc, policy = best_config
    if assoc == -1:
        assoc_str = "Fully Associative"
    elif assoc == 1:
        assoc_str = "Direct-Mapped"
    else:
```

```
            assoc_str = f"{assoc}-Way Set Associative"

        print(f"\nBest Configuration for {pattern.capitalize()} Access
Pattern:")
        print(f"  Cache Size: {cache_size} bytes")
        print(f"  Block Size: {block_size} bytes")
        print(f"  Type: {assoc_str}")
        print(f"  Replacement Policy: {policy}")
        print(f"  Hit Rate: {best_hit_rate:.4f}")
```

This code identifies and prints the best configuration for each access pattern:

1. For each pattern, it finds the configuration with the highest hit rate
2. It then formats and prints the details of this optimal configuration

Command-Line Interface

The `main` function provides a flexible command-line interface:

```
def main():
    parser = argparse.ArgumentParser(description='Cache Simulator')
    parser.add_argument('--cache-size', type=int, default=8192, help='Cache
size in bytes')
    parser.add_argument('--block-size', type=int, default=64, help='Block
size in bytes')
    parser.add_argument('--associativity', type=int, default=2,
                        help='Associativity (1 for direct-mapped, -1 for
fully associative)')
    parser.add_argument('--policy', type=str, default='LRU', choices=
['LRU', 'FIFO', 'LFU', 'RANDOM'],
                        help='Replacement policy')
    parser.add_argument('--num-accesses', type=int, default=10000,
help='Number of memory accesses to simulate')
    parser.add_argument('--address-range', type=int, default=32768,
help='Range of addresses to access')
    parser.add_argument('--pattern', type=str, default='random',
                        choices=['sequential', 'random', 'loop',
'locality'],
                        help='Memory access pattern')
    parser.add_argument('--mode', type=str, default='single',
                        choices=['single', 'compare-policies', 'compare-
associativities',
                                 'compare-block-sizes', 'comprehensive'],
                        help='Simulation mode')

    args = parser.parse_args()
```

This code sets up the command-line interface using Python's `argparse` module, defining parameters with sensible defaults.

```
if args.mode == 'comprehensive':
    run_comprehensive_analysis()
    return

# Generate memory access trace
trace = generate_memory_trace(args.pattern, args.num_accesses,
args.address_range)

if args.mode == 'single':
    # Run a single simulation
    simulator = CacheSimulator(
        args.cache_size,
        args.
```