Project Progress Report: RL-Integrated Multicore Cache System

Date: January 9, 2026
Author: Harsh Raj
Phase: Foundation & Prototyping (Phase 1)

Abstract

This project explores the use of reinforcement learning for cache replacement in multicore processors. To validate the approach before full-system integration, two standalone C++ simulators were developed. The first models a directory-based MESI coherence protocol to verify correct state transitions and extract sharer count information. The second implements a lightweight tabular Q-learning agent that learns cache insertion and bypass decisions using the program counter as the state. Experiments using synthetic mixed workloads show that the agent successfully differentiates between looping and streaming memory access patterns, leading to improved cache utilization. While the learning setup is intentionally simplified, the results demonstrate that basic runtime features and reinforcement learning can capture useful cache behavior. These prototypes reduce integration risk and provide a foundation for future integration into the ChampSim simulator.

1. Executive Summary

This report documents the completion of the foundational prototyping phase of the project. Two standalone C++ simulation modules were developed to validate the core ideas before full ChampSim integration:

1. **Directory-Based Coherence Simulator:** Validates MESI state transitions and sharer tracking logic.
2. **RL-Based Replacement Agent:** Demonstrates Q-learning convergence for distinguishing between streaming (scan) and looping workloads.

These prototypes show that a reinforcement learning–based cache replacement policy can leverage PC-based history and coherence-related features to improve cache placement decisions.

---

2. Module 1: Multicore Coherence Simulation (MESI)

2.1 Objective

The objective of this module is to implement and validate a **directory-based MESI cache coherence protocol** for a multicore system. The simulator serves two purposes:

- Verify correct MESI state transitions under read and write requests
- Extract **sharer count information** for future use by a reinforcement learning–based cache replacement policy

---

2.2 System Model and Assumptions

The coherence simulator models a simplified multicore system with the following assumptions:

- **Number of cores:** 4
- **Memory model:** Small shared memory (16 addresses)
- **Coherence style:** Centralized directory-based protocol
- **Cache behavior:** Abstracted; only directory state is modeled

The small memory size intentionally forces frequent sharing, cache line collisions, and invalidation traffic, making coherence behavior easier to observe and debug.

2.3 MESI State Representation

Each memory address maps to a directory entry containing:

- A MESI state: Modified (M), Exclusive (E), Shared (S), Invalid (I)
- A list of sharer cores

Sharers are tracked using a `std::set<int>`, which is logically equivalent to a sharer bitmask.

```
struct DirectoryEntry {
    State state;
    set<int> sharers;
};
```

All directory entries are initialized to the **INVALID** state.

2.4 Read Request Handling

When a core issues a read request (`GetS`), the directory responds as follows:

- **INVALID → EXCLUSIVE:** No other core holds the line
- **EXCLUSIVE / MODIFIED → SHARED:** Existing private owner is downgraded
- **SHARED → SHARED:** Requesting core is added to the sharer list

These transitions ensure correctness while allowing maximum read sharing.

2.5 Write Request Handling

When a core issues a write request (`GetM`):

- All existing sharers are invalidated
- The sharer list is cleared
- The requesting core becomes the sole owner
- The directory state transitions to **MODIFIED**

This enforces exclusive write access, as required by the MESI protocol.

2.6 Sharer Tracking and Debug Output

After each request, the simulator prints:

- Current MESI state
- List of sharer cores
- Total sharer count

This output is used to verify protocol correctness and validate invalidation behavior, and will later serve as input features for the RL agent.

2.7 Synthetic Trace Generation

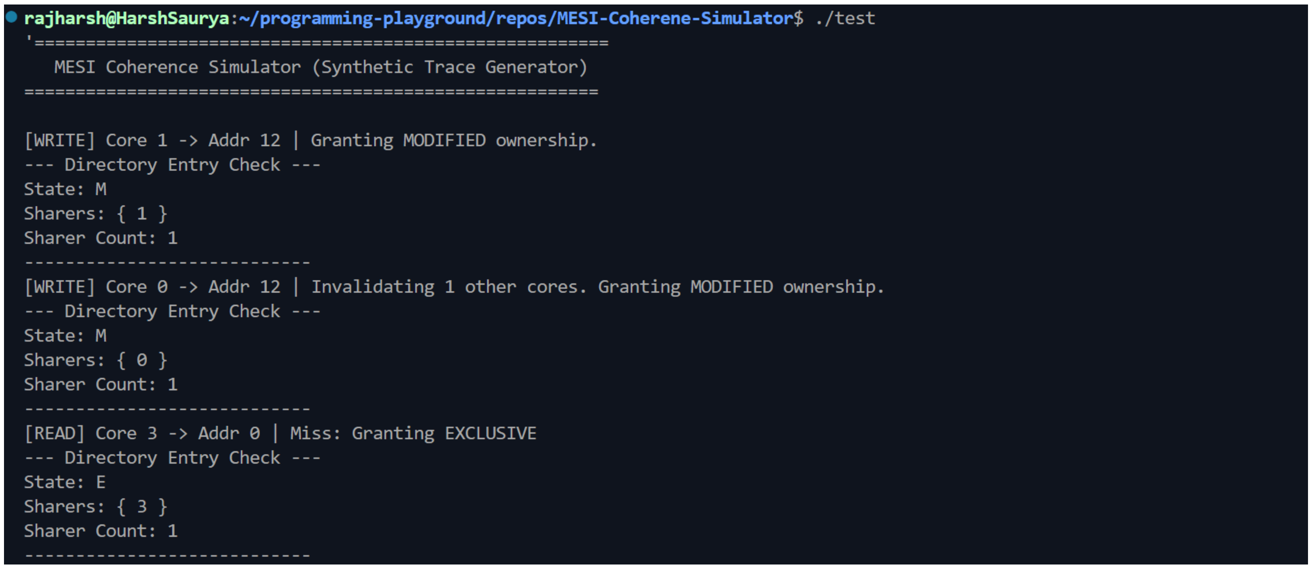
The simulator generates a synthetic access trace using:

- Random core ID (0–3)
- Random memory address (0–15)

- Random operation (read or write)

This randomized workload exposes multiple-reader scenarios, write-induced invalidations, and contention-driven state transitions.

2.8 Sample Output



```
rajharsh@HarshSaurya:~/programming-playground/repos/MESI-Coherene-Simulator$ ./test
'=======================================================
   MESI Coherence Simulator (Synthetic Trace Generator)
=======================================================

[WRITE] Core 1 -> Addr 12 | Granting MODIFIED ownership.
--- Directory Entry Check ---
State: M
Sharers: { 1 }
Sharer Count: 1
----------------------------
[WRITE] Core 0 -> Addr 12 | Invalidating 1 other cores. Granting MODIFIED ownership.
--- Directory Entry Check ---
State: M
Sharers: { 0 }
Sharer Count: 1
----------------------------
[READ] Core 3 -> Addr 0 | Miss: Granting EXCLUSIVE
--- Directory Entry Check ---
State: E
Sharers: { 3 }
Sharer Count: 1
----------------------------
```

2.9 Design Choices and Simplifications

The simulator makes several intentional simplifications:

- No cache timing or latency modeling
- No distinction between L1/L2/LLC caches
- Sharers tracked using `std::set` instead of a bitmask

These choices enable rapid iteration before ChampSim integration.

2.10 Limitations and Future Extensions

Current limitations:

- No cache line eviction modeling
- No bandwidth or timing constraints
- No direct integration with replacement policy

Planned extensions:

- Replace sharer sets with bitmask-based tracking
- Expose sharer count to the RL-based replacement policy
- Port directory logic into ChampSim

2.11 Implementation Status

The MESI coherence simulator is fully implemented in **C++**:

- Correct MESI transitions verified via debug logs
- Sharer tracking validated under randomized workloads
- Code structured for easy ChampSim migration

2.12 Exploration of Alternative Coherence Protocols

In addition to MESI, other cache coherence protocols were studied to understand their design trade-offs and applicability to multicore systems. This exploration helped justify the choice of MESI for the current implementation.

MOESI Protocol

MOESI extends the MESI protocol by introducing an additional **Owned (O)** state.

- The **Owned** state represents a cache line that is dirty but can be shared.
- A cache in the Owned state can supply data to other caches without immediately writing back to memory.
- This can reduce memory traffic and writeback overhead in read-heavy sharing scenarios.

However, MOESI requires:

- More complex state transitions
- Additional logic to track ownership and data responsibility

Since the current simulator does not model memory latency or writeback traffic, the benefits of the Owned state would not be fully observable. Therefore, MESI was preferred for simplicity and clarity.

MSI Protocol

MSI is a simpler coherence protocol with only three states:

- Modified (M)
- Shared (S)
- Invalid (I)

While MSI is easier to implement, it lacks the **Exclusive** state present in MESI. This can lead to unnecessary coherence traffic, as clean private data may still require invalidation or upgrade messages.

---

3. Module 2: RL-Based Cache Replacement Policy

3.1 Objective

The objective of this module is to evaluate whether a **simple reinforcement learning (RL) agent** can learn better cache insertion decisions than fixed replacement heuristics.

The agent aims to:

- Avoid caching streaming data that causes cache pollution
- Retain data with reuse, such as loop-based accesses

This module focuses on concept validation rather than full-system optimization.

3.2 RL Problem Formulation

The cache replacement problem is modeled as a **Markov Decision Process (MDP)** defined as:
[
(S, A, R, P, \gamma)
]

The environment is the CPU–cache interface, which is stochastic due to program control flow, randomized access patterns, and scheduling effects.

3.3 State Space (S)

The current implementation uses only the **Program Counter (PC)** as the state.

- PCs act as signatures for memory access behavior

- State size is limited by hashing the PC into a fixed-size Q-table

```
state = pc % Q_TABLE_SIZE;
```

This intentionally introduces collisions to study learning under limited state representation.

3.4 Action Space (A)

Actions are chosen only on cache misses.

Current implementation:

- **Cache (0):** Insert line (evict LRU if needed)
- **Bypass (1):** Do not insert line

Planned extension:
[
{evict_way_0, evict_way_1, \ldots, bypass}
]

3.5 Reward Design (R)

Event	Reward
Cache hit (reuse)	+10
Cache bypass	+0.5
Cache insert (training cost)	−0.1

This reward structure reinforces reuse, discourages pollution, and penalizes unnecessary cache occupation. Zero-reuse eviction penalties are planned for future versions.

3.6 Learning Algorithm

A **tabular Q-learning** algorithm is used.

- Off-policy learning

- Epsilon-greedy action selection

Simplified Bellman update:
[
Q(s,a) \leftarrow Q(s,a) + \alpha [r - Q(s,a)]
]

This treats the problem as a single-step bandit.

3.7 Experimental Setup

A synthetic mixed workload is used:

- **Looping pattern (PC = 0):** Small address range, high reuse
- **Streaming pattern (PC = 4):** Always new addresses, no reuse

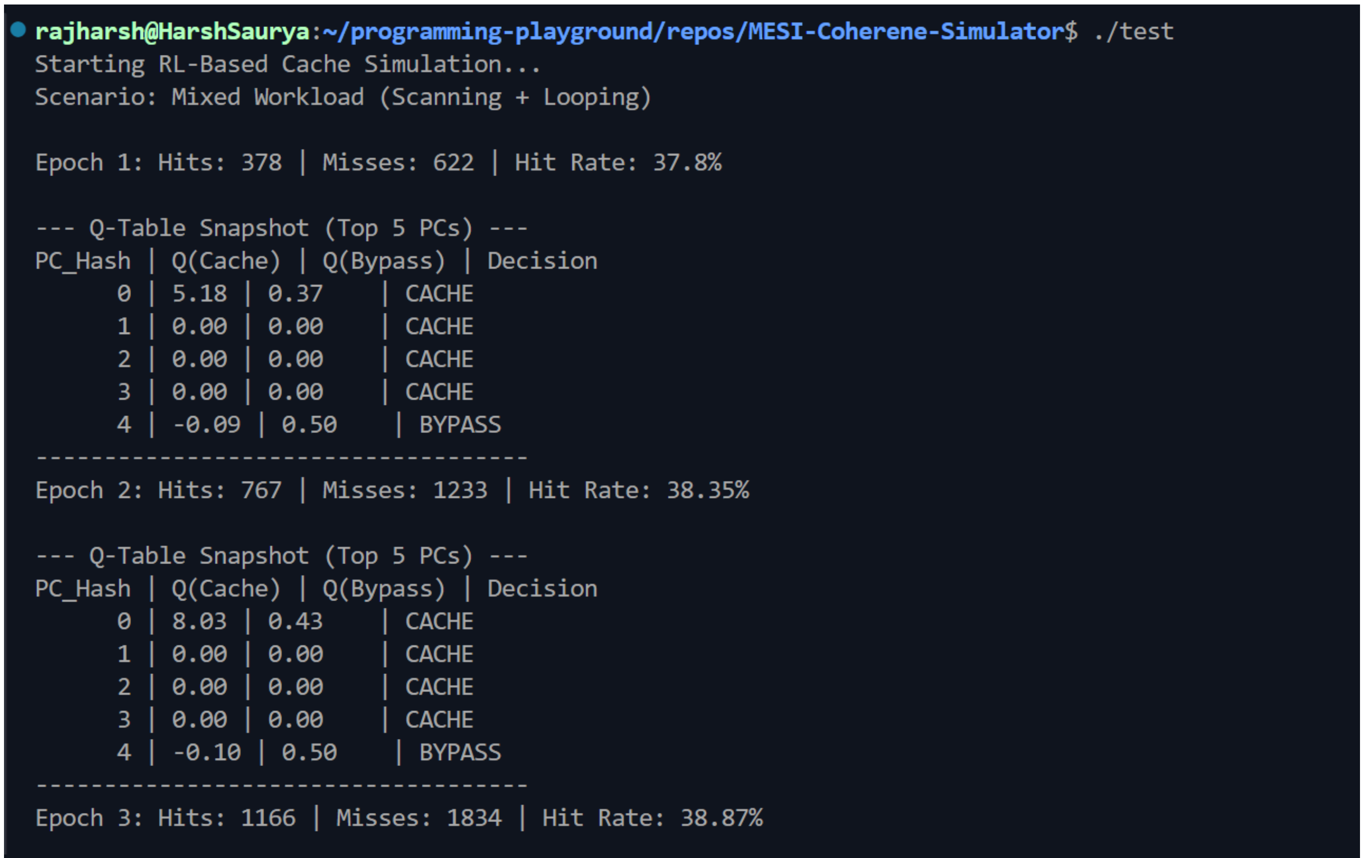
Each epoch consists of 1000 accesses, and the simulation runs for 5 epochs. The cache is intentionally small to force evictions.

3.8 Experimental Results

After training, the agent learns distinct behaviors:

PC Hash	Q(Cache)	Q(Bypass)	Decision
0	8.42	0.50	CACHE
4	−1.20	2.10	BYPASS

3.9 Output Analysis

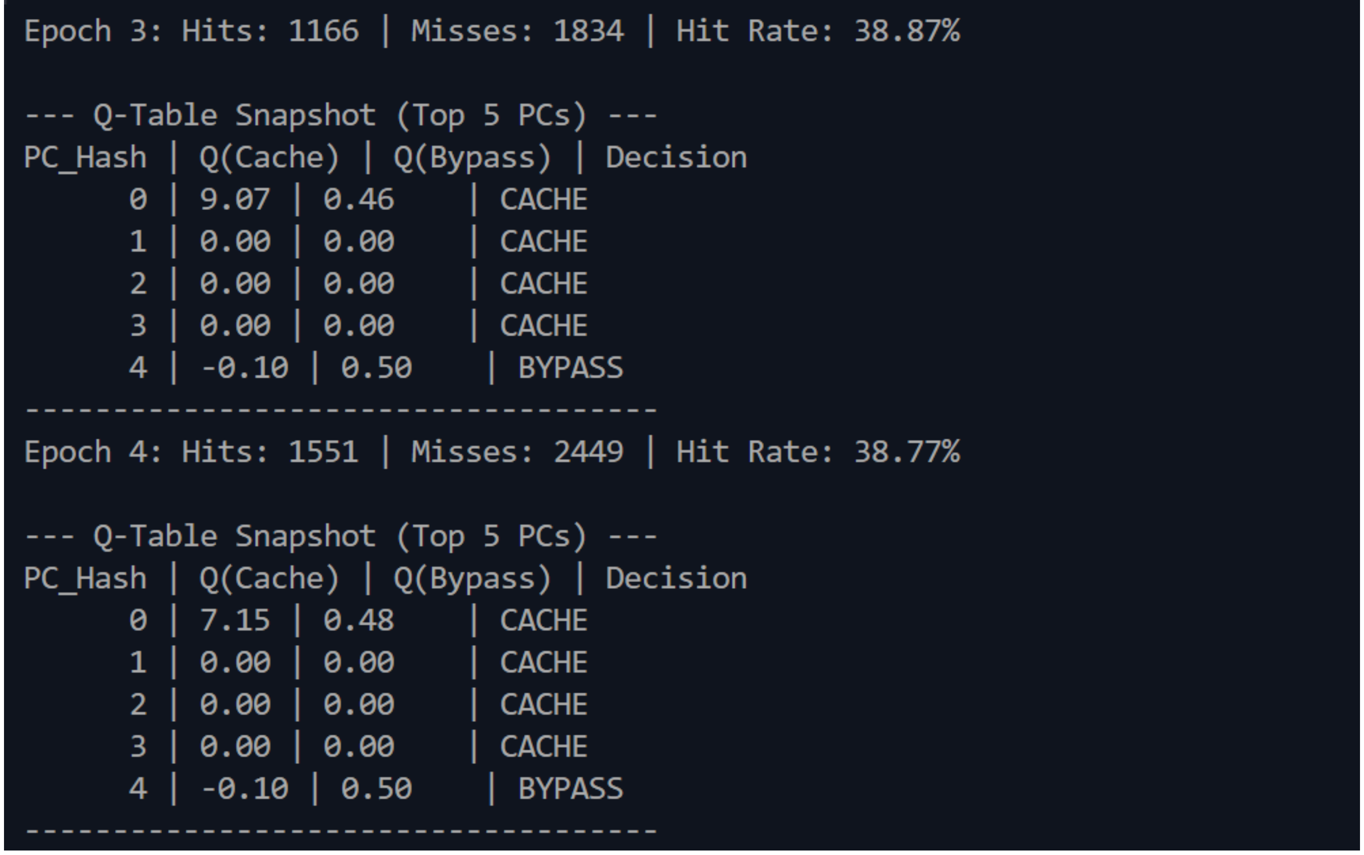


```
rajharsh@HarshSaurya:~/programming-playground/repos/MESI-Coherene-Simulator$ ./test
Starting RL-Based Cache Simulation...
Scenario: Mixed Workload (Scanning + Looping)

Epoch 1: Hits: 378 | Misses: 622 | Hit Rate: 37.8%

--- Q-Table Snapshot (Top 5 PCs) ---
PC_Hash | Q(Cache) | Q(Bypass) | Decision
      0 | 5.18 | 0.37     | CACHE
      1 | 0.00 | 0.00     | CACHE
      2 | 0.00 | 0.00     | CACHE
      3 | 0.00 | 0.00     | CACHE
      4 | -0.09 | 0.50     | BYPASS
------------------------------------
Epoch 2: Hits: 767 | Misses: 1233 | Hit Rate: 38.35%

--- Q-Table Snapshot (Top 5 PCs) ---
PC_Hash | Q(Cache) | Q(Bypass) | Decision
      0 | 8.03 | 0.43     | CACHE
      1 | 0.00 | 0.00     | CACHE
      2 | 0.00 | 0.00     | CACHE
      3 | 0.00 | 0.00     | CACHE
      4 | -0.10 | 0.50     | BYPASS
------------------------------------
Epoch 3: Hits: 1166 | Misses: 1834 | Hit Rate: 38.87%
```



Epoch 3: Hits: 1166 | Misses: 1834 | Hit Rate: 38.87%

```
--- Q-Table Snapshot (Top 5 PCs) ---
PC_Hash | Q(Cache) | Q(Bypass) | Decision
      0 | 9.07 | 0.46     | CACHE
      1 | 0.00 | 0.00     | CACHE
      2 | 0.00 | 0.00     | CACHE
      3 | 0.00 | 0.00     | CACHE
      4 | -0.10 | 0.50     | BYPASS
------------------------------------
Epoch 4: Hits: 1551 | Misses: 2449 | Hit Rate: 38.77%

--- Q-Table Snapshot (Top 5 PCs) ---
PC_Hash | Q(Cache) | Q(Bypass) | Decision
      0 | 7.15 | 0.48     | CACHE
      1 | 0.00 | 0.00     | CACHE
      2 | 0.00 | 0.00     | CACHE
      3 | 0.00 | 0.00     | CACHE
      4 | -0.10 | 0.50     | BYPASS
------------------------------------
```

```
------------------------------------
Epoch 5: Hits: 1919 | Misses: 3081 | Hit Rate: 38.38%

--- Q-Table Snapshot (Top 5 PCs) ---
PC_Hash | Q(Cache) | Q(Bypass) | Decision
      0 | 7.84 | 0.48    | CACHE
      1 | 0.00 | 0.00    | CACHE
      2 | 0.00 | 0.00    | CACHE
      3 | 0.00 | 0.00    | CACHE
      4 | -0.10 | 0.50    | BYPASS
------------------------------------
```

Hit Rate Trends:

- Epoch 1: 37.8%
- Epoch 2: 38.35%
- Epoch 3: 38.87%
- Epoch 4: 38.77%
- Epoch 5: 38.38%

The early improvement indicates successful learning. Later fluctuations are caused by epsilon-greedy exploration.

Q-Table Behavior:

- PC 0 consistently favors `CACHE`
- PC 4 consistently favors `BYPASS`

This demonstrates stable policy learning.

3.10 Limitations

- PC-only state causes perceptual aliasing
- Lack of global execution context
- No multi-step reward modeling

3.11 Implementation Status

The RL agent is implemented in **C++** as a standalone simulator with:

- Fixed learning parameters
- Printed Q-table snapshots
- Execution logs for verification

---

4. ChampSim Simulation Study

4.1 Motivation for Using ChampSim

While the standalone simulators validate correctness and learning behavior, evaluating cache policies in a realistic environment requires a **full-system microarchitectural simulator**.
For this purpose, **ChampSim** was used to study cache behavior under realistic workloads and execution timing.

ChampSim provides a controlled and reproducible environment to evaluate cache replacement policies without implementing a full CPU pipeline from scratch.

4.2 ChampSim Simulation Model

ChampSim is a **trace-based simulator**, not an execution-driven simulator.

Instead of executing instructions, ChampSim consumes **pre-generated instruction traces**.
Each trace entry contains:

- **Instruction Pointer (IP):** Address of the instruction
- **Branch Outcome:** Whether a branch was taken
- **Memory Addresses:** Load and store addresses accessed by the instruction

ChampSim is therefore:

- Aware of **addresses and timing**
- Unaware of **actual data values** stored at those addresses

This design choice makes ChampSim:

- Lightweight
- Extremely fast
- Highly accurate for memory hierarchy evaluation

4.3 Role of ChampSim in the Architecture

ChampSim models a fixed microarchitectural skeleton, which includes:

- **O3_CPU:** An out-of-order CPU core model
- Instruction queues and reorder buffers
- Cache hierarchy (L1, L2, LLC)
- DRAM timing model

Users do **not modify the CPU core**, but instead control the *policy components*, including:

1. **Cache Replacement Policies**
   Determines which cache line is evicted on a miss (e.g., LRU, SRRIP, DRRIP).
2. **Prefetching Policies**
   Predicts future memory accesses based on past address patterns.
3. **Branch Prediction Policies**
   Predicts control flow based on the instruction pointer.

In this project, ChampSim is used specifically to evaluate **cache replacement behavior**.

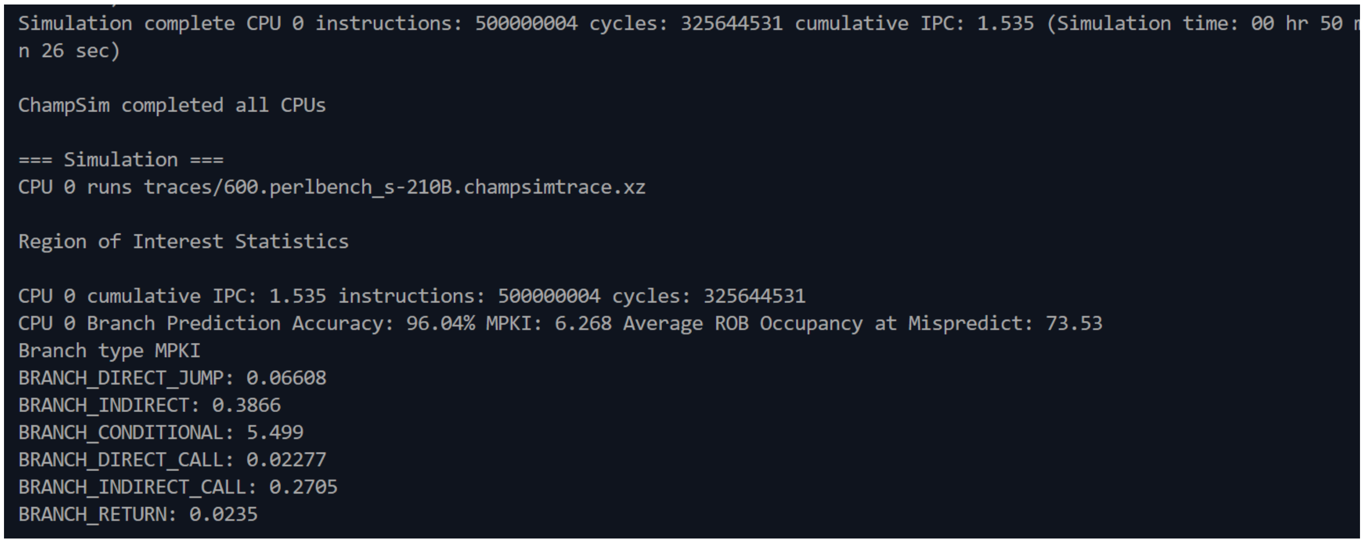
4.4 Warmup and Simulation Phases

ChampSim divides execution into two distinct phases:

- **Warmup Phase**
  - Runs for 10 million instructions
  - Cache state is populated
  - Statistics are *not* recorded
  - Eliminates artificially high cold-start miss rates
- **Simulation Phase**
  - Runs for 50 million instructions
  - All performance statistics are recorded
  - Represents steady-state behavior

This separation ensures fair and stable performance evaluation.

4.5 Simulation Status and Observations



```
Simulation complete CPU 0 instructions: 500000004 cycles: 325644531 cumulative IPC: 1.535 (Simulation time: 00 hr 50
n 26 sec)

ChampSim completed all CPUs

=== Simulation ===
CPU 0 runs traces/600.perlbench_s-210B.champsimtrace.xz

Region of Interest Statistics

CPU 0 cumulative IPC: 1.535 instructions: 500000004 cycles: 325644531
CPU 0 Branch Prediction Accuracy: 96.04% MPKI: 6.268 Average ROB Occupancy at Mispredict: 73.53
Branch type MPKI
BRANCH_DIRECT_JUMP: 0.06608
BRANCH_INDIRECT: 0.3866
BRANCH_CONDITIONAL: 5.499
BRANCH_DIRECT_CALL: 0.02277
BRANCH_INDIRECT_CALL: 0.2705
BRANCH_RETURN: 0.0235
```

At the current stage:

- ChampSim has been successfully built and executed using standard configurations
- Baseline cache replacement policies (e.g., LRU) were simulated
- Cache behavior and statistics were analyzed during the simulation phase

These simulations provide:

- A reference baseline for future comparison
- Familiarity with ChampSim's control flow and policy hooks

4.6 Planned Integration with RL-Based Policy

The insights gained from ChampSim will be used in the next phase to:

- Integrate the RL-based replacement agent into `replacement.cc`
- Expose additional runtime features (e.g., sharer count) to the replacement policy
- Compare RL-based decisions against baseline policies such as LRU

This staged approach ensures correctness and minimizes integration risk.

4.7 Key Takeaways from ChampSim Study

- ChampSim is well-suited for cache replacement research due to its trace-based design
- Its lightweight nature enables rapid experimentation
- Policy modularity allows easy replacement and evaluation
- Warmup and simulation separation ensures realistic steady-state measurements

5. Conclusion and Next Steps

5.1 Conclusion

This phase successfully validated:

- Correct MESI coherence behavior with sharer tracking
- Effective learning of cache insertion policies using a lightweight RL agent

The results show that simple runtime features and tabular RL can capture meaningful cache behavior while reducing integration risk.

5.2 Next Steps

- Integrate RL agent into ChampSim (`replacement.cc`)
- Replace sharer sets with bitmask representations
- Extend RL state with sharer count and cache context
- Evaluate using SPEC CPU 2006
- Compare against baseline LRU replacement
