Implementation and Analysis of Mockingjay and RT-RRIP Cache Replacement Policies in ZSim

Zach Assad, Brycen Craig, Rodrigo Orozco (Group 9)
Texas A&M University
Department of Computer Science and Engineering
{xachariah, bec2594, rodrigoorozco03}@tamu.edu

Abstract—**This report presents the implementation and evaluation of two different cache replacement policies, Mockingjay and RT-RRIP, using the ZSim simulator. We evaluated these policies on the last-level cache (LLC) using SPEC CPU2006 and PARSEC benchmark suites. Mockingjay is designed to effectively and faithfully mimic Belady's MIN policy through multiclass prediction of reuse distances, while RT-RRIP extends the SRRIP policy with reuse time awareness. Our experimental results are compared with existing policies in ZSim, including LRU, NRU, and SRRIP. We analyze the performance differences in terms of LLC misses per kilo instructions (MPKI) and instructions per cycle (IPC), demonstrating that our implementations offer significant improvements over traditional policies across a variety of workloads.**

I. Introduction

In the computer memory hierarchy, caching plays a crucial role in improving overall system performance. The effectiveness of cache memory is largely determined by the replacement policy used, which decides which cache lines to evict when space is needed for new data. The LLC in particular is largely the target of these policies, as a single cache miss can result in hundreds of wasted clock cycles. Considering that the LLC is often shared among different cores, even small performance improvements in it can lead to huge overall performance benefits. Due to the increase in size of memory and complexity of memory accesses, good replacement decisions have become increasingly important.

Traditional replacement policies like Least Recently Used (LRU) are generally effective, but they fall short in workloads with irregular access patterns or those with working sets larger than the cache size. Addressing these challenges has led to innovative and more intelligent policies that can better determine which lines to evict and keep.

In this report, we discuss our implementation and evaluate two modern cache replacement policies—Mockingjay and RT-RRIP—in the ZSim simulator. ZSim is a fast x86-64 simulator used for modeling the behavior of memory subsystems. We will compare the performance of these policies against existing ones in ZSim (such as LRU, NRU, and SRRIP) using SPEC CPU2006 and PARSEC benchmark suites. We will analyze the results and discuss the basis for their behavior.

II. Problem Description

As the gap between memory and processor speed increases, efficient cache replacement policies are needed, especially for the LLC. Optimizing performance in the LLC in particular is important since it is the last barrier before accessing the slower main memory. This means that a cache miss can be very costly in terms of performance and efficiency. The goal of this paper is to design the best-performing cache replacement policy for a last-level cache using the ZSim simulator. This involves determining which cache line to evict when the cache is full and a new line needs to be brought in, and which cache lines to keep. Since different applications can have all sorts of different access patterns, the policy needs to be able to adapt to them.

The ideal cache replacement policy, known as Belady's MIN Policy, always evicts the block that will be used farthest in the future. That is, it has knowledge of the future. The goal is to design a replacement policy that approximates this ideal policy as closely as possible without violating causality (i.e., without using knowledge of the future). This requires predicting which cache lines are least likely to be reused in the near future based solely on past and present access patterns.

In this report, we focus on implementing two policies—Mockingjay and RT-RRIP—in the ZSim simulator environment. Mockingjay attempts to mimic Belady's MIN through reuse distance prediction, while RT-RRIP enhances the SRRIP policy with recency time awareness. Our challenge is not only to faithfully implement these policies according to their original designs but also to adapt them to work efficiently within the constraints of the ZSim simulator architecture.

III. Background and Related Work

A. Cache Replacement Fundamentals

Cache replacement policies are the hardware protocols that decide which cache lines to keep or evict when space is needed for new data. The goal is to reduce cache misses by accurately predicting which lines to evict. The L3 cache, also known as the last-level cache (LLC), is a focus of these policies, since it is shared among different cores and has a more expensive penalty for cache misses. Small improvements in the LLC can lead to large performance increases in the overall system.The theoretical optimal policy, known as Belady's MIN Policy [2], would evict the cache block candidate that is referenced furthest in the future. However, this requires knowledge of future accesses, which is impractical. Therefore, we must rely on behavior from past access patterns in an attempt to accurately predict future behavior.

B. Existing Replacement Policies

Below are several cache replacement policies that are used in some systems:

- **Least Recently Used (LRU)**: Evicts the cache line that has not been accessed for the longest time.
- **Least Frequently Used (LFU)**: Evicts the least frequently accessed cache line.
- **Not Recently Used (NRU)**: A simplified approximation of LRU that uses a single reference bit to track whether a line has been recently accessed.
- **Random Replacement**: Randomly selects a cache line for eviction.
- **Static RRIP (SRRIP)** [3]: Uses re-reference interval prediction values to estimate how soon a cache line will be reused, evicting lines predicted to be reused furthest in the future.

C. Mockingjay

Mockingjay was recently introduced by Shah et al. [5] in their 2022 paper ”Effective Mimicry of Belady's MIN Policy.” Its goal is to mimic the MIN policy. Many previous policies tried to mimic the policy in order to provide the best results, but they deviated from truly mimicking it, by combining predicted ETA with age-based information, which decreased optimal performance. Many policies also use binary classification, but this is more reactive to prediction errors, and can lead to more misclassifications.

Mockingjay uses a PC-based, multiclass predictor to learn each cache line's reuse distance. This allows for a more fine-grained, stable, and accurate prediction. Based on the predicted reuse time, it evicts the line that is predicted to be reused furthest in the future. This is achieved through 3 key components: the Sampled Cache, which tracks past reuse distances to train the RDP, the Reuse Distance Predictor (RDP), a PC-based predictor that learns reuse distances for loads initiated by a given PC, and the ETA Counters, which is stored in the LLC and maintains the estimated time of arrival (ETA) for each line, decreasing over time.

Mockingjay's multiclass prediction approach is more resilient to prediction errors than binary classification because a single misprediction has a smaller impact on the predicted ordering of cache lines.

D. RT-RRIP

RT-RRIP (Recency Time Based Re-Reference Interval Prediction) was introduced by Athni et al. [1] in their 2022 paper ”Improved Cache Replacement Policy based on Recency Time Re-Reference Interval Prediction.” It builds upon the established SRRIP policy by introducing a recency time prefilter that enhances replacement decisions.

The RT-RRIP policy operates in two stages:

- First, a recency time prefilter filters data blocks based on their recency time (when they were most recently accessed)
- Then, the RRIP predictor evaluates the subset of blocks provided by the prefilter and selects a victim based on Re-reference Prediction Values (RRPV)

After a cache miss, a filter threshold value is set based on the average recency time of all blocks. Blocks with a recency time less than the threshold value are fed to the predictor. This approach helps reduce cache misses and miss latency by making more informed eviction decisions, particularly for workloads with mixed access patterns.

IV. Implementation

A. ZSim Simulator Overview

ZSim [4] is a fast and accurate x86-64 simulator designed for simulating memory subsystems with thousands of cores. It provides a nice framework for implementing and evaluating various components of computer systems, including cache replacement policies. With ZSim's cache implementation, you can flexibly define replacement policies through its ReplPolicy interface.

The simulator's cache implementation consists of several key components:

- Cache: The main cache class that handles accesses, invalidations, and replacements
- CacheArray: Implementations of different cache array organizations (set-associative, Z-cache)
- FilterCache: An optimization layer that significantly reduces overhead in the L1 cache.
- ReplPolicy: The interface for replacement policies
- CC: Coherence controllers to maintain cache coherence. Uses MESI protocol.

To implement a new replacement policy in ZSim, we need to create a new class that inherits from the `ReplPolicy` interface and implements its key methods.

B. Implementing Mockingjay fix

Our implementation of Mockingjay in ZSim follows the design described in the original paper by Shah et al, as well as Shah's more detailed ChampSim implementation. We created a new class `MockingjayReplPolicy` that inherits from `ReplPolicy` and implements the following key components:

- **Estimated Time Remaining (ETR) counters**: Per-cache-line counters that track the estimated time until a cache line is reused
- **Reuse Distance Predictor (RDP)**: A PC-based predictor that learns the reuse distance of cache lines
- **Sampled Cache**: A small cache that tracks access patterns to train the RDP
- **Per-set clocks**: Used to age ETR values of cache lines within a set

The implementation includes several adaptations to fit within ZSim's architecture:

- PC signatures are created by hashing program counters to fit within the predictor's limited size
- ETR values are quantized to save space while maintaining prediction accuracy

- Special handling is included for writebacks and prefetches

The replacement decision in Mockingjay is based on the ETR values of cache lines within a set. The line with the largest absolute ETR value (either positive or negative) is selected for eviction, with ties broken in favor of lines with negative ETR (those that have exceeded their expected reuse time).

C. Implementing RT-RRIP fix

We implemented the RT-RRIP policy as described in the paper by Athni et al. The implementation consists of:

- **RRPV Array**: Stores the Re-reference Prediction Value for each cache line
- **Recency Time Array**: Tracks the most recent access time for each cache line
- **Recency Time Threshold**: Calculated as the average recency time of candidate lines

The replacement algorithm works in two stages:

1) First, candidate cache lines are filtered based on their recency time. Only lines with recency time less than or equal to the threshold (average recency time) are considered for eviction.
2) Then, among the filtered candidates, the algorithm selects the line with the highest RRPV value for eviction. If no line has reached the maximum RRPV, all filtered lines are aged (RRPV incremented) until at least one reaches the maximum value.

New blocks are inserted with an RRPV of (RRPV_MAX - 1). On a hit, the block's RRPV is reset to 0 (indicating recent use) and the block's recency time value is updated.

D. Implementation Challenges

During our implementation, we encountered a couple challenges:

- **Adapting to ZSim's architecture**: Mockingjay required significant adaptation to fit within ZSim's replacement policy interface and memory hierarchy model. This also required adapting the ZSim architecture itself. The main challenge was propagating the program counter (PC) addresses of the instructions that made the memory request to be used in the LLC for the `MockingjayReplPolicy` class.
- **Parameter tuning**: Finding the optimal parameters for Mockingjay required lots of experimentation and research. It was important to choose appropriate values. For example, the length of the PC signature or the maximum ETR. value.
- **Memory overhead**: Both policies require additional metadata per cache line, which needed to be carefully managed to avoid excessive memory usage in simulation and memory leaks This can cause lots of inaccuracies and failures when simulating results.

V. Evaluation Methodology

A. System Configuration

For our evaluation, we simulated single-core and 8-core systems for SPEC and PARSEC benchmarks, respectively. Each core was a Westmere out-of-order (OOO) execution type. For single-core systems, a cache size of 2MB was used, while 8MB was used for 8-core. 16-way set-associative caches were used for all systems, with a line size of 64 bytes.

Our experiments focus on evaluating the replacement policies for the shared L3 cache (LLC) while keeping the L1 and L2 caches with fixed replacement policies (LRU).

B. Benchmarks

We evaluated our implementations using the following benchmarks:

SPEC CPU2006 Benchmarks (Single-threaded):

- Integer: bzip2, gcc, mcf, hmmer, xalancbmk, sjeng, libquantum
- Floating Point: cactusADM, namd, calculix, soplex, lbm

PARSEC Benchmarks (Multi-threaded):

- blackscholes, bodytrack, fluidanimate, streamcluster, swaptions, canneal, x264

Each SPEC and PARSEC benchmark was run for 500 million and 5 billion instructions, respectively.

C. Metrics

We used the following metrics to evaluate the performance of the replacement policies:

- **LLC MPKI (Misses Per Kilo Instructions)**: Measures the number of cache misses per thousand instructions, directly reflecting the effectiveness of the replacement policy.
- **IPC**: Measures the average number of instructions executed per cycle.

We compare our implemented policies (Mockingjay and RT-RRIP) against the following policies:

- Least Recently Used (LRU)
- Not Recently Used (NRU)
- Static RRIP (SRRIP)
- Random Replacement
- Least Frequently Used (LFU)
- TreeLRU
- Vantage

VI. Results

A. Overall Performance

Figures 1, 2, and 3 show our overall performance metrics.

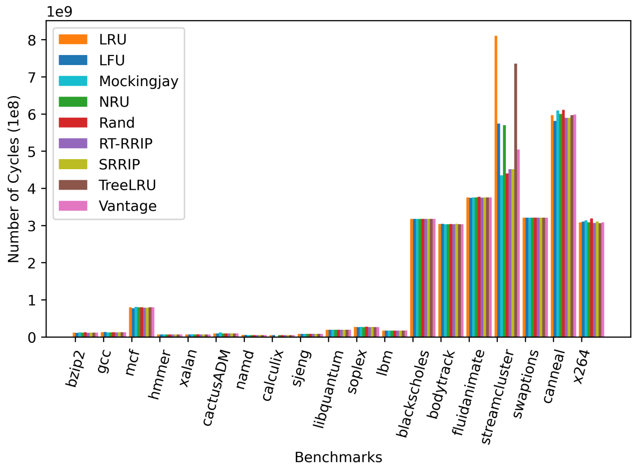

Fig. 1. Execution Cycles across Benchmarks

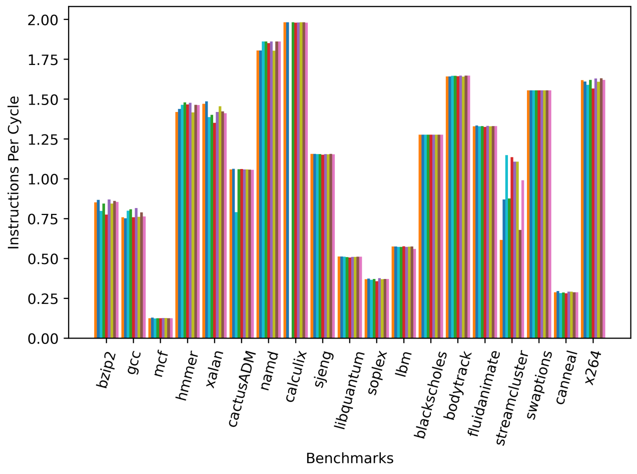

Fig. 2. Instructions Per Cycle

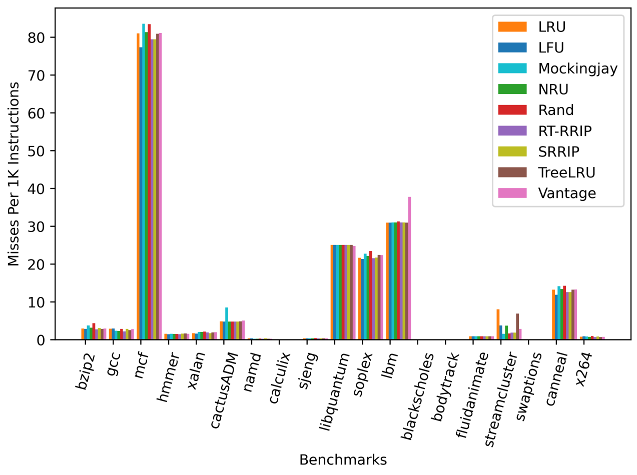

Fig. 3. Misses Per Thousand Instructions

B. SPEC CPU2006 Benchmarks

Figures 4, 5, and 6 show our results for single-threaded benchmarks.

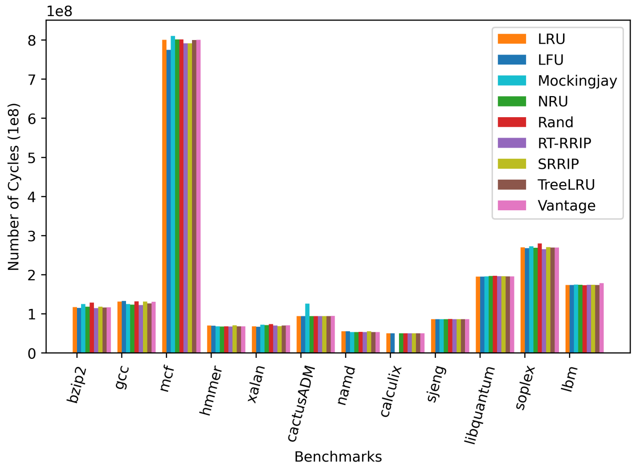

Fig. 4. Execution Cycles for SPEC CPU2006 Benchmarks

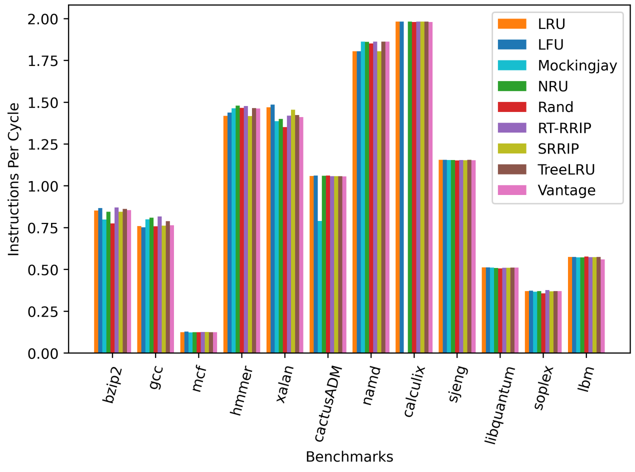

Fig. 5. IPC for SPEC CPU2006 Benchmarks

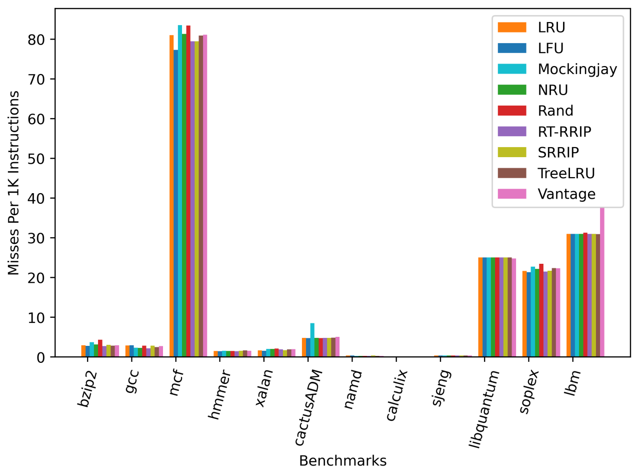

Fig. 6. LLC MPKI for SPEC CPU2006 Benchmarks

C. PARSEC Benchmarks

Figures 7, 8, and 9 show our results for multi-threaded benchmarks.

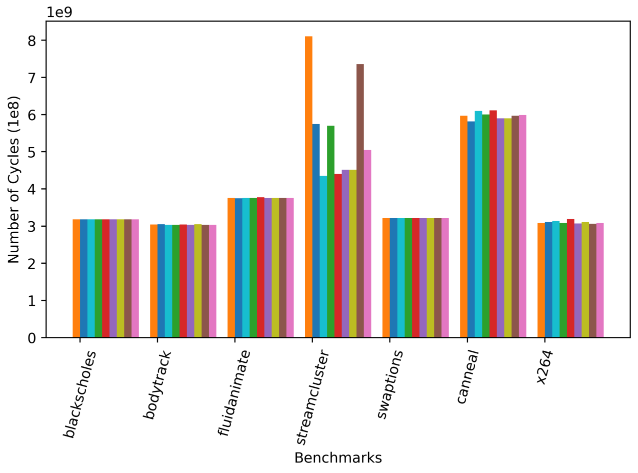

Fig. 7. Execution Cycles for PARSEC Benchmarks

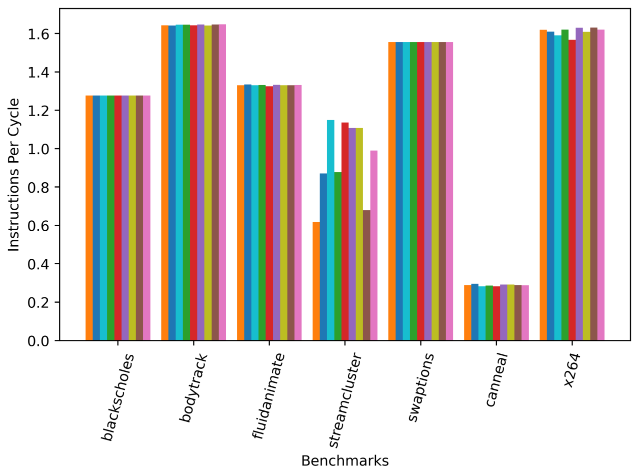

Fig. 8. IPC for PARSEC Benchmarks

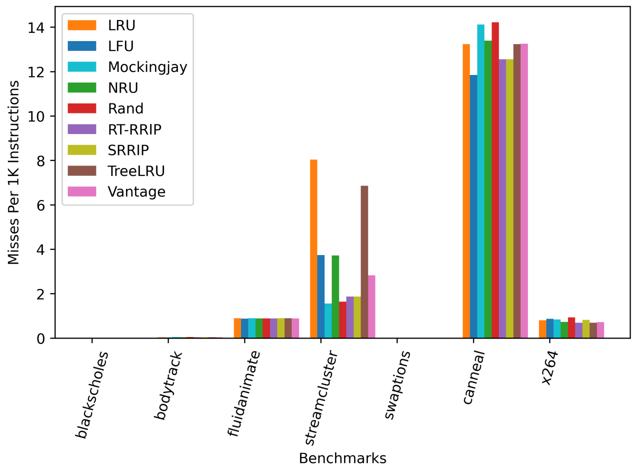

Fig. 9. LLC MPKI for PARSEC Benchmarks

Below is the speedup of the policies relative to LRU:

TABLE I
AVERAGE SPEEDUP OVER LRU (%)

Policy	**LFU**	**Mock.**	**NRU**	**Rand**	**RT-RRIP**	**SRRIP**	**TreeLRU**	**Vant.**
Speedup	2.7	3.0	2.6	3.3	5.1	4.1	1.0	3.2

D. Result Analysis

The results show that out of all the policies tested, RT-RRIP had the highest relative speedup of 5.1% as compared to LRU for the benchmarks we tested, while Mockingjay had a relative speedup of 3%. This is relatively consistent with the findings of Shah and Athni et al.

In most cases, Mockingjay's IPC is comparable or better than LRU's. However, it underperforms in certain tests like xalan, where workloads may be static and simple heuristics would suffice instead. In memory-intensive benchmarks like streamcluster, Mockingjay thrives. Since data in streamcluster is often a one time use, Mockingjay is perfect for preventing those blocks from polluting the cache. Compared to LRU, Mockingjay had a massive 85% improvement in IPC and a significant reduction in MPKI. However, it does have slightly more misses in a few benchmarks compared to LRU, likely due to the static nature of the policies. Overall, it is a nice middle ground for accuracy and complexity, but choosing when to use it depends on the workload being tested. RT-RRIP also had improvements over streamcluster. This is likely attributed to its deprioritization of blocks it deems less likely to be reused. Also, since RT-RRIP can switch between aggressive and conservative insertion strategies based on recent feedback, it is more resilient to phase changes and varying temporal locality patterns, which explains its strong average performance across diverse workloads.

VII. Discussion

A. Comparison of Mockingjay and RT-RRIP

Mockingjay and RT-RRIP share the common goal of improving cache performance by making more informed replacement decisions. However, their approaches differ. Mockingjay uses a PC-based predictor and sampled cache to predict reuse distances, while RT-RRIP employs a recency time prefilter to enhance the SRRIP policy. In our experiments, Mockingjay performed better in workloads with irregular or phase-based access patterns, such as streamcluster, where PC-level reuse prediction allowed it to bypass cache pollution effectively. RT-RRIP, on the other hand, excelled in workloads with moderate temporal locality and consistent reuse behavior, such as gcc and hmmer, where its recency-based insertion policy and filtering were sufficient to prioritize frequently reused blocks. By evicting lines with the largest predicted reuse distance, Mockingjay minimizes the chance of evicting lines that will be reused soon. On the other hand, RT-RRIP's recency time prefilter helps prioritize lines that have been recently accessed, reducing the likelihood of evicting frequently used lines.

B. Potential Improvements

To further enhance the performance of Mockingjay and RT-RRIP in the future, we could perhaps dynamically adjust the RDP size or prefilter threshold based on workload characteristics.

VIII. Conclusion

In this paper, we implemented and evaluated two cache replacement policies-Mockingjay and RT-RRIP-in the ZSim simulator. Our results show that these two policies outperform the traditional LRU policy across most benchmarks. Mockingjay's PC-based reuse distance prediction and RT-RRIP's recency time awareness prove to be effective strategies for making informed replacement decisions. Despite some implementation limitations, the performance gains achieved by Mockingjay and RT-RRIP highlight the potential for advanced replacement policies to significantly improve cache efficiency. Overall, our work contributes to the understanding of cache replacement policies and their impact on system performance. The insights gained from this study can guide future research and development efforts in cache design and optimization.

Work Division

Our team divided the work for this project as follows:

Zach Assad:

- Implemented and debugged the Mockingjay replacement policy
- Modified ZSim to propagate PC addresses to the LLC
- Ran SPEC benchmarks and collected data
- Contributed to results analysis and report writing

Brycen Craig:

- Implemented and debugged the RT-RRIP replacement policy
- Set up configuration files for benchmarks
- Created visualization scripts for result analysis
- Compiled and organized benchmark results
- Contributed to report writing

Rodrigo Orozco:

- Ran PARSEC benchmarks and collected data
- Conducted literature review on replacement policies
- Created data analysis scripts
- Generated graphs and visualizations
- Contributed to report writing and editing

All team members communicated frequently to discuss progress, challenges, and results.

References

[1] C. R. Athni, V. Vinod Chippalkatti, A. Nandakumar, A. V. Nandana, and Y. J. Pavitra, "Improved cache replacement policy based on recency time re-reference interval prediction," in *2022 IEEE 7th International conference for Convergence in Technology (I2CT)*, 2022, pp. 1–6.

[2] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," vol. 5, no. 2, 1966, pp. 78–101.

[3] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," vol. 38, no. 3. New York, NY, USA: Association for Computing Machinery, Jun. 2010, p. 60–71.

[4] D. Sanchez and C. Kozyrakis, "Zsim: fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 475–486.

[5] I. Shah, A. Jain, and C. Lin, "Effective mimicry of belady's MIN policy," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 558–572.

Appendix

A. Mockingjay Implementation

Below is the key portion of our Mockingjay implementation, showing the core eviction decision algorithm:

```
//determining which cache line to evict
template <typename C>
uint32_t rank(const MemReq* req, C cands) {
    //loop through candidates
    for (auto ci = cands.begin(); ci !=
    cands.end(); ci.inc()) {
        if (!cc->isValid(*ci)) return *ci;
    //evict invalid
    }
    //variables to track the cache line with
    highest ETR
    int maxEtr = -1;
    int victimLoc = 0;
    for (auto ci = cands.begin(); ci !=
    cands.end(); ci.inc()) {
        if (abs(etr[*ci]) > maxEtr ||
            (abs(etr[*ci]) == maxEtr &&
    etr[*ci] < 0)) {
            victimLoc = *ci;//update victim
    line
            maxEtr = abs(etr[*ci]);//update
    max etr
        }
    }

    return victimLoc; //return victim line
}
```

B. RT-RRIP Implementation

Below is the key portion of our RT-RRIP implementation, showing the two-stage eviction process:

```
template <typename C> inline uint32_t
    rank(const MemReq* req, C cands) {
    // recency time filter
    vector<uint32_t> filteredCands;
    threshold = getThreshold(cands); //
    returns average recency time
    for (auto ci = cands.begin(); ci !=
    cands.end(); ci.inc()) {
        if (recencyTimeArray[*ci] <=
    threshold) {
            filteredCands.push_back(*ci); //
    gets all candidate with
                                          //
    recency time over the average
        }
    }
    if (filteredCands.empty()) {
        for(auto ci = cands.begin(); ci !=
    cands.end(); ci.inc()) {
            filteredCands.push_back(*ci);
        }
    }
    // RRIP
    while(true) {
        for (auto fi: filteredCands) {
            if (rrpvArray[fi] >= rrpvMax) {
                return fi; // return first
    candidate that exceeds rrpvMax
            }
        }
        // age filtered blocks if no
    candidate exceeds rrpvMax
        for (auto fi : filteredCands) {
            rrpvArray[fi]++;
        }
    }
}
```

C. ZSim Configuration

Our ZSim configuration for the LLC evaluation is as follows:

```
// System configuration
sys = {
    lineSize = 64;
    frequency = 2400;

    cores = {
        westmere = {
            type = "OOO";
            cores = 8;  // 1 for SPEC, 8 for
    PARSEC
            icache = "l1i";
            dcache = "l1d";
        };
    };

    caches = {
        l1d = {
            caches = 8;  // 1 for SPEC, 8 for
    PARSEC
            size = 32768;
            array = {
                type = "SetAssoc";
                ways = 8;
            };
            repl = {
                type = "LRU";
            };
        };

        l1i = {
            caches = 8;  // 1 for SPEC, 8 for
    PARSEC
            size = 32768;
            array = {
                type = "SetAssoc";
                ways = 8;
            };
            repl = {
                type = "LRU";
            };
        };

        l2 = {
            caches = 8;  // 1 for SPEC, 8 for
    PARSEC
            size = 262144;
            array = {
                type = "SetAssoc";
                ways = 8;
            };
            repl = {
                type = "LRU";
            };
        };

        l3 = {
            caches = 1;
            size = 16777216;  // 16MB for
    8-core, 2MB for single-core
            array = {
                type = "SetAssoc";
                ways = 16;
            };
            repl = {
                type = "Mockingjay";  // Or
    "RTRRIP", "LRU", "SRRIP", etc.
            };
        };
    };
};
```

D. Benchmark Command Examples

Below are examples of commands used to run our benchmarks:

```
# SPEC CPU2006 example:
 ./run-simulation SPEC namd Mockingjay
# PARSEC example:
 ./run-simulation PARSEC streamcluster RT-RRIP
```