CSCE 469/614: Effective Mimicry of Belady's MIN Policy

Lohitaksh Allampalli, Ben Armstrong, Mathimalar Mathivanan
Texas A&M University
Department of Computer Science and Computer Engineering

***Abstract*—Cache replacement policies have improved a lot in recent years, often using prediction to decide which cache lines to keep or remove. Some approaches use simple binary decisions, such as predicting whether a line will be reused soon or not, based on how often or how recently it was accessed. In this work, we introduce a new method that uses multiclass prediction to estimate how far in the future each cache line will be reused. This prediction is based on the program counter (PC) that brought the line into the cache. By using more detailed information, our policy can make better decisions about which lines to evict. It also handles prediction mistakes more effectively and gets closer to the behavior of the ideal Belady's MIN policy. Our results show that this method improves cache performance compared to traditional binary-based approaches.**

I. Introduction

Efficient cache management remains a central challenge in modern processor design, particularly as applications grow in complexity and working sets continue to expand. The effectiveness of a cache replacement policy significantly impacts system performance, as it determines which data to retain and which to evict when space is limited. Historically, simple heuristics such as Least Recently Used (LRU) and Most Recently Used (MRU) have been widely adopted due to their ease of implementation. However, these methods are often inadequate for dynamic workloads with irregular memory access patterns.

To address these shortcomings, researchers have explored prediction-based cache replacement techniques that aim to anticipate future reuse of cache lines. These methods shift away from fixed rules and instead use insights gathered from past behavior to guide replacement decisions. For instance, Khan et al. [1] proposed a machine learning-based policy that predicts cache line reuse based on runtime features such as access frequency and instruction-level behavior. Jain and Lin [2] introduced a Belady-inspired model that leverages historical patterns to improve replacement accuracy by learning from reuse outcomes across applications. Additional research, such as [5]–[7], has shown that predictive approaches can lead to significant improvements in cache hit rates and system throughput when compared to traditional heuristics.

Most of these predictive strategies, however, reduce the cache replacement decision to a binary classification problem—deciding whether a cache line will be reused soon or not. While effective to some extent, this binary view introduces two fundamental limitations. First, minor mispredictions can result in suboptimal evictions, especially when useful lines are mistakenly classified. Second, when many lines fall into the same predicted category, systems often revert to fallback heuristics like LRU to break ties, thereby diluting the predictive benefit.

To address these issues, researchers have turned toward multiclass prediction models, which aim to estimate more granular information—specifically, the reuse distance, or how far in the future a cache line will be accessed again. Keramidas et al. [3] first explored this concept using lightweight reuse distance predictors. Vantrease et al. [4] later extended this idea by introducing the concept of Estimated Time of Arrival (ETA), allowing cache lines to be ordered based on their predicted time of reuse. Similar strategies, such as those proposed in [8], attempt to use learned temporal locality characteristics to make more fine-grained decisions about eviction priority.

Despite the theoretical advantages of ETA-based models, their adoption in practice has been limited. Many previous designs still fall back on age-based heuristics in cases of uncertainty, weakening their adherence to ETA principles. Moreover, prediction inaccuracies in reuse distance estimates can affect overall performance, and studies like [4] have shown only modest gains—around 2–3% IPC improvement over LRU in realistic workloads. Consequently, recent efforts have also investigated hybrid schemes [9], reinforcement learning [10], and program phase-aware strategies [11] as potential alternatives.

In this paper, we present Mockingjay, a novel ETA-based cache replacement policy that is designed to closely approximate Belady's MIN policy. Unlike previous ETA-based approaches that blend predictive and heuristic strategies, Mockingjay remains fully committed to ETA-driven eviction. It predicts the reuse distance of cache lines and always evicts the line with the longest predicted time until reuse. Our approach demonstrates significant improvements in cache efficiency and performance, even outperforming an LRU-managed cache that is twice as large.

Mockingjay is built on three key components:

A Sampled Cache, which maintains a compact, representative history of cache accesses across selected sets. This structure is 8× the size of each sampled set and stores tags, timestamps, and PC signatures for training.

A Reuse Distance Predictor (RDP), implemented as a direct-mapped cache indexed by PC signatures. It is trained using temporal difference learning, ensuring it adapts gradually to access patterns while being resilient to outliers.

An ETA Counter, implemented as an Estimated Time Remaining (ETR) counter for each cache line. The ETR is initialized using the RDP and is decremented on every access to another line. The line with the largest absolute ETR is chosen for eviction. In certain cases, if the new block has a predicted ETA larger than any existing line, it is bypassed entirely—eliminating the need for eviction.

Mockingjay also defers eviction decisions to runtime—ETAs are assigned at insertion but interpreted only when eviction is needed—allowing the policy to operate with more up-to-date information. This delayed comparison model provides greater flexibility and accuracy than conventional schemes that make early, static decisions.

By combining fine-grained reuse prediction, ETA-guided decision making, and efficient implementation, Mockingjay establishes a new benchmark in the field of cache replacement policies. It demonstrates how multiclass prediction, when applied faithfully, can outperform both traditional heuristics and recent state-of-the-art techniques.

II. Problem Statement

Modern computing systems rely heavily on efficient memory hierarchies to deliver high performance, and cache management plays a central role in this effort. However, selecting which data to evict from the cache remains a difficult and unresolved challenge. Traditional replacement policies like Least Recently Used (LRU) and Random Replacement are easy to implement but struggle in real-world scenarios where memory access patterns are complex and constantly changing. These basic strategies fail to adapt to workload behavior, leading to frequent cache misses and underutilized cache capacity.

While recent advances have introduced prediction-based techniques to improve eviction decisions, many of these approaches reduce the problem to a simple yes-or-no classification—predicting whether a cache line will be reused or not. This binary perspective is often too coarse to capture the full complexity of memory access behavior. A single misprediction can lead to poor eviction choices, and when multiple lines fall into the same predicted category, fallback heuristics like LRU are used to break ties. This undermines the benefits of prediction and limits overall performance gains.

The challenge becomes even more pronounced in the last-level cache (LLC), where reuse is infrequent, delayed, and difficult to recognize. The subtlety of reuse behavior at this level demands a replacement policy that goes beyond binary decisions and simple heuristics. What is needed is a more fine-grained, robust solution that can accurately anticipate when a line will be reused and make informed eviction decisions based on that information—particularly in systems where cache contention is high and prediction errors are costly.

III. Related Work

The cache replacement problem has been a major focus of research ever since Belady introduced the MIN policy, which sets a theoretical ideal by always evicting the cache line that will be used farthest in the future. While this perfect foresight isn't possible in real systems, Belady's work has inspired many practical approaches that aim to approximate it [12]. In this section, we explore prior research efforts, grouping them into three categories: (1) memoryless replacement policies, (2) prediction-based methods, and (3) reuse prediction techniques.

A. Memoryless Replacement Policies

Memoryless policies make eviction decisions based solely on the current state of the cache without considering historical behavior. One of the earliest and most widely adopted strategies is Least Recently Used (LRU), which approximates temporal locality by evicting the line that has not been accessed for the longest period. While LRU performs well for workloads with strong temporal locality, it suffers under some conditions where the working set exceeds the cache size [15].

Least Frequently Used (LFU) attempts to reduce some challenges faced by LRU by prioritizing the eviction of blocks that have been accessed the least number of times. However, LFU is slow to adapt to changes in working set behavior and can retain outdated blocks.

A more advanced example of a memoryless approach is the Static Re-Reference Interval Prediction (SRRIP) policy [6], which assigns a fixed reuse priority to each block when it is inserted into the cache. This helps the system better handle thrashing by delaying eviction for blocks likely to be reused soon. However, since SRRIP relies on static thresholds, it may struggle to adapt to changing access patterns.

B. Prediction-Based Policies

Predictive cache replacement policies work by trying to anticipate the future behavior of cache blocks based on patterns observed from past memory accesses. One of the earliest attempts in this direction was dead block prediction [1], where the idea was to classify cache blocks as either likely to be reused ("alive") or unlikely to be reused ("dead").

A major breakthrough came with Hawkeye [2], which took a more direct approach by learning caching decisions that would closely mimic Belady's optimal policy. Hawkeye analyzes a long history of past memory accesses and uses program counter (PC)-based classification to label cache blocks as either cache-friendly or cache-averse based on their simulated behavior under Belady's MIN. While highly effective, Hawkeye's reliance on binary classification makes it vulnerable to prediction errors, which can negatively impact replacement decisions.

Building on these ideas, Glider [15] incorporates deep learning methods, using perceptron-based predictors to refine cache block classification. Although Glider achieves better prediction accuracy than Hawkeye, it introduces additional hardware complexity to support the learning model.

More recently, machine learning has further expanded the predictive capabilities of cache management. For example, Saad and Jones [10] apply reinforcement learning techniques, training cache policies through policy gradients to adapt over time. Similarly, Boroumand et al. [10] use neural networks in Neural Cache to predict future memory access patterns, improving replacement decisions. MLCache, developed by Carlson et al., takes a different path by employing decision trees trained on access features, striking a balance between the simplicity of traditional predictors and the sophistication of deep learning approaches.

While these machine learning-based methods offer impressive adaptability, they often come with significant overhead in terms of storage and computation. This challenge has motivated lighter-weight alternatives like Mockingjay, which uses multiclass reuse prediction to achieve strong performance while maintaining a practical hardware footprint.

C. Reuse Prediction Policies

Another important direction in cache management research focuses on predicting reuse distance — that is, estimating the number of accesses between two uses of the same data block to guide eviction decisions. Instead of simply classifying blocks into binary categories, reuse distance predictors attempt to provide a more detailed estimate of how long a block will remain unused.

Early reuse-based techniques [3][4] protect cache blocks until their predicted reuse distance expires. However, these methods can become inefficient when predictions are inaccurate, potentially leading to unnecessary cache pollution. To address this, Leeway [16] introduces an adaptive strategy that dynamically adjusts protection periods based on observed variability in reuse timing, although it still struggles with occasional mispredictions.

More recent efforts like RDBP [17] build on this idea by combining reuse distance estimates with cache bypassing and placement decisions, improving cache occupancy and efficiency. SHIELD [18] extends reuse distance-based strategies to emerging memory technologies like STT-RAM, balancing the trade-offs between energy efficiency and device endurance. Similarly, RRDP [19] by Bhattacharjee and Venkataramani uses region-based analysis to predict reuse patterns more accurately, capturing spatial locality rather than relying solely on program counters.

The EVA policy [20] offers a different perspective by modeling reuse behavior probabilistically, using age distributions instead of predicting specific reuse distances.

In contrast to these protection-driven approaches, Mockingjay focuses on predicting the relative ordering of reuse times at eviction, allowing it to more closely approximate Belady's MIN policy while avoiding the complexity and overhead of absolute reuse distance prediction.

IV. Our Solution

The solution discussed in this paper is known as Mockingjay, and it replicates Belady's MIN Policy. Belady's MIN policy is a clairvoyant ideal cache replacement policy where lines that are used furthest in the future are those evicted from the cache. As mentioned previously, there are 3 main features that work together to achieve this:

A. Sampled Cache

The sampled cache plays a crucial role in modern cache replacement policies by maintaining a compact but long-term history of memory accesses. Instead of storing full cache lines, the sampled cache records only metadata, specifically the tag, timestamp, and program counter (PC) signature of each access. It is organized into 32 sampled sets, with each set maintaining the history of past accesses to a subset of cache blocks.

For a 16-way main cache, each sampled set tracks the last 128 accesses. To implement this efficiently, the sampled cache is designed as a 5-way set-associative structure with 512 sets. Conceptually, the 512 sets can be viewed as a collection of 32 smaller sub-caches, where each sub-cache consists of 16 sets and maintains the detailed access history for one sampled main cache set. This organization allows the sampled cache to monitor reuse behavior across a wide range of accesses without significant hardware overhead.

When inserting a new access into the sampled cache, entries are managed using a Least Recently Used (LRU) replacement policy within each set. If a sampled set is full, the oldest entry based on the recorded timestamp is evicted to make room for the new one. This ensures that the sampled cache always reflects the most recent access patterns while gradually aging out outdated entries.

To handle timestamp wraparound, the system adds a constant value, $1 \ll$ TIMESTAMP_BITS, to the current timestamp before calculating differences. Although wraparound can cause the current timestamp to appear smaller than the previous one, such cases are rare in practice because lines are evicted once they are observed to be more than 128 sampled set accesses old. By limiting the maximum allowed reuse window, the sampled cache prevents incorrect aging calculations due to timestamp overflow, ensuring reliable tracking of reuse behavior.

Through this lightweight but carefully structured design, the sampled cache enables accurate reuse modeling and prediction training for advanced cache management systems like Hawkeye and Mockingjay, bridging the gap between practical replacement strategies and Belady's ideal behavior.

B. Reuse Distance Predictor (RDP)

The Reuse Distance Predictor (RDP) is implemented as a directly mapped array, where program counter (PC) signatures serve as indices. Each entry in the RDP stores a predicted reuse distance along with the block number associated with the corresponding PC signature. This lightweight design enables fast lookup and update operations, making it suitable for real-time cache management.

During execution, whenever a cache block is accessed, its PC signature is used to query the RDP. If an entry exists, the predicted reuse distance provides an estimate of how far into the future the block is expected to be reused. This prediction guides the replacement policy by influencing eviction priorities: blocks with longer predicted reuse distances are more likely to be evicted earlier.

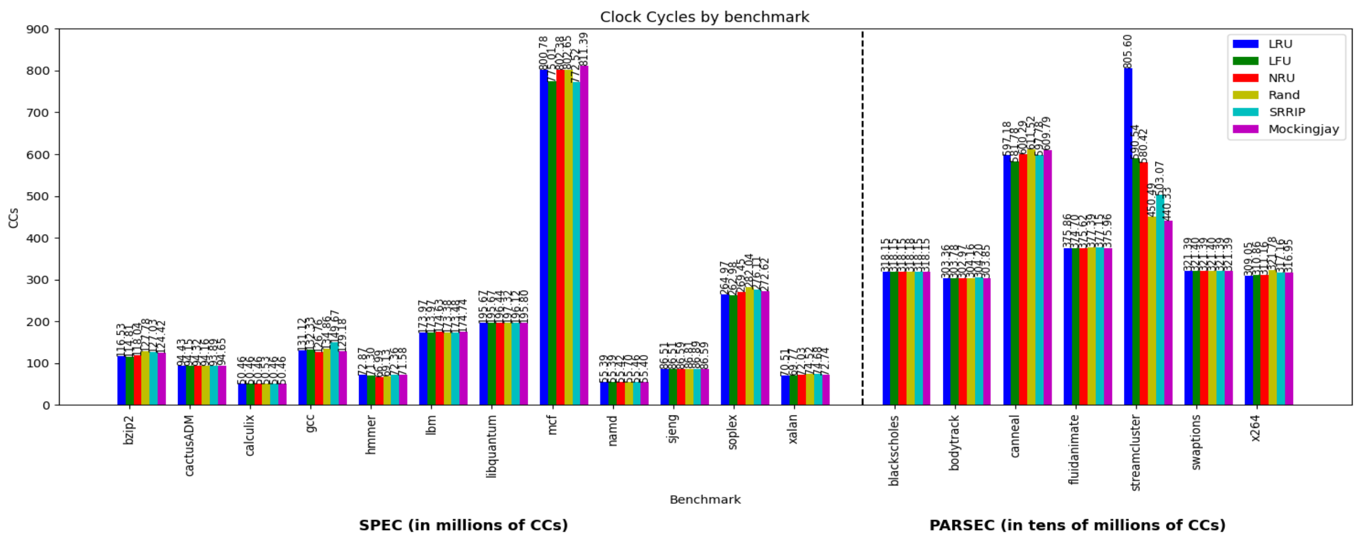


Fig. 1. Clock Cycles by benchmark.

The RDP is trained dynamically based on observed reuse behavior. When a sampled cache line is evicted, the actual reuse distance (measured as the number of accesses between consecutive uses) is computed and used to update the RDP entry corresponding to the line's PC signature. If multiple PC signatures map to the same RDP entry (i.e., due to hash collisions), the new reuse information overwrites the previous prediction, allowing the predictor to adapt to changing program behavior over time.

By focusing on PC signatures, the RDP captures instruction-specific reuse patterns, enabling more fine-grained and accurate prediction than simpler cache history-based heuristics. Its compact size and direct-mapped structure ensure minimal hardware overhead, while its predictive accuracy significantly improves cache replacement decisions, helping policies like Mockingjay to better approximate Belady's optimal behavior.

C. *ETR Counters*

The ETR (Expected Time of Reuse) counters are used to track when each cache line is expected to be accessed again, based on predicted reuse distances. When a new block enters the cache, its ETR value is set using the output from the Reuse Distance Predictor (RDP), which estimates how far into the future the block will be reused [1]. Instead of updating these counters on every access, the policy updates them every eight accesses to a given set, making it more efficient while still capturing temporal behavior.

Over time, these counters are decremented to reflect that reuse is getting closer—however, they can go below zero. That's why eviction decisions are based on the absolute value of the ETR counter. The block with the largest absolute ETR value is considered least likely to be reused soon and is chosen for eviction [2].

To distinguish between typical working set behavior and streaming or scan-like patterns, a threshold is introduced. If a block's ETR exceeds a set maximum value (104 in this implementation), it's considered part of a scan and excluded from further ETR updates. This avoids punishing blocks with long but predictable reuse distances, allowing the policy to preserve the core working set while gracefully handling scans [3].

This strategy allows the policy to approximate the behavior of Belady's MIN algorithm more closely than traditional methods like LRU or SRRIP. By combining predicted reuse distances with a thoughtful update strategy and classification threshold, the policy achieves better eviction decisions in practical systems [1], [4].

V. Evaluation

A. *Methodology*

Simulator: Our implementation of Mockingjay was done in the Zsim simulator. It was originally written by Daniel Sanchez at Stanford University. It supports both single core and multi core systems and provides an intuitive framework with ample abstraction to implement new polices.

Benchmarks: We simulated our implementation on a multitude of CPU2006 benchmarks. For single core uses, we use 7 SPEC benchmarks with integer operations and 5 SPEC benchmarks for floating point operations. It is to be noted that there were originally 7 floating point benchmarks, but both milc and leslie3d were unable to be simulated due to configuration issues that the teaching assistant told us to ignore. As for multicore simulation, we utilize 8 PARSEC benchmarks.

Other Policies: We are comparing performance to many other existing replacement policies. Zsim provides native implementations for LRU, LFU, NRU, and Random. Thus, we ran simulations on all of these policies under the same set of benchmarks. Further, we implemented SRRIP in homework

4, so we ran simulations on that policy as well. SRRIP utilizes one of the crucial components in Mockingjay, multiclass prediction, so comparing to this will reveal how much Mockingjay benefits from the long history and predictor. LRU is also an important policy to benchmark and compare because the original paper proposing Mockingjay made many comparisons to LRU and claimed about 5.7% improvement over it.

Processor Configuration: Per SPEC benchmark, we are using a single core OOO westmere processor with three levels of cache, where the last level cache (LLC) is sized at 2MB with 8 banks, and set associative with 16 ways. The PARSEC benchmarks use a very similar set up but the LLC is 8MB. We are only testing the performance of Mockingjay in the LLC.

B. Metrics

The key metrics chosen to measure the success of each replacement policy were instructions per cycle (IPC), misses per kilo-instruction (MPKI), and clock cycles. The higher the IPC, and the lower the MPKI and CCs, the better a replacement policy performed. Out of these metrics, MPKI is the most obvious signifier of success as the only thing we're changing in our implementation is the replacement policy, and MPKI is directly related to the number of misses. Additionally, due to the fact that all replacement policies ran the same simulations with the same instruction counts, the cache accesses are the same, so MPKI is a very accurate representation of the miss rate of a cache. Similarly, due to there being no variability in anything but the replacement policy, IPC is directly related to cache accuracy. Clock cycles can are inversely related to cache accuracy, and provide more insight onto the main issues of certain benchmarks. For example, when most replacement policies have similar IPC and MPKI values, but the number of clock cycles is very high, it can be deduced that that benchmark is very compute-heavy.

Additionally, we took the geometric mean of the IPC and MPKI of benchmark results by replacement policy, and compared SPEC, PARSEC, and combined results to see what class of benchmarks each replacement policy excelled or struggled with. We did not do this for clock cycles as the numbers were simply too large and would have been difficult to track accurately. Additionally, the PARSEC benchmarks have many more clock cycles and would have skewed the combined data if a replacement policy performed poorly on even a single benchmark.

VI. Results

The following graphs display the results of simulations for the Mockingjay replacement policy compared to results from simulations for SRRIP, LRU, LFU, NRU, and Random replacement policies. On most benchmarks, Mockingjay performed roughly the same as other replacement policies, excluding mcf, canneal, soplex and streamcluster. A possible reason why Mockingjay under performs on mcf, canneal, and soplex is that they are all pointer-chase heavy or have irregular access patterns. Pointer-chase heavy memory access patterns mean there are many heap-allocated memory accesses, which reduces spatial locality and temporal locality (as memory may be freed in a hard-to-predict way). Irregular access patterns mean that reuse distances will not be very accurate as blocks that are predicted to be reused in the distant future may be reused much before their predicted ETA.

MCF is a network optimization problem that uses lots of linked lists and graphs, which are dynamically allocated and require lots of pointer chasing, which is why it is bad for all replacement policies, but since Mockingjay tries to predict reuse intervals with higher accuracy than the other replacement policies, it runs into the issue of mispredicting by a larger margin than other policies. Canneal is a simulation of annealing which is a physics-based optimization process that has sporadic data reuse and pointer chasing, so reuse-prediction will underperform compared to something like LFU that has a lower capability for holding onto dead blocks. Soplex deals with large sparse matrices, which is already not great for caching, and the reuse time for elements is usually very fast and the blocks are dead after very few accesses. This means that Mockingjay keeps recently accessed dead blocks around longer than they should be kept believing that they will be reused again shortly.

On the other hand, Mockingjay excelled at streamcluster. Streamcluster is a simulation of organizing streaming data in real time, where it takes in data as a stream then operates on that data afterward in phases. This means that the memory accesses are very predictable, and Mockingjay's aggressive aging works very well, as it predicts reuse within a phase, then can evict previous phases without too many extra misses. This is also why SRRIP is better than LRU and LFU, as it can estimate reuse distances well enough to protect the working set of this benchmark without keeping dead blocks for too long.

From our granular simulation results (graphs), it seems Mockingjay excels in some categories, but is roughly average across all benchmarks. It does not seem likely that the benefits of Mockingjay outweigh the hardware overhead required to implement it.

TABLE I
SPEC Benchmark Performance

Policy	IPC	IPC % better	MPKI	MPKI % better
LRU	0.8079	0.00%	3.2533	0.00%
LFU	0.8134	0.68%	3.1964	1.75%
NRU	0.8117	0.47%	3.1766	2.36%
Rand	0.7945	-1.65%	3.4381	-5.68%
SRRIP	0.7897	-2.26%	3.6083	-10.91%
Mockingjay	0.8003	-0.95%	3.3288	-2.32%

Regarding the mean results displayed above and below, it seems Mockingjay outperforms all other policies on average. This is most likely due to how well it performed on streamcluster, and the fact that it did not perform significantly worse than other replacement policies on the other benchmarks. However, the results most likely do not outweigh the cost and overhead time for the extra hardware required.

It is notable that the total mean IPC calculated with the

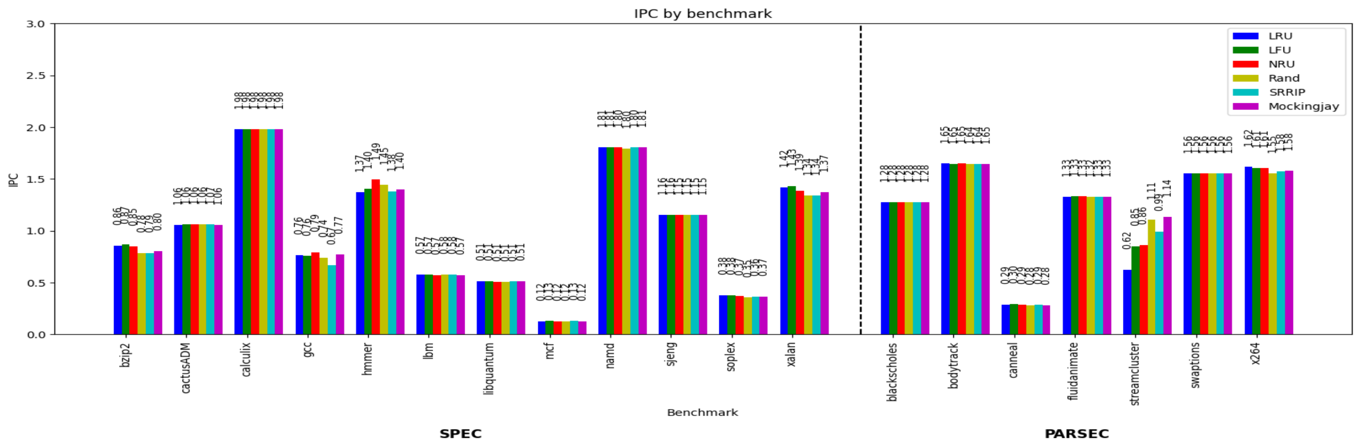


Fig. 2. Instructions per clock cycle by benchmark.

TABLE II
PARSEC BENCHMARK PERFORMANCE

Policy	IPC	IPC % better	MPKI	MPKI % better
LRU	1.0195	0.00%	0.4426	0.00%
LFU	1.0482	2.81%	0.4059	8.30%
NRU	1.0469	2.68%	0.4241	4.19%
Rand	1.0638	4.34%	0.4013	9.33%
SRRIP	1.0574	3.71%	0.4097	7.44%
Mockingjay	1.0679	4.74%	0.3888	12.16%

TABLE III
COMBINED SPEC + PARSEC PERFORMANCE

Policy	IPC	IPC % better	MPKI	MPKI % better
LRU	0.8237	0.00%	1.4399	0.00%
LFU	0.8526	3.51%	1.2973	9.90%
NRU	0.8498	3.17%	1.3471	6.44%
Rand	0.8453	2.62%	1.3798	4.18%
SRRIP	0.8350	1.37%	1.4782	-2.66%
Mockingjay	0.8546	3.75%	1.2942	10.12%

results of our simulations is only $\tilde{2}$% off of what the reference paper claimed it to be. Due to the fact that we ran different simulations, and were adapting this replacement policy to a different simulator, our results are better than anticipated.

VII. CONCLUSIONS

This project implemented Mockingjay, a modern day cache replacement policy, and explored its limitations and accomplishments in comparison to various commonly used policies, namely LRU, LFU, NRU, Random, and SRRIP.

Based on the literature review conducted for this project, it is clear that modern cache replacement research is deeply focused on mimicking the behavior of Belady's MIN policy (also known as the clairvoyant algorithm because it requires knowledge of the future to be optimal). In general, this has proven to be difficult because all algorithms must accomplish this goal through predictive measures (making decisions based on a history of prior evictions). The goal is to improve performance while keeping hardware overhead as small as possible.

Mockingjay attempts to mimic MIN by keeping an immensely long history of past accesses for a small sample of sets in the Last-Level-Cache (LLC). Coupled with this, it uses multiclass prediction similar to SRRIP so that there is more hysteresis in the decision making for eviction. Further, in order to reduce the effect of prediction errors, Mockingjay ranks candidates based on the relative ordering of predicted reuse distances instead of the distances themselves. Lastly, the policy defers this prioritization until the time of eviction so that more information is available.

As a result of all of these components, this policy costs 32KB of extra hardware. This is an immense overhead, especially compared to legacy algorithms like LRU or NRU which just require a few bits. Though the paper claims to mimic MIN almost exactly, being just 0.3% shy of its performance (being 5.7% better than LRU), our attempts to replicate the policy only yielded at most 3.75% improvement over LRU. We believe the hardware overhead necessary for this policy (the size of an L1 cache) outweighs the performance improvement that we observed from our simulations.

Our results are less impressive than what is claimed in the paper. This may be attributed to our choice of simulator and the way that PC tracking differs. Regardless, Mockingjay still needs the least amount of hardware for the class of improvements that it makes. Other policies that reach this level of improvement involve Machine Learning and Reinforcement Learning which require much more overhead than this policy.

REFERENCES

[1] M. M. Khan, S. M. Khan, and A. Moshovos, "Machine Learning-Based Cache Management for Last-Level Caches," *IEEE Trans. Comput.*, vol. 69, no. 4, pp. 560–573, Apr. 2020.
[2] 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), Seoul, Korea, Republic of*, 2022, pp. 558-572
[3] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Cache Replacement Based on Reuse Distance Prediction," in *Proc. Int. Conf. Comput. Design (ICCD)*, 2007, pp. 245–250.

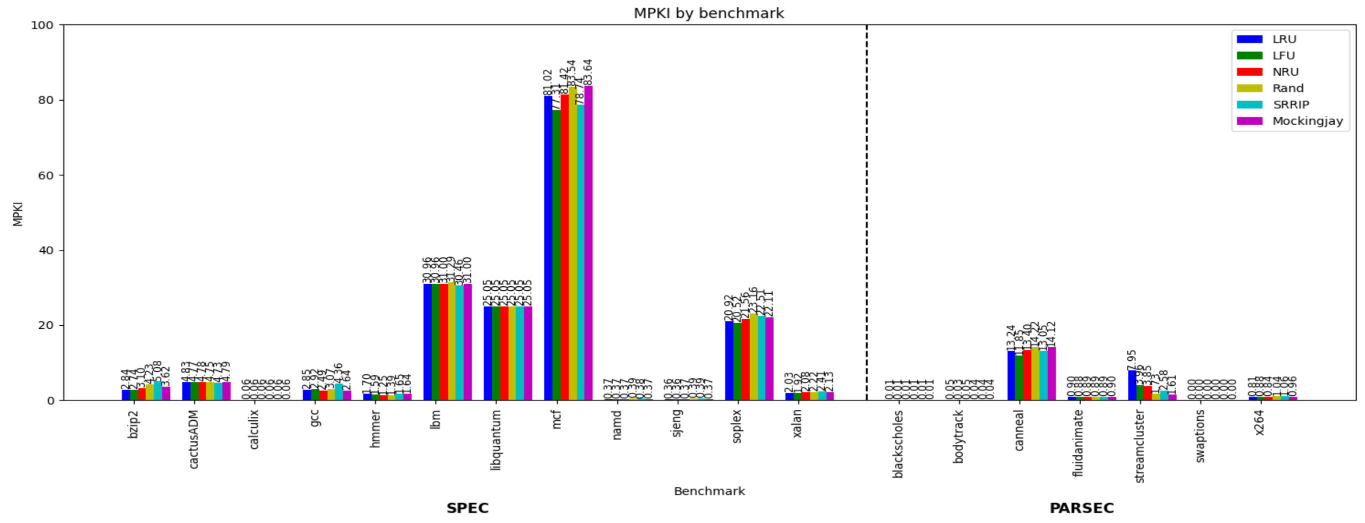


Fig. 3. Misses per kilo-instruction by benchmark.

[4] D. Vantrease, M. M. Swift, and D. A. Wood, "Revive: Reuse Prediction for Variable-Retention Caches," in *Proc. 48th IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2015, pp. 260–271.

[5] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance," *ACM SIGMETRICS*, 2002.

[6] A. Jaleel et al., "High Performance Cache Replacement Using Re-reference Interval Prediction," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2010, pp. 60–71.

[7] H. H. Najaf-abadi et al., "Adaptive Cache Management for Achieving Performance Predictability," in *Proc. Int. Conf. Parallel Architecture and Compilation Techniques (PACT)*, 2017.

[8] M. Ferdman et al., "Clearing the Clouds: A Study of Emerging Scale-Out Workloads on Modern Hardware," in *Proc. 17th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[9] Z. Yu and A. Srivastava, "A Study on Hybrid Cache Replacement Policies," in *Proc. Int. Conf. Supercomputing (ICS)*, 2013.

[10] A. Agrawal et al., "Learning to Evict: A Deep Reinforcement Learning Framework for Cache Replacement," *arXiv preprint arXiv:2006.16239*, 2020.

[11] K. Woo and D. M. Tullsen, "Temporal Memory Streaming for Multimedia Applications," *IEEE Comput. Archit. Lett.*, vol. 3, no. 1, pp. 19–22, Jan. 2004.

[12] L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966.

[13] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer, "High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)," in *Proc. 37th Int. Symp. Comput. Archit. (ISCA)*, 2010, pp. 60–71.

[14] A. Jain and C. Lin, "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement," in *Proc. 43rd Int. Symp. Comput. Archit. (ISCA)*, 2016, pp. 78–89.

[15] J. Daniel and B. Mutlu, "Glider: A Machine Learning-Driven Cache Replacement Policy," in *Proc. 53rd IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2020, pp. 924–936.

[16] M. Ferdman et al., "Leeway: Addressing Variability in Cache Reuse," in *Proc. Int. Symp. High-Performance Computer Architecture (HPCA)*, 2017, pp. 128–139.

[17] X. Zhou, T. Liu, and S. Ren, "RDBP: Reuse Distance-Based Bypassing and Placement for Last-Level Cache," in *Proc. Design, Automation and Test in Europe (DATE)*, 2019, pp. 1397–1402.

[18] S. Mittal, "SHIELD: A Reuse Distance Based Policy for STT-RAM Based Last-Level Caches," in *Proc. Int. Symp. Quality Electronic Design (ISQED)*, 2016, pp. 428–434.

[19] R. Bhattacharjee and G. Venkataramani, "Accurate Reuse Distance Prediction for Effective Cache Management," in *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2014, pp. 1–10.

[20] B. Beckmann and D. A. Wood, "EVA: Efficient Victim Cache Management via Global Replacement," in *Proc. 44th IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2011, pp. 143–154.

[21] A. Jain and C. Lin, "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement," in *Proc. 43rd Int. Symp. Comput. Archit. (ISCA)*, 2016, pp. 78–89.

[22] M. M. Khan, S. M. Khan, and A. Moshovos, "Machine Learning-Based Cache Management for Last-Level Caches," *IEEE Trans. Comput.*, vol. 69, no. 4, pp. 560–573, Apr. 2020.

[23] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Cache Replacement Based on Reuse Distance Prediction," in *Proc. Int. Conf. Comput. Design (ICCD)*, 2007, pp. 245–250.

[24] D. Vantrease, M. M. Swift, and D. A. Wood, "Revive: Reuse Prediction for Variable-Retention Caches," in *Proc. 48th Int. Sym*

Work Division

Mathimalar Mathivanan contributed to multiple phases of the project, starting with drafting the initial version of the proposal report. She conducted a thorough reading and understanding of the research paper, enabling her to lead the initial cache implementation. She developed the core cache logic, created a sample testbench to validate the design, and iteratively refined the implementation by integrating timestamp tracking and the LRU replacement policy. Additionally, she contributed significantly to the final report by writing the Abstract, Introduction, Problem Statement, and Related Work sections.

Ben Armstrong began work on this project by thoroughly researching the topic and implementation of the replacement policy, including a read-through of the research paper, ChampSim implementation, and ZSim source code. He started the first set of large scale changes in the GitHub by providing support for PC accesses in the replacement policy files. He then wrote the code for simulation of the ETR counters, handled simulation logistics and extensive verification and debugging of the entire project, and generated plots and analyzed gathered data. He added instructions to the README for how to gather data and generate plots, and he contributed to the report by writing the Our Solution section with help from Mathimalar Mathivanan, the Results section, and the Evaluation section with help from Lohitaksh Allampalli.

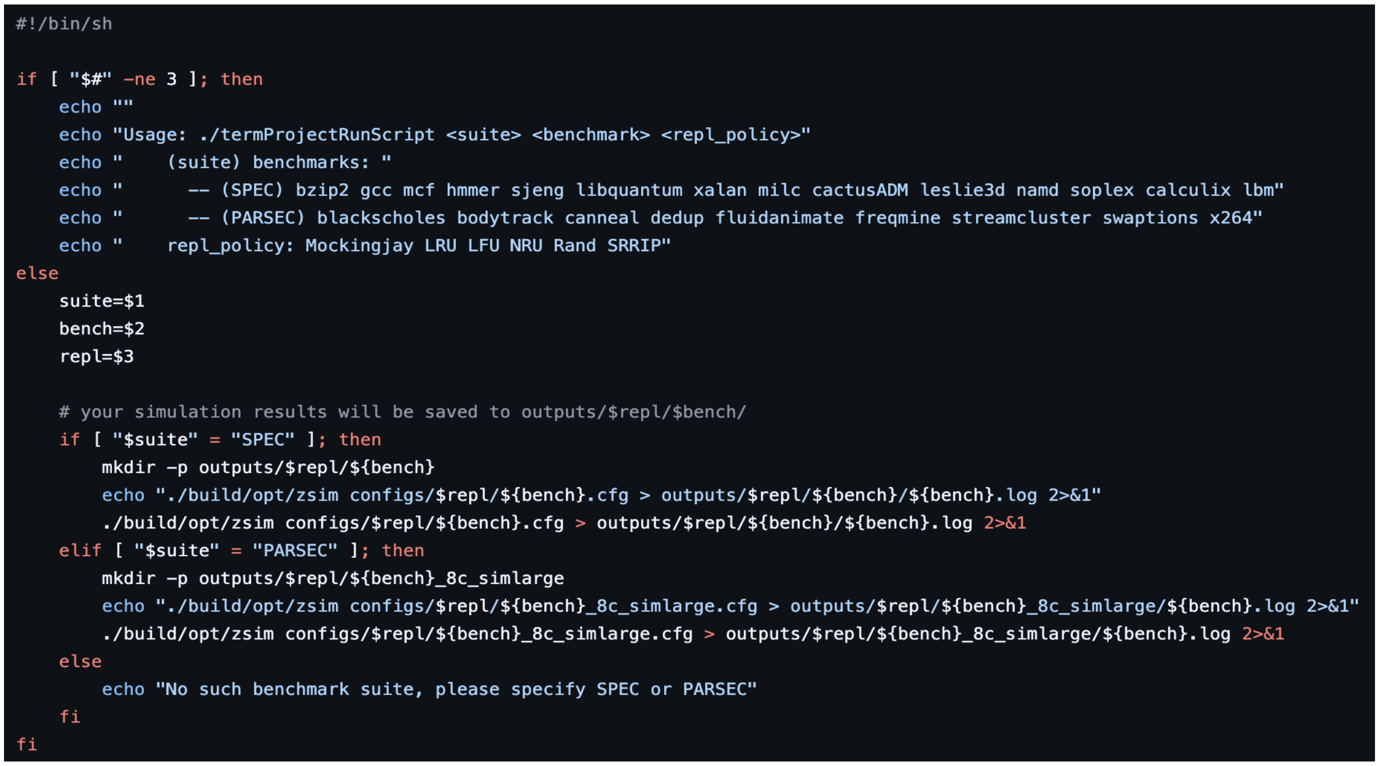
Lohitaksh Allampalli began this project by fully understanding both the conceptual purpose and implementation details in the original paper. After this, he began programming much of the needed helper functions. He completed the implementation of the sampled cache and made the reuse distance predictor (RDP). After this, he worked with Ben Armstrong to initialize the data structures and debug our implementation. He suggested using an unordered map for the sampled cache instead of a 2D array which resolved the segmentation fault in our code. Further, he made the configuration files to set up the simulation for Mockingjay. In the report, he wrote the Conclusion section and the Methodology in the Evaluation section. He also wrote the Project Description, Structure, and Instructions sections in the README for the artifacts.

APPENDIX

The artifacts for this project (simulator, code, configurations, run scripts, results, etc) are all provided in our GitHub which is part of the CSCE 614 classroom organization. The following is the link to the GitHub: https://github.com/CSCE-469-614-EJKIM-S25/group10

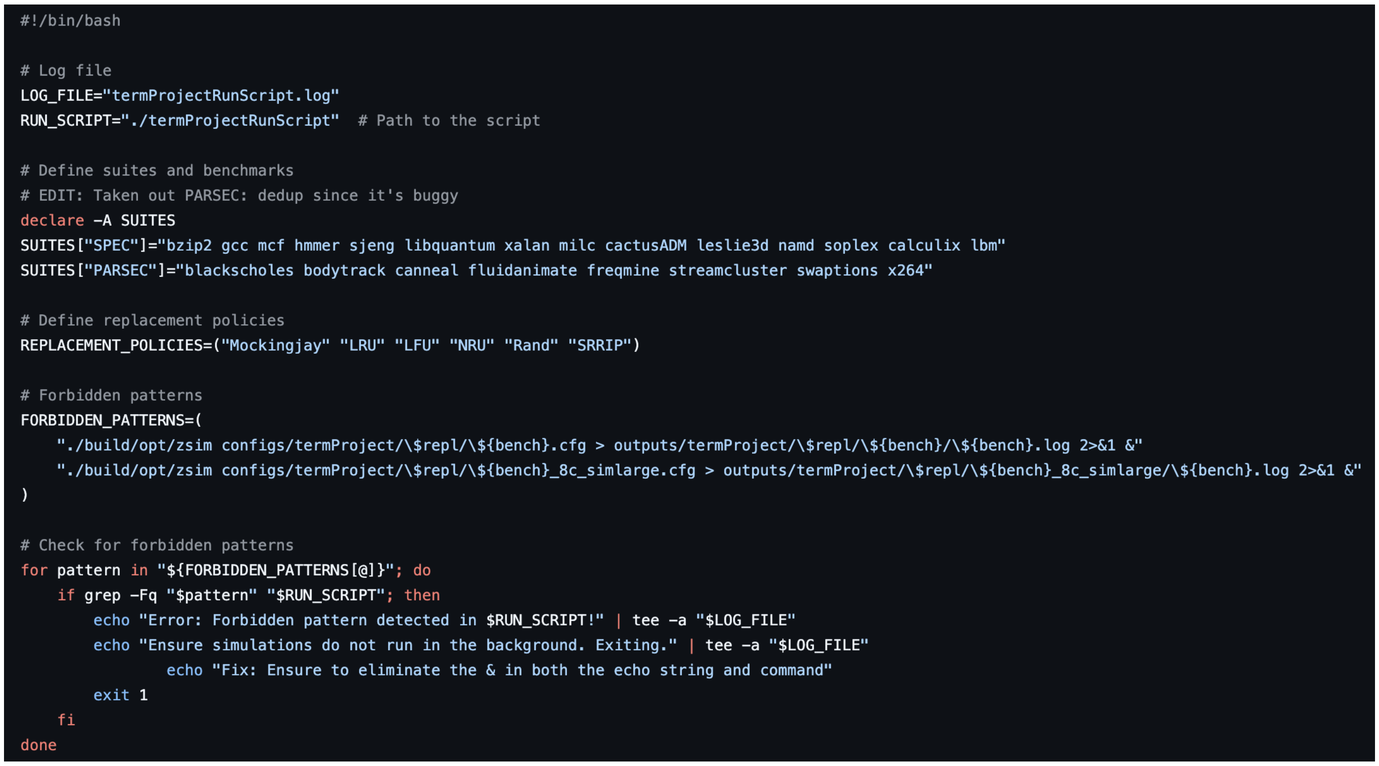
We used the same run script code (Fig. 4) as what was provided in Homework 2 and 4, but we removed the & because we did not want these commands to run in the background.

The reason for removing the & is to enable simulations to run without supervision. We created a bash script (Fig. 5 and Fig. 6) that executes the run script every time one simulation completes. This enabled us to execute the script and leave it unsupervised overnight, with the results waiting for us the next day.



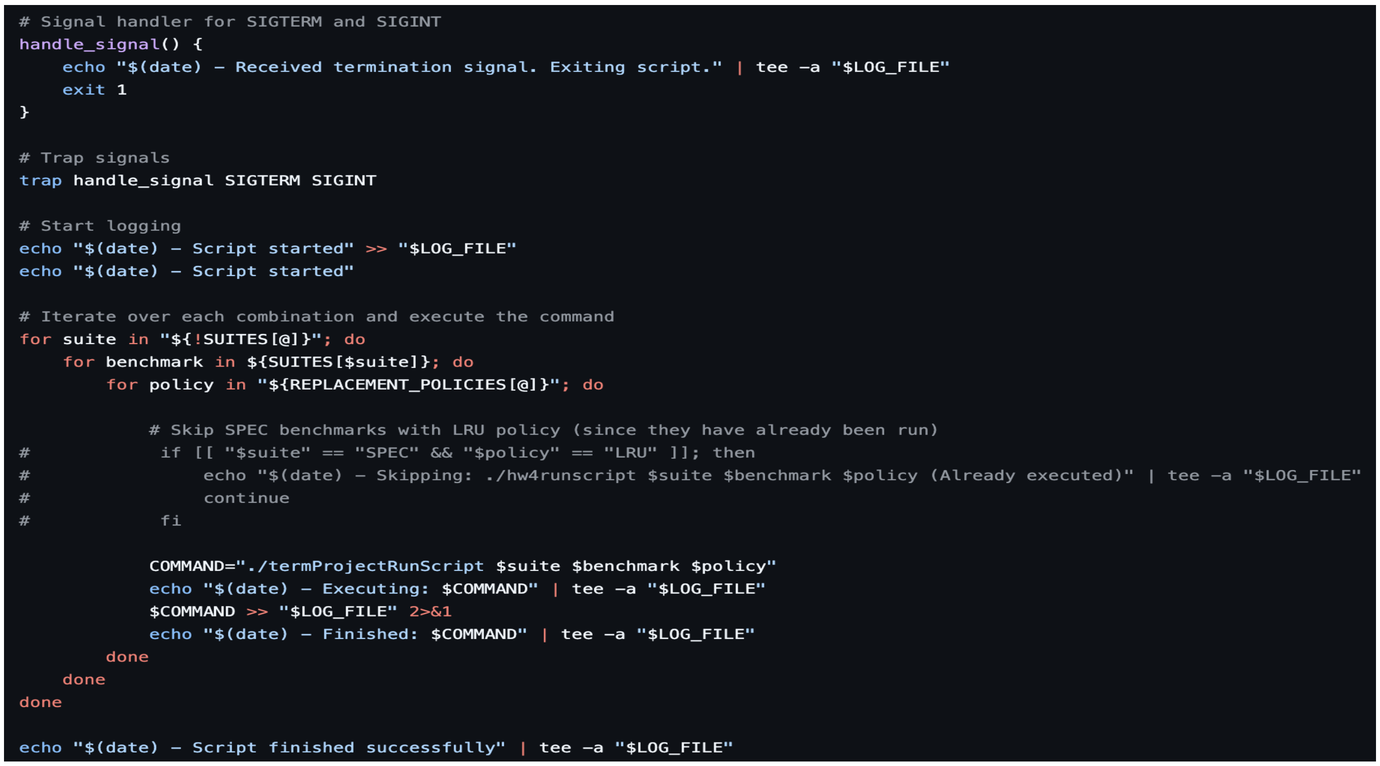
```sh
#!/bin/sh

if [ "$#" -ne 3 ]; then
    echo ""
    echo "Usage: ./termProjectRunScript <suite> <benchmark> <repl_policy>"
    echo "    (suite) benchmarks: "
    echo "      -- (SPEC) bzip2 gcc mcf hmmer sjeng libquantum xalan milc cactusADM leslie3d namd soplex calculix lbm"
    echo "      -- (PARSEC) blackscholes bodytrack canneal dedup fluidanimate freqmine streamcluster swaptions x264"
    echo "    repl_policy: Mockingjay LRU LFU NRU Rand SRRIP"
else
    suite=$1
    bench=$2
    repl=$3

    # your simulation results will be saved to outputs/$repl/$bench/
    if [ "$suite" = "SPEC" ]; then
        mkdir -p outputs/$repl/${bench}
        echo "./build/opt/zsim configs/$repl/${bench}.cfg > outputs/$repl/${bench}/${bench}.log 2>&1"
        ./build/opt/zsim configs/$repl/${bench}.cfg > outputs/$repl/${bench}/${bench}.log 2>&1
    elif [ "$suite" = "PARSEC" ]; then
        mkdir -p outputs/$repl/${bench}_8c_simlarge
        echo "./build/opt/zsim configs/$repl/${bench}_8c_simlarge.cfg > outputs/$repl/${bench}_8c_simlarge/${bench}.log 2>&1"
        ./build/opt/zsim configs/$repl/${bench}_8c_simlarge.cfg > outputs/$repl/${bench}_8c_simlarge/${bench}.log 2>&1
    else
        echo "No such benchmark suite, please specify SPEC or PARSEC"
    fi
fi
```

Fig. 4. Source code of the run script used to execute the command given policy, suite, and benchmark.



```bash
#!/bin/bash

# Log file
LOG_FILE="termProjectRunScript.log"
RUN_SCRIPT="./termProjectRunScript"  # Path to the script

# Define suites and benchmarks
# EDIT: Taken out PARSEC: dedup since it's buggy
declare -A SUITES
SUITES["SPEC"]="bzip2 gcc mcf hmmer sjeng libquantum xalan milc cactusADM leslie3d namd soplex calculix lbm"
SUITES["PARSEC"]="blackscholes bodytrack canneal fluidanimate freqmine streamcluster swaptions x264"

# Define replacement policies
REPLACEMENT_POLICIES=("Mockingjay" "LRU" "LFU" "NRU" "Rand" "SRRIP")

# Forbidden patterns
FORBIDDEN_PATTERNS=(
    "./build/opt/zsim configs/termProject/\$repl/\${bench}.cfg > outputs/termProject/\$repl/\${bench}/\${bench}.log 2>&1 &"
    "./build/opt/zsim configs/termProject/\$repl/\${bench}_8c_simlarge.cfg > outputs/termProject/\$repl/\${bench}_8c_simlarge/\${bench}.log 2>&1 &"
)

# Check for forbidden patterns
for pattern in "${FORBIDDEN_PATTERNS[@]}"; do
    if grep -Fq "$pattern" "$RUN_SCRIPT"; then
        echo "Error: Forbidden pattern detected in $RUN_SCRIPT!" | tee -a "$LOG_FILE"
        echo "Ensure simulations do not run in the background. Exiting." | tee -a "$LOG_FILE"
                echo "Fix: Ensure to eliminate the & in both the echo string and command"
        exit 1
    fi
done
```

Fig. 5. Source code of the bash script used to automatically iterate over every benchmark and policy using the run script



```bash
# Signal handler for SIGTERM and SIGINT
handle_signal() {
    echo "$(date) - Received termination signal. Exiting script." | tee -a "$LOG_FILE"
    exit 1
}

# Trap signals
trap handle_signal SIGTERM SIGINT

# Start logging
echo "$(date) - Script started" >> "$LOG_FILE"
echo "$(date) - Script started"

# Iterate over each combination and execute the command
for suite in "${!SUITES[@]}"; do
    for benchmark in ${SUITES[$suite]}; do
        for policy in "${REPLACEMENT_POLICIES[@]}"; do

            # Skip SPEC benchmarks with LRU policy (since they have already been run)
#            if [[ "$suite" == "SPEC" && "$policy" == "LRU" ]]; then
#                echo "$(date) - Skipping: ./hw4runscript $suite $benchmark $policy (Already executed)" | tee -a "$LOG_FILE"
#                continue
#            fi

            COMMAND="./termProjectRunScript $suite $benchmark $policy"
            echo "$(date) - Executing: $COMMAND" | tee -a "$LOG_FILE"
            $COMMAND >> "$LOG_FILE" 2>&1
            echo "$(date) - Finished: $COMMAND" | tee -a "$LOG_FILE"
        done
    done
done

echo "$(date) - Script finished successfully" | tee -a "$LOG_FILE"
```

Fig. 6. (Cont.) Source code of the bash script used to automatically iterate over every benchmark and policy using the run script