ECE565 - Fall 24 - Final Project - Shepherd Cache

Sudeepa Dongala, Sourag Cherupattamoolayil, Vishal Kevat

Group 26

Abstract

Reducing cache misses is an important aspect of improving the performance of a system. Since locality gets filtered through the L1 cache, L2 incurs significant cache misses when using the base Least Recently Used (LRU) replacement. The optimal replacement policy (OPT) uses future knowledge about the access pattern to decide which line to replace. But, it is not practical to implement this since it requires knowledge about the future. Shepherd Cache tries to bridge this gap by emulating OPT without future knowledge. In Shepherd Cache, a small part of the cache called the Shepherd Cache (SC) decides whether to replace itself or a line from the Main Cache (MC). We implemented a slightly modified version of the SC, evaluated two configurations for SC/MC split with three different cache sizes (a total of six configurations), and compared them against LRU and FIFO which showed that it performs better than FIFO but is only comparable to LRU.

1 Introduction

Modern computing systems rely heavily on multi-level cache hierarchies to mitigate the performance gap between fast processors and slower memory subsystems. The design of cache replacement policies is pivotal in determining the overall efficiency of these systems. While the Least Recently Used (LRU) policy remains a standard choice for L1 caches due to its simplicity and efficacy in leveraging temporal locality, its effectiveness diminishes in L2 caches where temporal locality is less pronounced. This discrepancy results in a significant performance gap when compared to the theoretically optimal replacement policy (OPT), which requires knowledge of future memory accesses to make ideal decisions.

To bridge this gap, researchers have proposed various strategies that approximate OPT's behavior without explicit future knowledge. Among these, the Shepherd Cache (SC) offers a novel architecture that emulates OPT through a dual-layer structure comprising a Main Cache (MC) and an auxiliary Shepherd Cache [1]. The SC gathers insights about future access patterns while hosting cache lines temporarily, enabling the MC to make replacement decisions that closely approximate OPT. This architecture demonstrates significant improvements in cache performance and overall system efficiency.

This project delves into the implementation of the Shepherd Cache, exploring its design principles, performance benefits, and practical feasibility. We replicate the proposed methodology in a simulation environment, compare its performance against conventional strategies like LRU, and analyze its potential to close the gap between LRU and OPT. The findings highlight that Shepherd Cache performs better than FIFO but its performance is almost the same as that of LRU.

2 Related work

Cache replacement policies have been a major focus of research, especially when it comes to addressing conflict misses and improving the efficiency of L2 caches. This section takes a closer look at some key strategies that have shaped the development of proactive and hybrid approaches to cache management.

2.1 Reactive Cache Management Strategies

Conflict misses, caused by multiple memory blocks competing for the same cache location, remain a key challenge in cache management. One notable solution is the Victim Cache, introduced by Jouppi [2], which mitigates conflict misses by maintaining a small fully associative cache near the L1 cache. By storing recently evicted blocks temporarily, the Victim Cache offers a second chance for evicted lines, reducing latency and improving hit rates. However, this approach is reactive in nature and does not proactively guide replacement decisions during eviction, a limitation addressed by Shepherd Cache (SC).

2.2 Hybrid and Adaptive Policies

To overcome the static nature of traditional policies like LRU and LFU, the Adaptive Replacement Policy (ARP) dynamically integrates multiple strategies based on workload behavior [3]. By combining recency and frequency awareness, ARP effectively balances hit rates across diverse workloads. Similarly, the Dynamic Insertion Policy (DIP) addresses the inefficiencies of LRU in L2 caches by dynamically adjusting the insertion position of cache lines in the LRU stack [4]. Lines with poor temporal locality are placed deeper in the stack to ensure faster eviction, while frequently reused lines are prioritized. This adaptive approach effectively conserves valuable cache space and reduces misses by tailoring replacement decisions to workload characteristics. Despite their adaptability, both ARP and DIP are reactive policies that do not leverage proactive mechanisms to approximate optimal replacement decisions.

2.3 Proactive and Optimal Approximation Strategies

The Optimal Replacement Policy (OPT) represents the theoretical ideal for cache management, replacing the cache line that will not be used for the longest period. However, its reliance on future knowledge renders it impractical for real-world systems. Shepherd Cache (SC) addresses this limitation by approximating OPT through a dual-layer structure comprising a Main Cache (MC) and an auxiliary Shepherd Cache [1]. The SC gathers insights about future access patterns while hosting cache lines temporarily, enabling the MC to make informed replacement decisions. This proactive approach sets it apart from traditional and adaptive policies.

2.4 Contributions of Shepherd Cache

Unlike prior work, the Shepherd Cache proactively uses metadata and rank-based heuristics to approximate optimal replacement decisions. By balancing SC and MC allocations and leveraging a simple FIFO mechanism for SC blocks, it has been shown in literature to reduce cache misses and outperform traditional strategies like LRU and FIFO in various scenarios. This project builds on these principles, introducing a modified SC design and evaluating its performance under multiple configurations to further explore its practical benefits.

3 Implementation

In this section, we discuss our implementation of the Shepherd Cache and how it is different from the original implementation.

The replacement policies in GEM5 are designed with four key functionalities, each serving a specific role in managing the behavior of cache lines. The **invalidate** function is used to mark a cache line as invalid, ensuring that outdated or unnecessary data is no longer considered valid for operations. The **touch** function is triggered whenever a cache line is accessed successfully, allowing the replacement policy to update metadata or statistics, such as access frequency or recency, to reflect this event. The **reset** function is responsible for initializing or re-initializing parameters associated with a cache line, typically when a new line is inserted into the cache. This ensures that the newly inserted line starts with appropriate values for the replacement algorithm to function correctly. Finally, the **getVictim** function is the core of the replacement policy, determining which cache line should be evicted when space is required for a new line. This function evaluates all potential candidates within the same cache index and selects a victim based on the specific logic of the implemented replacement policy, such as least-recently-used (LRU), least-frequently-used (LFU), or other custom heuristics.

At the start of the system's operation, the initialization function is invoked to prepare each cache index for subsequent operations. This setup phase ensures that all cache indices are in a ready state to accommodate incoming data. When a new cache line arrives, the **getVictim** function is called to determine the specific location within the targeted cache index where this new cache line should be placed. Once the placement is decided, the **reset** function is employed to insert the new cache line into the identified location, initializing its parameters appropriately. Subsequently, if this cache line is accessed, the **touch** function is triggered to update the associated metadata, ensuring that the replacement policy maintains accurate state information. If the cache line becomes invalid due to changes in the system state or coherence requirements, the **invalidate** function is used to mark it as invalid, effectively removing it from the active set of cache lines.

When all the ways within a particular index are occupied, the **getVictim** function is invoked again to determine which cache line should be evicted to make space for the new entry. This decision is guided by the specific logic of the implemented replacement policy. A significant part of our Shepherd Cache Replacement Algorithm implementation is focused within this function, which governs the decision-making process for evictions.

3.1 Extension of ReplacementData for Shepherd Policy

To implement the Shepherd Cache Replacement Policy (SHP), we extended the existing 'ReplacementData' data structure in the cache subsystem. The new implementation introduces a custom structure, 'SHPReplData', which encapsulates metadata and functionality required by the policy.

Design and Attributes:

The 'SHPReplData' structure includes the following key attributes:

- **lastTouchTick:** Records the simulation tick when the cache block was last accessed. This attribute is critical for implementing Least Recently Used (LRU) replacement among blocks with invalid ranks and for determining the relative recency of blocks during replacement decisions.

- **tickInserted:** Captures the simulation tick when the cache block was inserted into the cache. This is used to implement First-In-First-Out (FIFO) replacement among SC blocks.

- **isMC:** A boolean flag indicating whether a block belongs to the Main Cache (MC) or Shepherd Cache (SC) category.

- **arraySize:** An integer specifying the SC-associativity, which determines the size of the rank matrix. This value helps allocate sufficient memory for managing ranks associated with each SC block.

- **dynamicArray:** A dynamically allocated 1D array (long int*) created for each cache block. The size of this array is determined by arraySize. It stores the ranks corresponding to SC blocks, essentially forming the rank matrix mentioned in the original Shepherd policy paper.

- **arrayIndex:** Valid only for SC blocks, this attribute holds the index in the rank matrix where the ranks corresponding to the current SC block are stored.

Implementation Details:

The design of 'SHPReplData' extends the standard 'ReplacementData' functionality by incorporating metadata specific to the Shepherd policy. The inclusion of timestamps (lastTouchTick and tickInserted) enables hybrid replacement logic:

- LRU is applied among blocks with invalid ranks.
- FIFO among the SC blocks, ensuring a simple and straightforward algorithm to manage their replacement in the cache.

The dynamicArray provides a flexible structure to store ranks for each SC block. These ranks form the backbone of the Shepherd policy, enabling the prioritization of blocks during replacement.

One deviation from the original Shepherd paper is the handling of the next value counter. Instead of introducing a new data structure, we optimized the implementation by reusing lastTouchTick to infer the relative ranking of blocks. This approach simplifies the design and minimizes additional memory overhead while maintaining the policy's core functionality.

3.2 Cache Initialization and Transition to Shepherd Cache State

At the initial stage, when the cache is empty, all blocks are inserted as SC blocks, and once all blocks at a given cache index have been inserted, the cache undergoes a transition into Shepherd Cache state.

Transition to Shepherd Cache State:

Once all blocks in a particular index are inserted, the cache enters the Shepherd Cache state. In this state, the cache is divided into s SC blocks and $n - s$ MC blocks, where:

- s is the SC-associativity, specifying the number of SC blocks in each index.
- $n - s$ represents the remaining MC blocks, these are the main cache blocks on which we try to emulate the optimal replacement policy using the look-ahead window provided by SC blocks.

This division occurs based on the FIFO logic that uses the tickInserted timestamps to maintain the correct order of insertion. The oldest s blocks will be marked as SC blocks and their arrayIndex will be set. The remaining $n - s$ blocks will be marked as MC blocks. From this point onwards, any access to the blocks is used to set ranks for the SC blocks. These ranks are central to the Shepherd policy and will guide eviction decisions. As the ranks are updated, they are used to determine the victim block when there is a need to evict a cache entry.

3.3 Cache Hit Behavior in Shepherd Cache Algorithm

When a cache block is accessed and results in a hit, the Shepherd Cache algorithm prescribes specific actions to update the metadata associated with the block. These updates ensure that the ranking mechanism remains accurate and consistent with the policy's principles. The primary focus during a hit is on managing the block's rank with respect to SC blocks and ensuring that it reflects the most recent access appropriately.

Handling Ranks on a Cache Hit:
In the Shepherd Cache algorithm:

1. If the rank of the block with respect to any SC is invalid, it is updated using the next value counter (current tick value in our case).
2. If the block already has a valid rank, the rank remains unchanged to preserve the stability of the ranking mechanism.

Our implementation replicates this behavior through the touch function by:

- Updating lastTouchTick:
  - Upon a hit, the lastTouchTick is updated to curTick, representing the simulation tick when the block is accessed. This timestamp is critical for maintaining access recency information.

- Updating Invalid Ranks:
  - If any rank of the block with respect to SC blocks is invalid, it is updated to curTick.
  - This ensures that the rank accurately reflects the block's first access since the SC block was inserted.
- Preserving Valid Ranks:
  - If a rank is already valid, it remains unchanged. This prevents unnecessary updates and preserves the existing rank hierarchy among blocks.

Ensuring Stability of Ranks:

When an SC block is first inserted, all ranks with respect to it are initialized to an invalid value. The first time a block is accessed after this insertion, its invalid ranks are updated to reflect the access tick. Subsequent accesses do not alter these ranks since they are already valid, maintaining the integrity of the ranking system. This behavior aligns with the Shepherd Cache policy's principle that ranks should only change when necessary, avoiding frequent updates that could disrupt the ranking order.

Rationale:

By managing ranks in this manner, the Shepherd Cache policy ensures that:

- The ranking system accurately reflects block usage patterns, particularly for SC blocks.
- Only the first access after an SC block's insertion triggers rank updates, minimizing computational overhead.
- Valid ranks remain stable, reducing the risk of unnecessary rank fluctuations that could lead to suboptimal eviction decisions.

3.4 Cache Miss Handling and Block Replacement

In the Shepherd Cache Replacement Policy, a cache miss triggers the need to replace one of the cache blocks in the given index. The replacement decision is guided by a combination of rank-based evaluation and a hybrid LRU mechanism, ensuring an intelligent and adaptive approach to maintaining cache efficiency.

Candidate Selection for Replacement:

Upon a cache miss, the candidates for replacement are selected from two categories within the cache index:

1. MC blocks.
2. The oldest SC block.

Replacement Logic:

The replacement logic is implemented in the getVictim function, which follows these steps:

1. Finding the Oldest SC Block: Using the tickInserted attribute, the algorithm identifies the SC block with the earliest insertion time. This block becomes part of the candidate pool along with the MC blocks.
2. Evaluating Ranks:
    - If any of the candidate blocks have invalid ranks (not accessed after the Oldest SC is inserted), the algorithm applies the LRU mechanism on those blocks using the last-TouchTick attribute. The block with the least recent access is selected as the victim for eviction.
    - If all candidates have valid ranks, the block with the largest rank is selected as the victim.

Handling Eviction and Block State Transition:

If an MC block is selected as the victim:

1. The oldest SC block transitions into the MC:
    - The isMC attribute of the SC block is set to true, marking it as MC.
    - The rank matrix entry associated with this SC block is preserved but reinterpreted for the transitioning block.

The rank matrix entry (arrayIndex) associated with the Oldest SC block is transferred to the newly inserted block, and the ranks with respect to the new SC will be updated in this index. The victim block location is marked as an SC block (isMC set to false). This is to make sure that the newly placed block will become SC (Since the new block will be placed here). Ranks of all blocks with respect to the new SC are reset to invalid values.

Inserting the New Block:

The new block is inserted using the reset function, which performs the following operations:

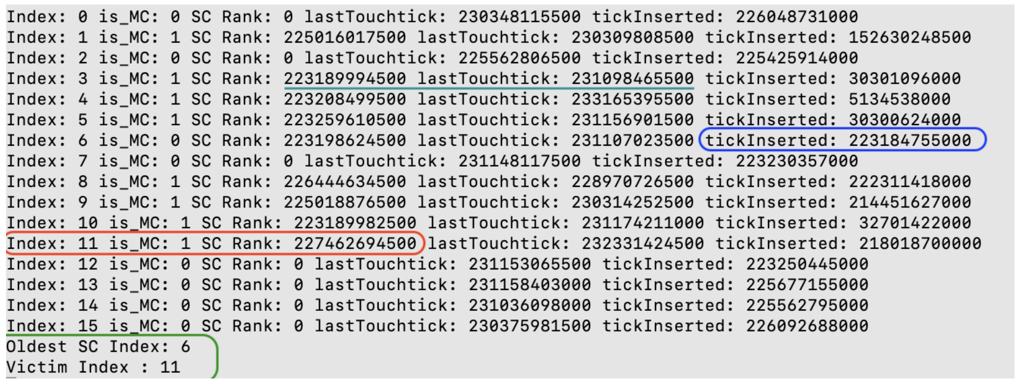
1. Metadata Initialization:
    - The lastTouchTick and tickInserted attributes of the new block are set to curTick (the current simulation tick), initializing the block's access metadata.
2. Rank Initialization:
    - The rank of the new SC block with respect to itself is set to an invalid value, as per the Shepherd Cache algorithm.
    - Ranks of other blocks with respect to the new SC block are initialized to 0, signifying that this SC block is the youngest and cannot be replaced by other SC blocks.

3.5 Verification of SC Algorithm

To validate the functionality of the Shepherd Cache (SC) algorithm, we printed some data as shown in Figure 1. The algorithm operates under two scenarios: (1) when all MC entries have valid ranks, meaning each entry has been accessed at least once after the oldest SC was inserted, and (2) when at least one MC entry has an invalid rank. In the first case, the SC replacement algorithm is used, while in the second case, LRU is applied among the entries with invalid ranks.

Case 1: Validation of SC Replacement Algorithm for Valid Ranks

In Figure 1, we first identify the oldest SC by filtering SC entries and selecting the one with the smallest tickInserted value. Here, SC entries are found at indexes 0, 2, 6, 7, 12, 13, 14, and 15, and the oldest SC is correctly identified as Index 6, with a tickInserted value of 223184755000. This matches the algorithm's printed result for the oldest SC index.



```
Index: 0 is_MC: 0 SC Rank: 0 lastTouchtick: 230348115500 tickInserted: 226048731000
Index: 1 is_MC: 1 SC Rank: 225016017500 lastTouchtick: 230309808500 tickInserted: 152630248500
Index: 2 is_MC: 0 SC Rank: 0 lastTouchtick: 225562806500 tickInserted: 225425914000
Index: 3 is_MC: 1 SC Rank: 223189994500 lastTouchtick: 231098465500 tickInserted: 30301096000
Index: 4 is_MC: 1 SC Rank: 223208499500 lastTouchtick: 233165395500 tickInserted: 5134538000
Index: 5 is_MC: 1 SC Rank: 223259610500 lastTouchtick: 231156901500 tickInserted: 30300624000
Index: 6 is_MC: 0 SC Rank: 223198624500 lastTouchtick: 231107023500 tickInserted: 223184755000
Index: 7 is_MC: 0 SC Rank: 0 lastTouchtick: 231148117500 tickInserted: 223230357000
Index: 8 is_MC: 1 SC Rank: 226444634500 lastTouchtick: 228970726500 tickInserted: 222311418000
Index: 9 is_MC: 1 SC Rank: 225018876500 lastTouchtick: 230314252500 tickInserted: 214451627000
Index: 10 is_MC: 1 SC Rank: 223189982500 lastTouchtick: 231174211000 tickInserted: 32701422000
Index: 11 is_MC: 1 SC Rank: 227462694500 lastTouchtick: 232331424500 tickInserted: 218018700000
Index: 12 is_MC: 0 SC Rank: 0 lastTouchtick: 231153065500 tickInserted: 223250445000
Index: 13 is_MC: 0 SC Rank: 0 lastTouchtick: 231158403000 tickInserted: 225677155000
Index: 14 is_MC: 0 SC Rank: 0 lastTouchtick: 231036098000 tickInserted: 225562795000
Index: 15 is_MC: 0 SC Rank: 0 lastTouchtick: 230375981500 tickInserted: 226092688000
Oldest SC Index: 6
Victim Index : 11
```

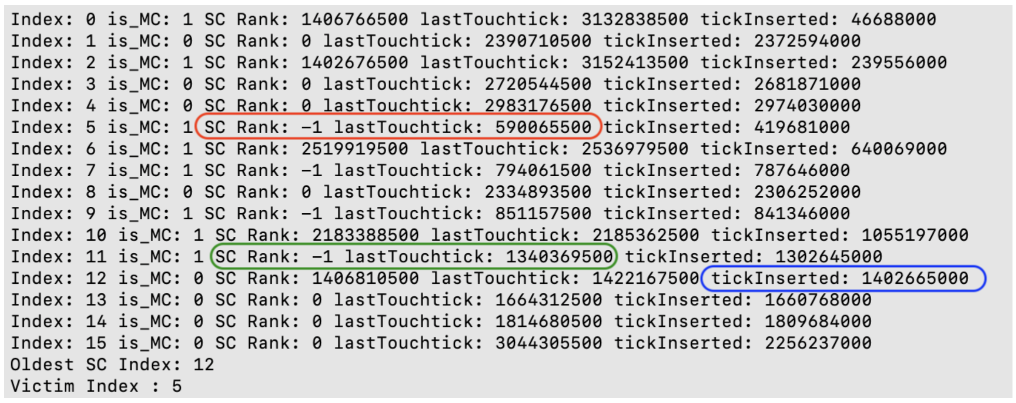
Figure 1: Shepherd Cache Algorithm Verification - Case 1

Next, the algorithm selects the victim for eviction by identifying the entry with the highest SC Rank among valid candidates. From the data, Index 11 has the highest SC Rank (227462694500) and is correctly chosen as the victim. Additionally, SC entries younger than the oldest SC (e.g., indexes 0, 2, 7, 12, 13, 14, 15) have an SC Rank of 0, ensuring they are not considered for eviction. Observing lastTouchTick values, we note that some entries, like Index 3, have been accessed multiple times after the SC Rank was assigned, but the rank remains unchanged as per the algorithm's design.

This verification confirms that the SC algorithm accurately identifies the oldest SC, applies appropriate ranks, and correctly determines the victim for eviction based on the highest rank.

Case 2: Validation of LRU for Invalid Ranks

The process of identifying the oldest SC in this case is the same as in Case 1. By evaluating the tickInserted values, the oldest SC is determined to be Index 12, with a tickInserted value of 1402665000, which matches the printed result in the data. In this case, several entries (e.g., Indexes 5, 7, 9, and 11) have a lastTouchTick that is less than the tickInserted value of the oldest SC. This indicates that these entries have not been accessed even once since the oldest SC was inserted into the cache. As a result, their ranks are invalid and are shown as -1 in the data as can be seen from Figure 2.



```
Index: 0 is_MC: 1 SC Rank: 1406766500 lastTouchtick: 3132838500 tickInserted: 46688000
Index: 1 is_MC: 0 SC Rank: 0 lastTouchtick: 2390710500 tickInserted: 2372594000
Index: 2 is_MC: 1 SC Rank: 1402676500 lastTouchtick: 3152413500 tickInserted: 239556000
Index: 3 is_MC: 0 SC Rank: 0 lastTouchtick: 2720544500 tickInserted: 2681871000
Index: 4 is_MC: 0 SC Rank: 0 lastTouchtick: 2983176500 tickInserted: 2974030000
Index: 5 is_MC: 1 SC Rank: -1 lastTouchtick: 590065500 tickInserted: 419681000
Index: 6 is_MC: 1 SC Rank: 2519919500 lastTouchtick: 2536979500 tickInserted: 640069000
Index: 7 is_MC: 1 SC Rank: -1 lastTouchtick: 794061500 tickInserted: 787646000
Index: 8 is_MC: 0 SC Rank: 0 lastTouchtick: 2334893500 tickInserted: 2306252000
Index: 9 is_MC: 1 SC Rank: -1 lastTouchtick: 851157500 tickInserted: 841346000
Index: 10 is_MC: 1 SC Rank: 2183388500 lastTouchtick: 2185362500 tickInserted: 1055197000
Index: 11 is_MC: 1 SC Rank: -1 lastTouchtick: 1340369500 tickInserted: 1302645000
Index: 12 is_MC: 0 SC Rank: 1406810500 lastTouchtick: 1422167500 tickInserted: 1402665000
Index: 13 is_MC: 0 SC Rank: 0 lastTouchtick: 1664312500 tickInserted: 1660768000
Index: 14 is_MC: 0 SC Rank: 0 lastTouchtick: 1814680500 tickInserted: 1809684000
Index: 15 is_MC: 0 SC Rank: 0 lastTouchtick: 3044305500 tickInserted: 2256237000
Oldest SC Index: 12
Victim Index : 5
```

Figure 2: Shepherd Cache Algorithm Verification - Case 2

Among these invalid-ranked entries, the algorithm applies the Least Recently Used (LRU) policy to determine the victim for eviction. To do this, we look at the lastTouchTick values of these entries. Here, Index 5 has the smallest lastTouchTick value (590065500), making it the least recently used entry. Therefore, Index 5 is correctly identified as the victim for eviction.

The printed victim index (Index 5) matches our logical verification, confirming that LRU is correctly applied when at least one entry has an invalid rank.

4 Results

In this section, we present the performance comparison of our Shepherd cache implementation with two other policies, FIFO and LRU. We ran all three policies on 15 benchmarks (SPEC-2017).

4.1 Benchmarks Data

We implemented the SC replacement policy and evaluated its performance using benchmarks across six configurations. The experiments were conducted on a 16-way associative L2 cache with the following SC and MC configurations:

1. SC: 4-way, MC: 12-way

2. SC: 8-way, MC: 8-way

For each configuration, we tested three L2 cache sizes: **512 kB, 1 MB, and 2 MB**. In all experiments, the L1D cache size and L1I cache size were set to 16 kB each. For the comparison with FIFO and LRU replacement policies, we used equivalent cache setups i.e. with 16-way associative caches of corresponding sizes.

Figures 3, 4 and 5 give the miss-rate with 512KB, 1MB, and 2MB L2 cache respectively. The corresponding CPI metric values are given in Figures 6, 7, and 8.

The miss rate data shows that Shepherd Cache (SC) consistently outperforms FIFO across all benchmarks and configurations. For smaller cache sizes like 512kB, SC-4, and SC-8 configurations demonstrate significant improvements over FIFO, particularly in benchmarks such as perlbench_s and gcc_s. However, when compared to LRU, SC performs closely in most benchmarks, with occasional slight advantages. In cases like mcf_s, the miss rates for SC and LRU are nearly identical, suggesting that the choice of replacement policy has minimal impact in these scenarios.

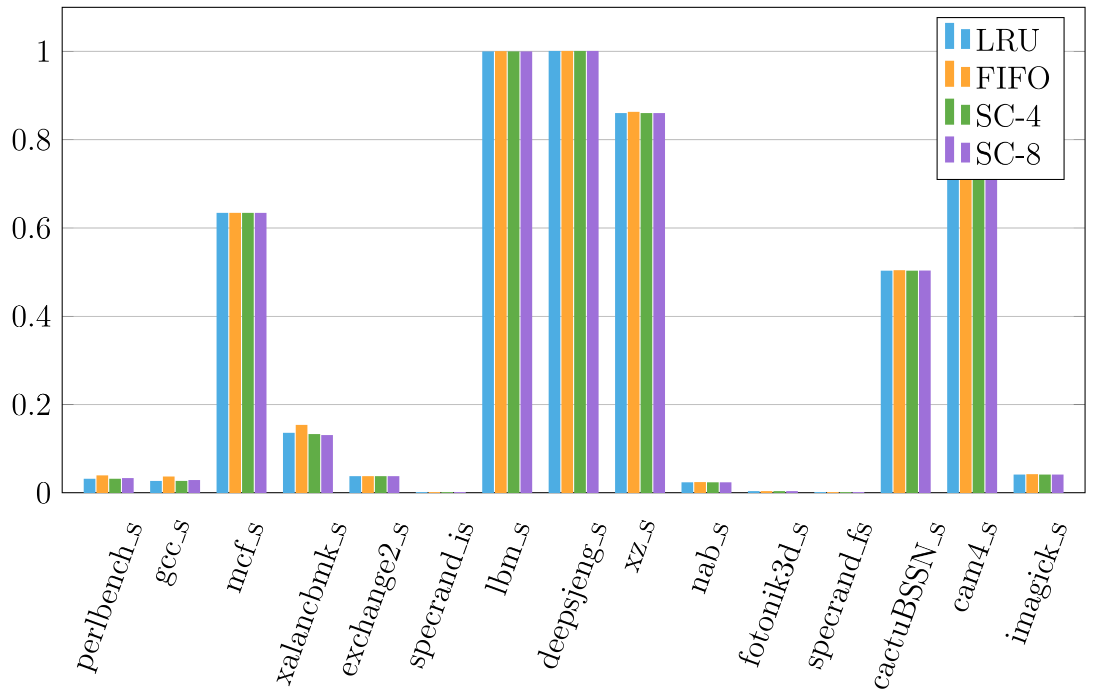


Figure 3: Miss-rate with 512KB L2 size

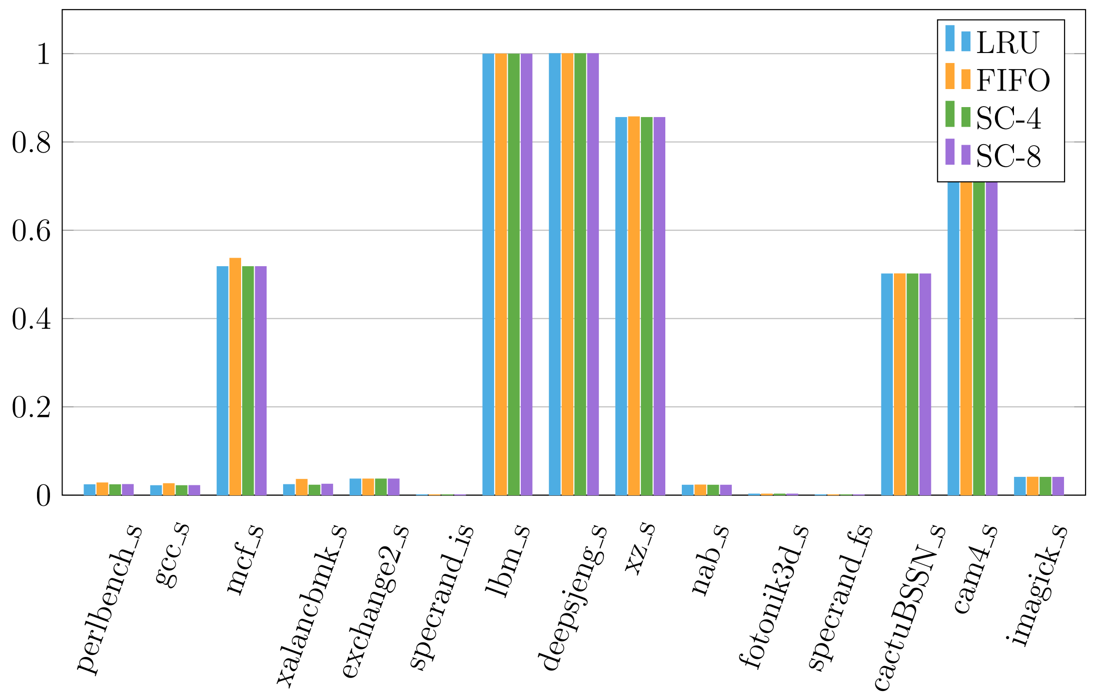


Figure 4: Miss-rate with 1MB L2 size

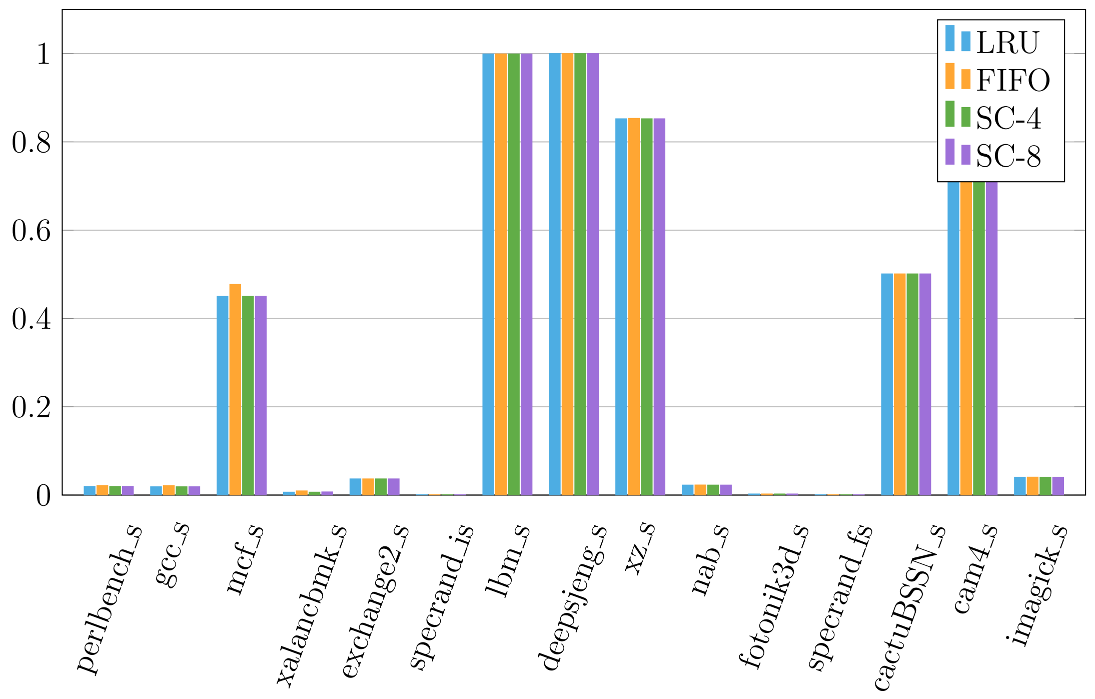


Figure 5: Miss-rate with 2MB L2 size

When looking at CPI, SC configurations generally follow the same trend as the miss rate. In benchmarks like perlbench_s and xalancbmk_s, SC shows a small advantage over LRU, indicating that rank-based evictions are effectively applied. However, in benchmarks such as gcc_s, the CPI values for SC and LRU are almost indistinguishable, reflecting their similar miss rates. Across all cache sizes, SC maintains consistent performance, with noticeable improvements over FIFO and occasional advantages over LRU, depending on the benchmark and configuration.

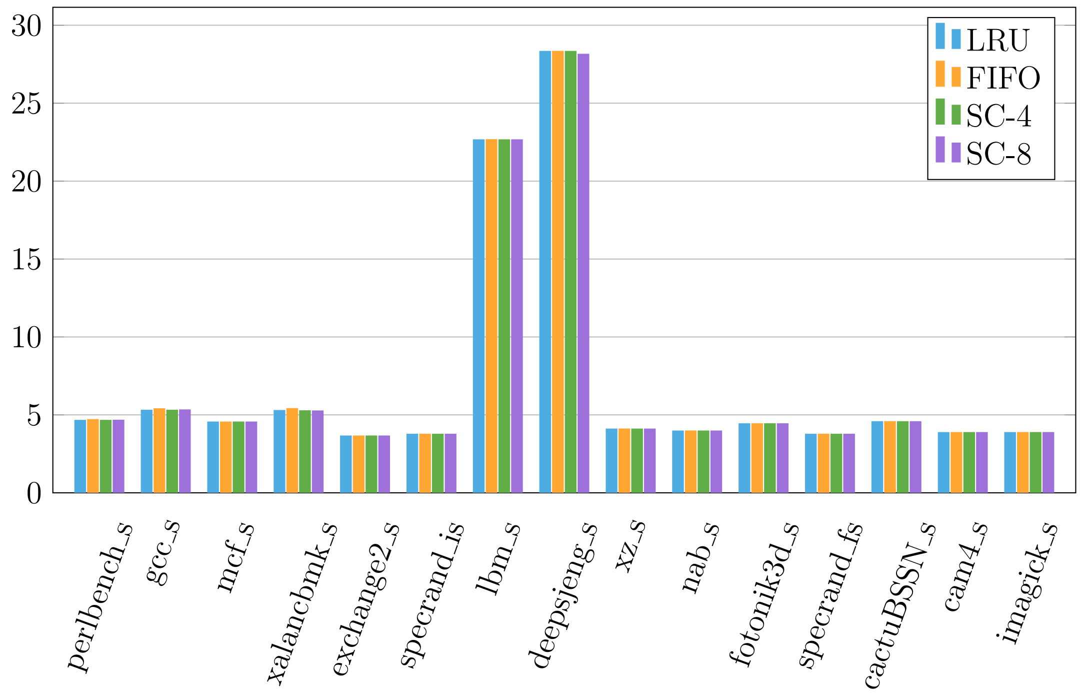


Figure 6: CPI with 512KB L2 size

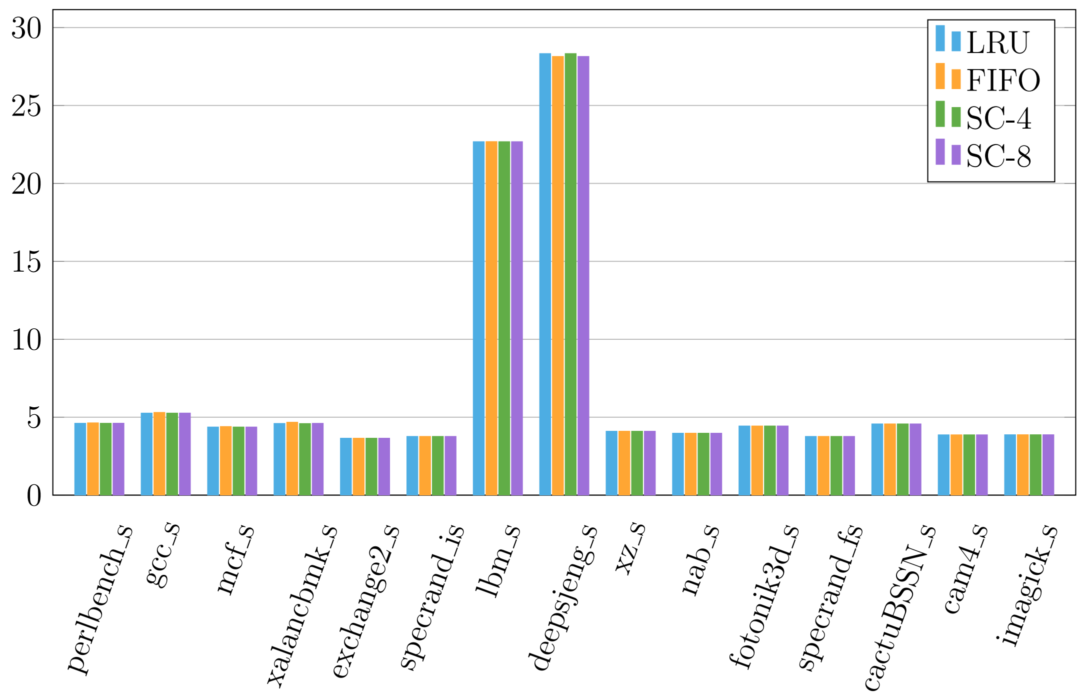


Figure 7: CPI with 1MB L2 size

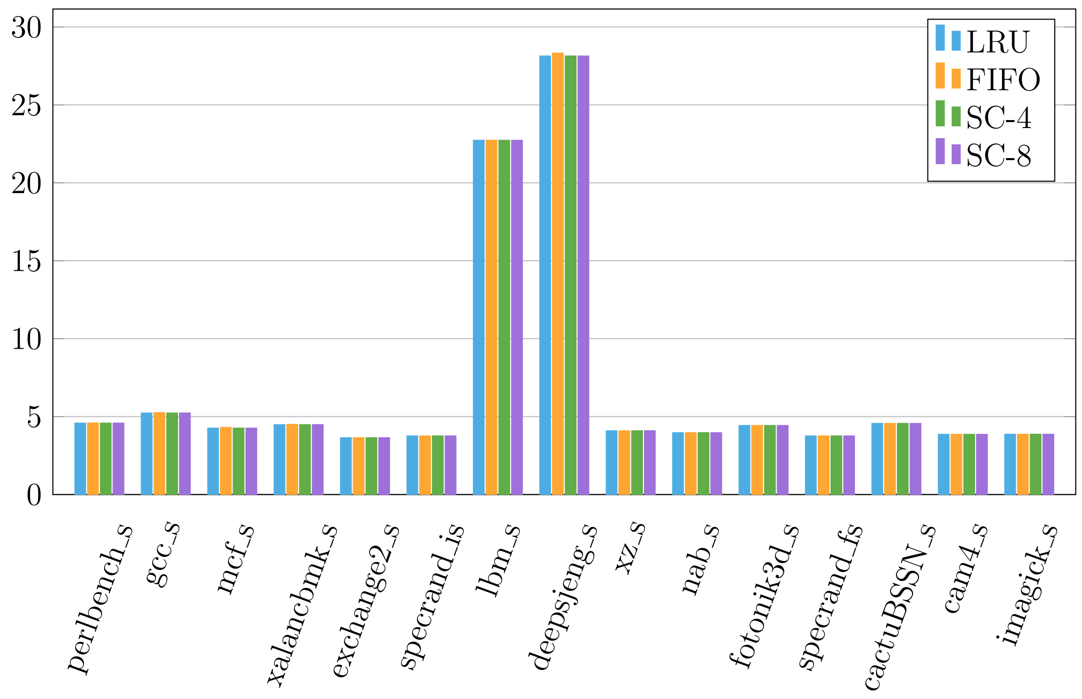


Figure 8: CPI with 2MB L2 size

4.2 Inferences

It is evident from the results that the SC performs better than the FIFO across all cases. But it performs almost similarly to LRU in most of the cases but slightly outperforms in specific scenarios. This behavior can be attributed to how SC handles invalid ranks. As discussed in Section 3.4, when at least one block in the MC has an invalid rank, the algorithm falls back to LRU for victim selection among those blocks. In such cases, the victim chosen by SC may align with what LRU would select, leading to similar performance. For SC to consistently outperform LRU, all MC blocks must be accessed at least once after the insertion of the oldest SC block, enabling the algorithm to leverage rank-based decisions. However, this scenario is relatively rare, limiting SC's opportunity to deviate significantly from LRU.

The slightly better performance of SC in some benchmarks stems from its ability to use rank-based evictions when sufficient access windows are available. Additionally, SC inherently avoids replacing the youngest blocks in the cache, as younger SC blocks are reserved for future evaluations. This decision can occasionally lead to less optimal replacements compared to LRU, which does not impose such restrictions. SC's performance is therefore highly dependent on workload characteristics. It is more effective when temporal locality is largely handled by the L1 cache, leaving L2 to optimize spatial locality or workloads with less frequent accesses. In scenarios where small improvements in miss rate are critical, especially for workloads with low temporal locality, SC provides value but comes at the cost of increased implementation complexity and hardware requirements.

5 Conclusion

We implemented a slightly modified version of the Shepard Cache in gem5 and evaluated it against other replacement policies like FIFO and LRU. Our results show that SC performs better only in comparison to FIFO. It performs almost the same as LRU. There might be workloads in which SC performs much better than LRU. This suggests that there might be other variables that were considered by the original authors when running the evaluation that are not obvious from the paper. Further research on using different policies within SC might help in improving its performance.

References

[1] Kaushik Rajan and Govindarajan Ramaswamy. "Emulating optimal replacement with a shepherd cache". In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE. 2007, pp. 445–454.

[2] Norman P Jouppi. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers". In: *ACM SIGARCH Computer Architecture News* 18.2SI (1990), pp. 364–373.

[3] Ranjith Subramanian, Yannis Smaragdakis, and Gabriel H Loh. "Adaptive caches: Effective shaping of cache behavior to workloads". In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE. 2006, pp. 385–396.

[4] Moinuddin K Qureshi et al. "Adaptive insertion policies for high performance caching". In: *ACM SIGARCH Computer Architecture News* 35.2 (2007), pp. 381–391.