CS 2510 - Project Report

Ziwei Quan
School of Computing and Information
University of Pittsburgh
Email: ziq15@pitt.edu

Shuhao Yu
School of Computing and Information
University of Pittsburgh
Email: shy158@pitt.edu

***Abstract*—In this project, we have chosen to reimplement a paper titled 'FIFO Queues are All You Need for Cache Eviction,' which was published at SOSP2023. The paper proposes a method for quickly evicting data accessed only once to improve the hit rate of FIFO. Caching is a crucial component in distributed systems, such as reducing communication overhead by caching hot data; solving data consistency issues with appropriate caching strategies to reduce the cost of data migration and synchronization. Our plan for the project is to understand this algorithm and compare its performance under different parameters (such as cache size) through experiments. Ultimately, we aim to gain a deeper understanding of cache eviction algorithms.**

I. Introduction

In this project, we focus on enhancing cache eviction strategies by implementing the method introduced in the paper "FIFO Queues are All You Need for Cache Eviction,"[1] presented at SOSP2023.

Cache is a high-speed buffer used in computer systems to temporarily store data, aiming to improve data access speed and system performance. It typically resides between the processor and main memory, storing recently or frequently accessed data to reduce accesses to main memory and enhance data access efficiency. Caches are critical in distributed systems for minimizing data retrieval times and network overhead by storing frequently accessed information. This project aims to explore a novel approach that prioritizes the eviction of single-access data to improve the efficiency of FIFO caches.

Our goal is to thoroughly understand this new eviction strategy and assess its performance across various scenarios, such as different cache sizes. We hypothesize that this method will yield a higher hit rate than traditional approaches because, based on observation, a significant portion of data inserted in reality is not accessed afterward. Therefore, we propose using S3-FIFO for rapid eviction of such data.

To evaluate this, we designed a systematic experimental framework involving simulation of different caching scenarios. These experiments allow us to compare the hit rates and eviction efficiency of the proposed method against existing strategies under controlled conditions.

The outcomes of our project are promising. We successfully implemented the algorithm and our initial results indicate that it consistently outperforms standard FIFO caching in scenarios characterized by high variability in data access patterns. This success was demonstrated through detailed performance metrics and comparative analysis, confirming the efficacy of the proposed cache eviction strategy.

II. Methods

A. Previous Methods

1) LRU: The LRU (Least Recently Used) cache algorithm is a commonly used block replacement algorithm for managing data in caches. Its core idea is to prioritize the eviction of data that has not been used for the longest time when the cache is full. In implementation, the LRU cache typically uses a doubly linked list and a hash table. The doubly linked list arranges each piece of data according to its order of use, with the most recently used data at the head of the list and the least recently used data at the tail. The hash table stores the value of each data item and a pointer to its corresponding node in the list, supporting fast access and update operations. When accessing or adding a data item, if the data already exists, it is moved to the head of the list; if the data does not exist and the cache is full, the data item at the tail of the list is removed, and new data is added to the head.

Although the LRU cache algorithm performs well in many scenarios, it also has some drawbacks. First, the LRU algorithm assumes that past access patterns will predict future ones, which may not always hold true, especially in environments where access patterns change frequently. Additionally, LRU implementation requires extra storage space to maintain the order of data items, which could be problematic in systems with strict memory constraints. Also, the efficiency of the LRU algorithm may decrease when handling large volumes of data with similar access times due to frequent updates to the linked list, leading to performance bottlenecks.

2) FIFO: The FIFO (First-In-First-Out) cache algorithm is a straightforward and intuitive block replacement algorithm used for managing data in caches. It operates on the principle of "first in, first out," meaning that the data items that

entered the cache earliest are the first to be replaced. In its implementation, the FIFO cache typically uses a queue structure to track the order in which data enters. When new data needs to be added to a full cache, the old data at the front of the queue is removed to make space for the new data.

The main drawback of the FIFO cache algorithm is that it may lead to a lower cache hit rate because it replaces data based solely on the order of entry into the cache, without considering the actual frequency of use or recent access of the data. This means that even if some data items are frequently accessed, they could still be replaced because they were among the earliest to enter the cache. This replacement strategy overlooks the usage patterns of data items, potentially leading to the premature eviction of high-value data, thus requiring more frequent reloads from slower storage, increasing latency, and reducing overall system efficiency.

B. Proposed Method

In recent years, although many newly proposed cache replacement algorithms are designed based on the LRU (Least Recently Used) principle, this paper introduces an optimized algorithm based on the FIFO (First-In-First-Out) principle. This optimized algorithm aims to retain the simplicity and intuitiveness of FIFO while addressing the issues of low cache hit rates inherent in traditional FIFO algorithms.

"The one-hit-wonder ratio" measures the proportion of objects that are requested only once in a trace. It is commonly used in content delivery networks (CDNs) due to a significant number of one-hit wonders. Although this ratio varies across different types of cache workloads, we observe that shorter request sequences (containing fewer unique objects) often have higher one-hit-wonder ratios. In our subsequent analysis, we quantify sequence length based on the number of unique objects.

start time	end time	sequence length (# objects)	# one-hit wonder	one-hit-wonder ratio
1	17	5	1 (E)	20%
1	7	4	2 (C, D)	50%
1	4	3	2 (B, C)	67%

Fig. 1. A shorter sequence has a higher one-hit-wonder ratio

Figure 1 illustrates this observation with a toy example. The request sequence comprises seventeen requests for five objects, with one object (E) accessed only once, resulting in a one-hit-wonder ratio of 20% for the sequence. Considering a shorter sequence from the 1st to the 7th request, where two of the four unique objects (C, D) are requested only once, leads to a one-hit-wonder ratio of 50%. Similarly, the one-hit-wonder ratio for a shorter sequence from the 1st to the 4th request is 67%.

Based on the above observation, the author proposes whether we can simply use a small probabilistic FIFO queue to ensure that one-hit wonders are removed after inserting a fixed number of objects?

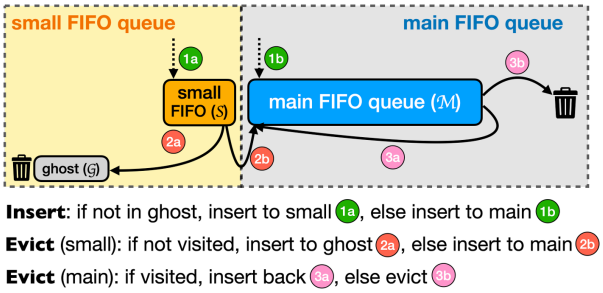

Fig. 2. An illustration of S3-FIFO

As shown in Figure 2, S3-FIFO uses three FIFO queues: a small FIFO queue (S), a main FIFO queue (M), and a ghost FIFO queue (G). We choose to allocate 10% of the cache space to S based on experiments, finding that this proportion works well. Then, M uses the remaining 90% of the cache space. The ghost queue G stores the same number of ghost entries (only hash key values, no actual data) as M.

Cache read: S3-FIFO uses two bits per object to track the object's access status, similar to a capped counter with a maximum frequency of 3. Cache hits in S3-FIFO increment the counter atomically by one. It's worth noting that due to the maximum limit being 3, most requests for hot data do not require updates.

Cache write: New objects are inserted into S if they are not in G; otherwise, they are inserted into M. When S is full, if an object at the tail has been accessed more than once, it is moved to M; otherwise, it is moved to G, and its access bits are cleared during the move. When G is full, it evicts objects in FIFO order. M uses an algorithm similar to FIFO-Reinsertion but tracks access information using two bits. Objects that have been accessed at least once are reinserted with one access bit set to 0 (similar to decreasing frequency by 1).

III. Expreimental Setup

Evaluation Environment. Unlike the resources that implement the whole experiment on 1 million cores could lab. The evaluation process in this project is based on two major open-source tools. First, cacheLib [3] which is the

pluggable cache engine to support the building and recording of S3-FIFO and other caches in the control group. Second, libCacheSim [2] which is a similator to run different caches with various algorithms like LRU, FIFO, and TwoQ. And our experiment and test will be implemented based on these tools.

Simulators. Since the recommended environment of libCacheSim is the Linux kernel system. In this project, we plan to setup all cache algorithms and cache movement implements on Ubuntu 20.04 LTS which is a classical version for most similar testbeds. To accomplish the environment of Ubuntu, a system simulator VMware Workstation Pro will support the performance of the whole experiment. And the detailed environment factors is shown in Figure3.

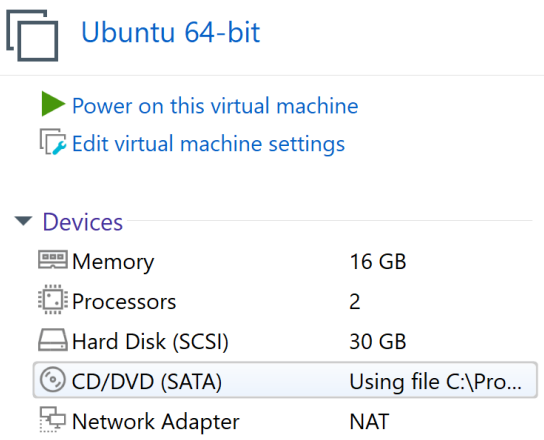

Fig. 3. An brief summary of environment parameters

Traces Datasets. We will evaluate different cache algorithms through trace datasets from various resources and companies. Thanks to the Parallel Data Lab at Carnegie Mellon University for providing enough and multiple-sizes-available datasets [4]. We will select datasets from the website to support our exploration.

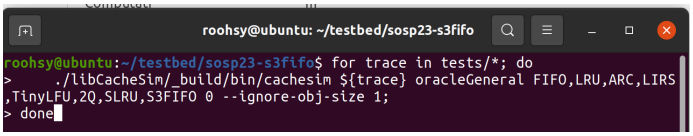

Fig. 4. A command to implement the tests

Implementations. After the preparation step, the project will follow the experiment setup rules that are provided by libCacheSim. We need to download the data and store them in a folder. Then using the command line shown in Figure4 to start the monitoring and recording the result on each different metrics. In local test on Ubuntu 20.04 LTS, we use a for loop to run all datasets in the folder traces.

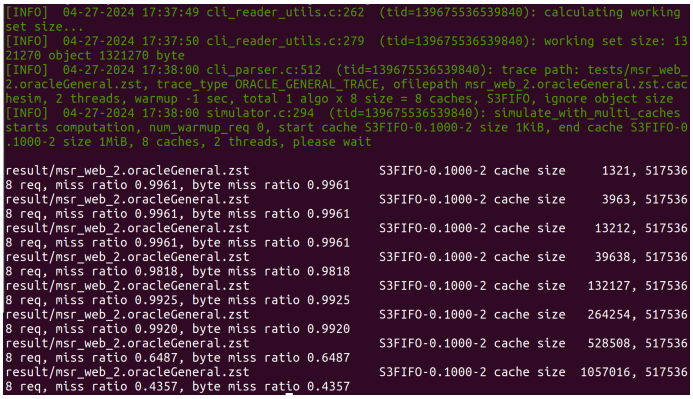

Fig. 5. A sample of output record

Results. In the cacheLibSim, it will collect the results and compress them into Z-standard files. Meanwhile, after each operation on different algorithms, the result record text will be outputted on the screen shown in Figure5.

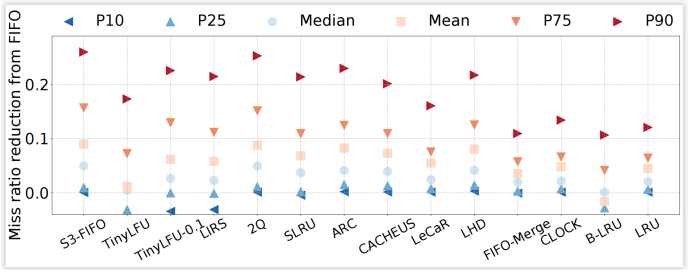

Fig. 6. A sample of miss ratio data plot

Plots. And then we can use the external library of Python: matplotlib to plot the straightforward statistics, which is similar as Figure6. The libCacheSim will store them as PDF format files. For different metrics, it has different command to implement different types of visualization

IV. Evaluation

In this project, we divide the evaluation of a cache algorithm into three major parts: efficiency (miss ratio), performance (throughput) , and device friendliness (flash-write bytes). All these three metrics can be monitored by libCacheSim and exported as a diagram to show their ability.

Miss Ratio. The miss ratio is the most important and straightforward factor of cache efficiency. It represents the ratio when requests cannot get correct information blocks in the cache over all requests. The ability and bandwidth usage of a cache algorithm can also be analyzed based on different percentiles of the complete test data. In Figure7, the higher reduction represents the lower miss ratio.

Throughput. Throughput will show the highest request quantity of the cache. In other words, a higher throughput

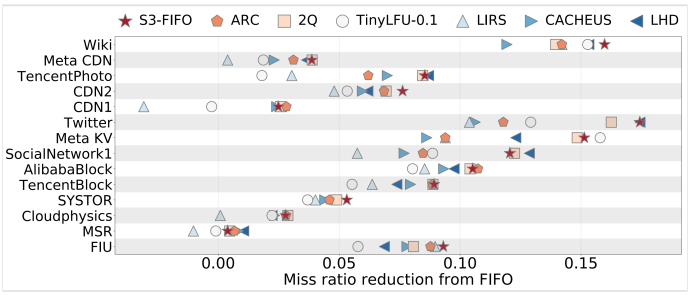

Fig. 7. A diagram of miss ratio results. In most datasets, the reduction (compared to FIFO) of S3-FIFO is in the top tier compared with other algorithms.

will reduce the pressure and workload on each core of the CPU, which composes the performance test. In this part, we will record the through as the total number of operations per second. The simulator will run different algorithms separately and combine them. We will use synthetic Zipf traces to show the results.

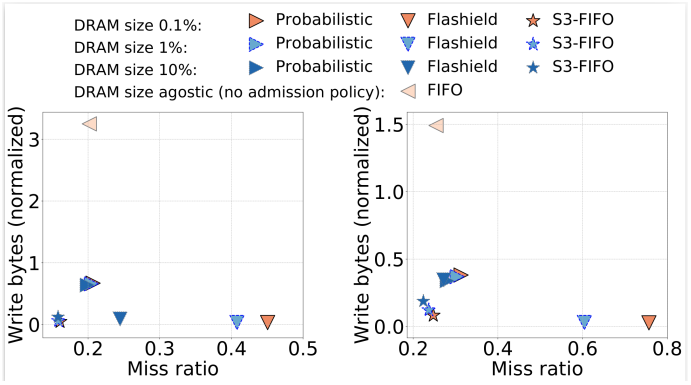

Fig. 8. A diagram of throughput. We can see that except for Segcache, S3-FIFO has a much higher progression than other algorithms.

Flash-write. Flash write is one interesting and special test case to evaluate friendliness like device health. The frequency of flash write is positively proportional to the lifetime of DRAM. In this part, we plan to choose the dataset that contains enough flash write operations like CDN from Tencent. And based on different ghost cache sizes, the result in Figure8 proves the average minimum write-bytes in S3-FIFO.

In this part, after comparing all three metrics, the results show that S3-FIFO contains multiple outstanding performances. It is hard to find an algorithm that is better than S3-FIFO on all tests. However, we cannot conclude the S3-FIFO is the best algorithm for the cache working. It is obvious that some new algorithms show their potential ability in throughput. For example, the Segcache algorithm imports timestamp and hash function have an outstanding performance in the throughput test shown in Figure9. And the following scalability of the S3-FIFO may not have enough advantages.

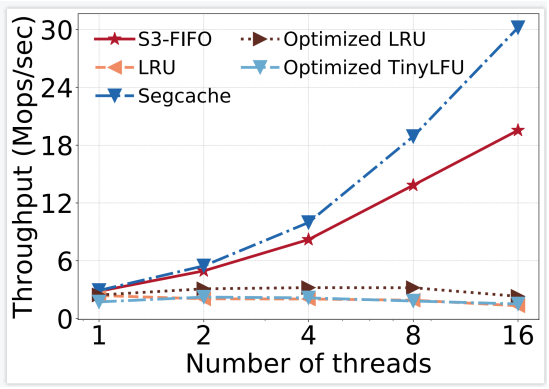

Fig. 9. A diagram of throughput. We can see that except for Segcache, S3-FIFO has a much higher progression than other algorithms.

But for current cache services and standards, S3-FIFO still is able to support devices to keep a high-performance work.

V. Limitations and Future Work

The main limitations of the project are the performance and ability of our devices. Ubuntu 20.04 LTS is a good testbed for libCacheSim. However, the whole simulator consumes too much space. Thus, more complex mechanisms like latency, access time, and performance on different cores CPU are not able to be implemented. And flash-write also causes the Ubuntu simulator to corrupt or stuck, which is still not solved by increasing RAM and cores. Thus, we uses the diagram that plot the data of the original paper of S3-FIFO to exhibit the comparisons.

VI. Conclusion

In this project, we can get a chance to practice a detailed cache evaluation. During the whole comparison and stress tests, the implementation of libCachSim and the building of the environment help us to understand the working mechanism of multiple cache algorithms. Meanwhile, we recover the similar results that the paper of S3-FIFO provides on the local machine, which proves and evaluates the stable performance of S3-FIFO on different tests.

References

[1] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, K. V. Rashmi. 2023. FIFO Queues are All You Need for Cache Eviction. In ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany. ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3600006.3613147

[2] libCacheSim - building and running cache simulations. https://github.com/1a1a11a/libCacheSim

[3] CacheLib - Pluggable caching engine to build and scale high-performance cache services. https://cachelib.org/

[4] cacheDatesets - Parallel Data Lab at Carnegie Mellon University. https://ftp.pdl.cmu.edu/pub/datasets/twemcacheWorkload/cacheDatasets/