University of Waterloo

CS-450 Computer Architecture

Course Winter 2024

Final Project: S3Random

József IVÁN GAFO

Alejandro FERNANDEZ-PANIAGUA BOUTUREIRA

1. ABSTRACT:

Software caches are essential nowadays, enabling a notable increase in data access speed and avoiding repeated computation efficiently. Some examples are Memcached and Linux page cache. When designing them, one of the most determining factors for its usefulness is the eviction algorithm they use. Over the years, the majority have used FIFO because of some of its nice properties: simple, efficient, and flash-friendly. However, they lack efficiency (high miss ratio). It is for this reason that there is still space for improvement and after the design of a new simple and scalable FIFO based algorithm that uses three static queues (S3-FIFO) on the paper titled "FIFO Queues are all you need for Cache Eviction"[1] by Juncheng Yang,Yazhuo Zhang, Ziyue Qiu, Yao Yue and K.V.Rashmi , we attempted to further exploit the already proven advantages of S3-FIFO against the rest of the algorithms, by adding a randomness component to its decision making logic, which we conjectured to preserve efficiency in the results but also decrease the complexity for its implementation.

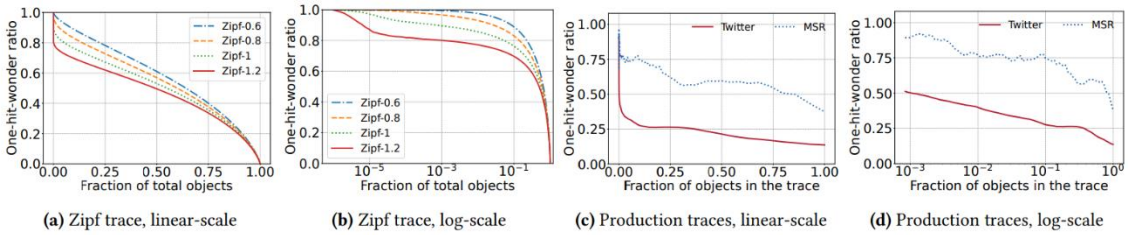
Hence, our work consists of the design of 3 alternative eviction algorithms whose fundamental structure is based on the S3-FIFO algorithm, aiming to perform approximately as well as the latter but also be a better option because of its simpler implementation. In global terms, all these approaches (S3-FIFO and our proposed algorithms) share a common goal underlying their structure: most objects in skewed workloads will only be accessed once in a short window of time, so it is critical to evict them early.

2. INTRODUCTION:

When it comes to measuring the validity of a cache, there are 3 main properties that a good cache should satisfy:

A cache should be efficient: it should have a low miss rate, allowing most of the requests to be fulfilled in short times; performant: serving data from the cache must result in minimal operations with a high throughput; and scalable: the number of cache hits it can serve per second grows with the number of CPU cores. The essence of a cache is the eviction algorithm, which dictates a cache's efficiency, throughput, and scalability. Since LRU is believed to be more efficient than FIFO, many proposed algorithms are often LRU-based, however they suffer from 2 problems: first, it requires two pointers per object, which is a significant storage over-head for relatively small workloads and secondly it is not scalable since a cache hit requires promoting the object to the head of the queue guarded by locking. In contrast, the algorithms that use simple FIFO-queue eviction policies decide to trade efficiency for throughput and scalability. The previous work we based our studies on proposed a simple, scalable, and efficient algorithm that uses only FIFO queues. The most important observation made in their work was that the object popularity in the cache workloads is usually skewed and was proven to follow a Zipf distribution form many empirical observations. Since a cache of size C only sees a sequence of C objects before evicting, most of these objects will be one-hit wonders (no request after insertion) when evicted, even though they may appear multiple times later in the entire trace. They showed that the median one-hit-wonder ratio over the complete trace

is 26% but when focusing on sequences that had only 10% of the objects in the trace, the median rocketed up to 72%. The following plots provide evidence for such a claim:



(a) Zipf trace, linear-scale (b) Zipf trace, log-scale (c) Production traces, linear-scale (d) Production traces, log-scale

Figure 1. An illustration comparing the different Zipf distributions. [1]

It is for this reason that they decided to devise S3-FIFO, which divides the cache into three static FIFO queues: small, main and ghost queue. The small queue is probationary and is relatively insignificant compared to the main queue (for example, 10% of the cache memory). It has the goal of filtering out the frequent one-hit wonders and stopping them from occupying a seat in the main queue, which is a technique summarized by the name of “quick demotion”. When evicting from the small queue, evicted objects are either sent to the ghost when not previously accessed or otherwise sent to the main. The main queue is the biggest queue (for example, 90% of the cache memory) and for eviction, it reinserts when the object is visited and otherwise evict completely from the cache. Finally, the Ghost queue does not have data but only stores as many ghost entries as the main queue. A diagram of how S3-FIFO works is shown below:

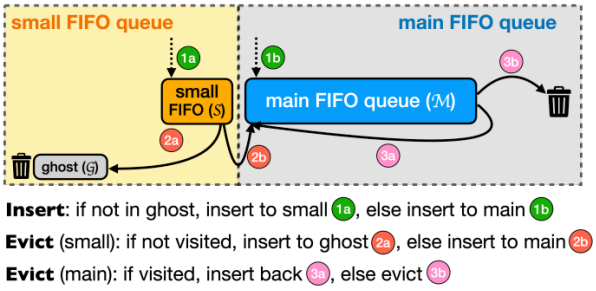
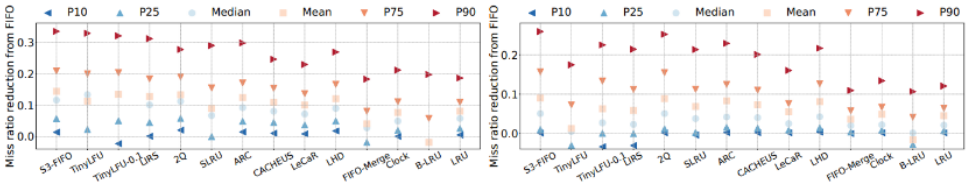


Figure2. An illustration of S3-FIFO [1]

On the other hand, the previous work we based our research on performed several evaluations of S3-FIFO and compared it with the other algorithms. For the traces, they used a total of 6594 production traces from 14 datasets, which contain over 856 billion requests, 61 billion objects, 21,008 TB traffic for 3,753 TB of data. In addition, simulations were done in libCacheSim [2], which is a cache simulator that is designed and tuned for high-throughput cache simulations that can process up to 20 million requests on a single CPU core.

Table 1. Datasets used in this work, the ones with no citation are proprietary datasets. For old datasets, we exclude traces with less than 1 million requests. The trace length used in measuring the one-hit-wonder ratio is measured in the fraction of objects in the trace.

Trace collections	Approx time	Cache type	time span (days)	# Traces	# Request (million)	Request (TB)	# Object (million)	Object (TB)	One-hit-wonder ratio full trace	10%	1%
MSR [104, 105]	2007	Block	7	13	410	10	74	3	0.56	0.74	0.86
FIU [83]	2008-11	Block	9-28	9	514	1.7	20	0.057	0.28	0.91	0.91
Cloudphysics [136]	2015	Block	7	106	2,114	82	492	22	0.40	0.71	0.80
CDN 1	2018	Object	7	219	3,728	3640	298	258	0.42	0.58	0.70
Tencent Photo [167, 168]	2018	Object	8	2	5,650	141	1,038	24	0.55	0.66	0.74
WikiMedia CDN [140]	2019	Object	7	3	2,863	200	56	13	0.46	0.60	0.80
Systor [84, 85]	2017	Block	26	6	3,694	88	421	15	0.37	0.80	0.94
Tencent CBS [163, 164]	2020	Block	8	4030	33,690	1091	551	66	0.25	0.73	0.77
Alibaba [2, 89, 139]	2020	Block	30	652	19,676	664	1702	117	0.36	0.68	0.81
Twitter [157]	2020	KV	7	54	195,441	106	10,650	6	0.19	0.32	0.42
Social Network 1	2020	KV	7	219	549,784	392	42,898	9	0.17	0.28	0.37
CDN 2	2021	Object	7	1273	37,460	4,925	2,652	1,581	0.49	0.58	0.64
Meta KV [11]	2022	KV	1	5	1,644	958	82	76	0.51	0.53	0.61
Meta CDN [11]	2023	Object	7	3	231	8,800	76	1,563	0.61	0.76	0.81



(a) Large cache size, 10% trace footprint

(b) Small cache size, 0.1% trace footprint

Figure 3. Each algorithm's miss ratio deduction(from FIFO) at different percentiles across all traces. A larger reduction is better. [1]

With respect to the proposed algorithms we made, there is a simple but interesting idea we decided to evaluate: what if the eviction of S3-FIFO was changed to be simpler to implement than what it currently is? What if instead of evicting based on FIFO it was done by randomly choosing an object? The answer to this last question is that in general it does not work very well... However, making a small adjustment to our ideas leads to reasonable comparative results. From our 3-member family of random algorithms, the first question was answered after the design of S3-RANDOM, which turned out to be a failure in terms of evaluation results. However, it was a success for seeding a new idea that gave birth to the third algorithm we proposed ; S3-FREQUENCIES gave competent results against S3-FIFO.The 3-member family of algorithms in an attempt to improve S3-FIFO will be explained in detail in the subsequent paragraphs but we want to remark here that the last one , S3-FREQUENCIES was the result of combining our first naive random idea with a logic that partially remembers the objects frequencies in the queues. These details consequently lead to the key results we obtained from S3-FREQUENCIES, which will all be illustrated from the simulations we performed as S3-FIFO did before.

3. BACKGROUND AND MOTIVATION:

The reader must be familiar with the basic concepts of a cache: what a cache is, the common terms used in cache topics (such as miss rate, cache block, queues, eviction policies) and a basic understanding of the common FIFO eviction algorithm, together with some of the high-level implementation details underlying it. Moreover, throughout the text. Caching allows you to efficiently reuse previously retrieved or computed data. The need for caching has become more pronounced with the advent of big data and real-time analytics. As data volumes grow and real-time processing speeds become a necessity, the ability to access and process data swiftly is critical. Without caching, systems would continually experience latency issues, leading to a subpar user

experience. Therefore, caching is not just a luxury, but a necessity in our current digital landscape. It's an essential component in the architecture of high-performance, scalable systems. We believe that although S3-FIFO provides an improved solution to the cache eviction landscape, it can still be improved. Specifically, we tried to keep its results and structure while changing some of its logic, with the intention of acquiring a similar algorithm in terms of efficiency but a simpler logic and less storage overhead in comparison, since we thought that this aspect is the main flaw that S3-FIFO presents.

4. Methodology

In this section, we are going to explain the methodology that we used for our project to evaluate and refine the S3FIFO cache eviction algorithm proposed in the research paper. Our approach involved on using the libCacheSim [2] simulator to compare the performance of different eviction strategies, with the aim of identifying different areas of improvement.

4.1. Experimentation Setup

This experimentation starts with the installation and configuration of the libCacheSim[2] simulator. This allows us to start the simulations of the cache under different workloads and eviction policies. Specially, we focused on comparing the performance of the following eviction algorithms: Random, RandomTwo and First-in-first-out (FIFO). These algorithms were chosen due to being simple and easy to implement.

4.2. Initial observations

From the Table 2, we observe that the eviction algorithms Random and RandomTwo eviction shows a lower miss rate compared to the FIFO on low cache sizes.

On the table 2 we can observe, that by increasing the cache size, the miss rate decreases even further, because, as we improve the capacity of the cache we can accommodate more data. Random eviction demonstrates a better efficiency than FIFO at handling unexpected access patterns due to its probabilistic eviction approach. Additionally, RandomTwo performs better than Random because it selects two randomly chosen objects and evicts the least frequent of the two objects, effectively maintaining the most access objects in the cache, and by consequence having better performances.

Cache Size	FIFO	RANDOM	RANDOMTWO
48	0.9121	0.9065	0.9034
146	0.8806	0.8709	0.868
489	0.8476	0.8444	0.8408
1469	0.8341	0.8321	0.8307
4897	0.8054	0.7991	0.8007
9794	0.7128	0.7292	0.7304
19589	0.6343	0.6288	0.632
39179	0.4316	0.4507	0.4557

Table2: Cache Size (bytes) VS Miss rate with OracleGeneral tracer with the eviction algorithms FIFO, Random and RandomTwo (results obtained from the libCacheSim[2] simulator)

Based on this observation, we propose the adoption of Random and RandomTwo algorithms over FIFO on the S3FIFO algorithm. Their advantage lies in their simplicity of implementation, as we don't need to track when the object was inserted on the cache. While RandomTwo incurs some overhead in frequency tracking, is still has less overhead than the FIFO queues.

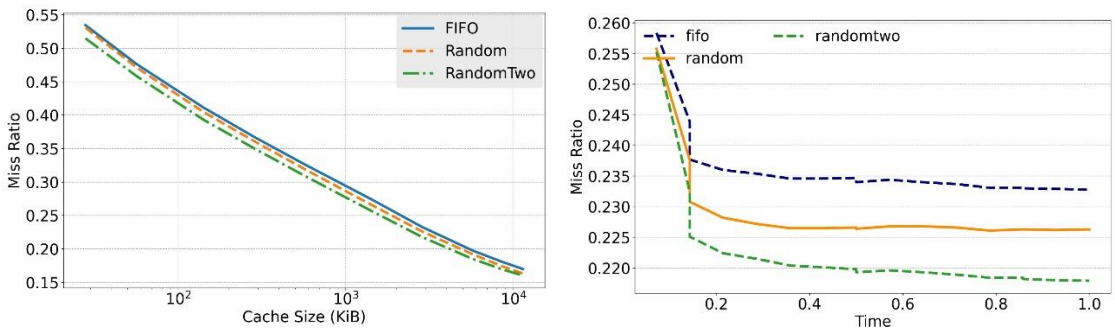
4.3. Evaluation Metrics

After obtaining the first observations of the algorithms random, we decided to do a performance evaluation of the algorithm to further analyse the trends of the eviction algorithms of Random and RandomTwo over FIFO.

From the graphs 1 and 2, we observed that the Random and RandomTwo perform much better compared than the FIFO eviction algorithm with bigger cache sizes and as time goes on.

On the graph1 we observe that all three algorithms perform similarly when changing the cache size, but we still see that as the cache grows bigger the lower miss rate we have, and we see that the best eviction algorithm of the three is RandomTwo, then Random and lastly FIFO.

On the graph2 we can see that over time the algorithm RandomTwo maintains a miss rate of around 0.220, the random with 0.225 and FIFO performing worse with a miss rate of 0.235.



Graph 1 and 2: Cache miss rate vs Cache size vs time with the twitter cluster 52, using a zipf workload using the eviction algorithms FIFO, Random and RandomTwo (graphs generated by the libCacheSim[2] simulator)

With these observations, we think that modifying the S3FIFO algorithm with the eviction algorithm Random and RandomTwo we can get better results as compared using FIFO because individually those eviction algorithms perform better than FIFO, and to implement them would be easier than FIFO as they are algorithms simpler than FIFO. We called this new modified algorithms S3Random and S3RandomTwo.

4.4. Analysis of the original S3-FIFO algorithm implementation

Before implementing the proposed algorithms, we decided to analyse how the S3-FIFO algorithm [1] was implemented on libCacheSim[2]. For this we accessed the simulator, and we navigated the different files until we found the eviction algorithm S3-FIFO.

We first observed that the implementations allow us to play the sizes of the small, main and ghost FIFO queues, but by default the small queue took 10% and the main 90% of the total size of the cache and the ghost queue that only store pointers occupies 90% of the cache. We observed that, they used a different version of FIFO where they combined the FIFO algorithm and 2-bit counter that counts the frequency of each object.

When accessing an object, it checks if is located on the small or the main queue, and if it exists we increase its frequency by one with a maximum of 3. If is not located on those queues they check the ghost queue, if we find it on the ghost queue we activate hit on ghost signal and the function returns as it was a miss, since we don't have the data. When we are going to insert we first check if is the hit on ghost signal is activated, if is activated then we insert on the main queue, else we insert it on the small queue.

When evicting, we first check if the small queue is full, if is full we evict on the small queue, else it means the main queue is full and we evict the main queue.

When evicting the small cache, we enter a loop that will iterate until either an object has been evicted or the small cache is empty. On the loop, we select the next victim, we then select the next victim to be evicted and check its frequency. If the frequency is bigger than the imposed threshold that by default is

Algorithm 1 S3-FIFO algorithm

Input: The requested object x, small FIFO queue $\mathcal{S}$, main FIFO queue $\mathcal{M}$, ghost FIFO queue $\mathcal{G}$

```
1: function READ(x)
2:     if x in S or x in M then                    ▷ Cache Hit
3:         x.freq ← min(x.freq + 1, 3)             ▷ Frequency is capped to 3
4:     else                                        ▷ Cache Miss
5:         insert(x)
6:         x.freq ← 0

7: function INSERT(x)
8:     while cache is full do
9:         evict()
10:    if x in G then
11:        insert x to head of M
12:    else
13:        insert x to head of S

14: function EVICT
15:    if S.size ≥ 0.1 · cache size then
16:        evictS()
17:    else
18:        evictM()

19: function EVICTS
20:    evicted ← false
21:    while not evicted and S.size > 0 do
22:        t ← tail of S
23:        if t.freq > 1 then
24:            insert t to M
25:            if M is full then
26:                evictM()
27:        else
28:            insert t to G
29:            evicted ← true
30:        remove t from S

31: function EVICTM
32:    evicted ← false
33:    while not evicted and M.size > 0 do
34:        t ← tail of M
35:        if t.freq > 0 then
36:            Insert t to head of M
37:            t.freq ← t.freq-1
38:        else
39:            remove t from M
40:            evicted ← true
```

algorithm1 [1]

1, we then evicted from the small queue and insert it on the main cache. If is not the case, we save its pointer on the ghost queue.

When evicting the main cache, we enter a loop that will be iterated until the main cache is full or an object has been evicted. As on the small cache, we select the next victim and check its frequency. If the frequency is 0 we evict it, else we subtract the frequency by one and we reinsert back to the main cache. And we repeat this until we evict an object from the main cache.

With this analysis, we came up with the idea of replacing the FIFO with the Random eviction algorithm and combining it with a 2-bit counter, and we decided to call it S3RandomFreq. From this modification we expect to perform better than S3FIFO[1], because individually the random eviction algorithm performs better as shown on the initial observations and the evaluation metrics and by combining it with a 2-bit counter we can filter object who are more frequently used from those who are not but also maintaining simplicity compared to S3FIFO, S3Random and S3RandomTwo.

5. Proposed Improvements and implementation

Based on the initial observations and evaluation metrics we observed that the Random and RandomTwo performed better and were simpler to implement than the FIFO algorithm, and we propose the S3Random and S3RandomTwo improvements over S3FIFO.

And from the analysis of S3FIFO we decided to combine the random algorithm with a 2-bit counter and we expect to perform better than S3FIFO,S3Randomand the S3RandomTwo with the cost of a little bit more complexity and overhead but still simpler than the S3FIFO.

5.1. S3Random and S3RandomTwo implementation

The idea behind S3Random is to have a simpler implementation than the original algorithm. This was achieved by removing the overhead that S3FIFO had; the 2-bit counter. And we also modified the eviction of the small and main cache. Being much simpler than S3FIFO, as on the eviction of the small queue we select the next victim, if that victim was previously hit we evict it from small and we promote it to the main cache, else we evict it from the small cache and we insert its pointer to the ghost queue. The main cache eviction is even simpler, we only select the next victim, and we evict it.

For S3RandomTwo the idea behind even though they have the same overhead of S3FIFO, we observed that individually the eviction scheme RandomTwo performs better than the FIFO and Random schemes. And as the implementation of S3RandomTwo it should be the same as S3Random. So, when explaining the S3Random it should be the same implementation as to S3RandomTwo.

To implement it on the simulator, we read the documentation of the libCacheSim[2] to add new algorithms and we had to program in c/c++. We first created our own init, we then created an instance of the cache of the simulator, and we declared the updated algorithms as shown on figures 4. After that we created our cache structure and declared the small, ghost and main queues and we used an already implemented cache eviction algorithm, for the case of S3Random we used the Random_init to create a Random eviction cache, and for S3RandomTwo we used RandomTwo_init to create a RandomTwo cache eviction scheme , and, we change its parameters corresponding to our desired structure, 90% for the main cache, 10% for the small cache and 90% for the main cache, as shown on figure 5.

```c
cache_t *S3Random_init(const common_cache_params_t ccache_params,
                       const char *cache_specific_params) {
    //We define creqte the cache structure init
    cache_t *cache =
      cache_struct_init("S3Random", ccache_params, cache_specific_params);

    //We define the new functions of the cache with the new structure
    cache->cache_init = S3Random_init;
    cache->cache_free = S3Random_free;
    cache->get = S3Random_get;
    cache->find = S3Random_find;
    cache->insert = S3Random_insert;
    cache->evict = S3Random_evict;
    cache->remove = S3Random_remove;
    cache->to_evict = S3Random_to_evict;
    cache->get_n_obj = S3Random_get_n_obj;
    cache->get_occupied_byte = S3Random_get_occupied_byte;
    cache->can_insert = S3Random_can_insert;
```

```c
//we create the caches
common_cache_params_t local_cache_param = ccache_params;
local_cache_param.cache_size = small_size;
//create small cache
params->small_random = Random_init(local_cache_param, NULL);
//create ghost cache
local_cache_param.cache_size= ghost_cache_size;
params->ghost_random=Random_init(local_cache_param,NULL);
//create main cache
local_cache_param.cache_size = main_cache_size;
params->main_random=Random_init(local_cache_param,NULL);
```

Figure 4 and 5: They show the implementation of the init of S3Random

The eviction algorithm is the same as S3FIFO, but we changed how the small and main cache evicts.

As observed on the figure 6, the small cache behaves on the same way as on S3FIFO, but with the difference that we don't adopt the while loop and the use of frequency's, becoming a simpler algorithm. However, if we use the RandomTwo eviction scheme even though we have the same overhead with frequencies we still have a simpler implementation on the small cache as we do less operations compared to S3FIFO.

```c
static void S3Random_evict_small(cache_t *cache, const request_t *req) {

    S3Random2_params_t *params = (S3Random2_params_t *)cache->eviction_params;
    //We define the caches to avoid redundancy
    cache_t *small= params->small_random;
    cache_t *main=params->main_random;
    cache_t *ghost=params->ghost_random;

    //We evict the small cache only if the occupied bytes is bigger than 0
    if ( small->get_occupied_byte(small) > 0) {
        // evict from small cache
        cache_obj_t *obj_to_evict =small->to_evict(small,req);

        //we check that there is no empty obj to be evicted
        DEBUG_ASSERT(obj_to_evict != NULL);

        // need to copy the object before it is evicted so that it can be insert it to main if need it
        copy_cache_obj_to_request(params->req_local, obj_to_evict);
```

```c
        //If object has promoted == true then we promote it to main
        if (obj_to_evict->S3Random.promoted == true) {
            // Update statistics
            params->n_obj_move_to_main += 1;
            params->n_byte_move_to_main += obj_to_evict->obj_size;

            //insert it to main
            cache_obj_t *new_obj = main->insert(main, params->req_local);

        }
        // The obj doesn't have promotion activated so we evict it and save the
        //pointer on the ghost cache
        else {
            ghost->get(ghost, params->req_local);
        }

    // remove from small cache, but do not update stat
    bool removed = small->remove(small, params->req_local->obj_id);
    assert(removed);
```

Figure 6: It shows the implementation of the eviction algorithm of the small cache

For the implementation of the main cache eviction is much more simple, as we only evict the next victim as shown on figure 7.

```
static void S3Random_evict_main(cache_t *cache, const request_t *req) {
    S3Random2_params_t *params = (S3Random2_params_t *)cache->eviction_params;
    //We define the caches to avoid redundancy
    cache_t *main=params->main_random;

    // evict from main cache
    //we only evict if the occupied space is bigger than 0
    if ( main->get_occupied_byte(main) > 0) {
        //we evict from main

        cache_obj_t *obj_to_evict = main->to_evict(main, req);
        //We check if we evicted the object
        DEBUG_ASSERT(obj_to_evict != NULL);



        //We need to do this to create a request to remove the object from the queue
        copy_cache_obj_to_request(params->req_local, obj_to_evict);


        // we remove the object to be evicted
        bool removed = main->remove(main, obj_to_evict->obj_id);
        // if we cannot remove it then we raise a personalized error
        if (!removed) {
          ERROR("cannot remove obj %ld\n", (long)obj_to_evict->obj_id);
        }
    }
}
```

Figure 7: The implementation of the main cache eviction of S3Random and S3RandomTwo

5.2. S3RandomFreq

The idea of this algorithm is to use the same implementation of S3FIFO including its 2-bit counter but changing the eviction scheme from FIFO to the Random algorithm because we observed that the Random has shown better results comparing it to the FIFO, and the Random cache requires less complexity than FIFO.

When implementing it on the simulator we did the same as with S3Random and S3RandomTwo, we defined the init with our own functions, and defined the cache sizes as shown on figure 4 and 5. And for the eviction of the small cache we only changed from figure 6, is the if instead of looking for a hit we look the frequency of the next victim is bigger than 1.

For the main evict the things change as now we implement a loop, where we take the next victim, we obtain its frequency. If the frequency is 0 we evict it from the cache else we subtract by 1 and we save the metadata of that object and we go to the next iteration until we evict an object. The implementation of this eviction can bee seen on the figure 8. Even though this algorithm looks the same as S3FIFO main eviction, is still a little bit simpler as we don't require to reinsert it on the main cache after selecting the next victim.

```
static void S3Randomfreq_evict_main(cache_t *cache, const request_t *req) {
    S3Randomfreq_params_t *params = (S3Randomfreq_params_t *)cache->eviction_params;
    //We define the caches to avoid redundancy
    cache_t *main=params->main_random;

    // evict from main cache
    //we only evict if the occupied space is bigger than 0
    bool evicted=false;
    while (!evicted && main->get_occupied_byte(main) > 0) {
        //we evict from main

        cache_obj_t *obj_to_evict = main->to_evict(main, req);
        //We check if we evicted the object
        DEBUG_ASSERT(obj_to_evict != NULL);
        int freq =obj_to_evict->S3Randomfreq.freq;


        //We need to do this to create a request to remove the object from the queue
        copy_cache_obj_to_request(params->req_local, obj_to_evict);

        if ( freq >0){
            //main->remove(main,obj_to_evict->obj_id);
            //obj_to_evict = NULL;
            obj_to_evict->S3Randomfreq.freq= MIN(freq,3)-1;
            obj_to_evict->misc.freq=freq;

        }else{

            // we remove the object to be evicted
            bool removed = main->remove(main, obj_to_evict->obj_id);
            // if we cannot remove it then we raise a personalized error
            if (!removed) {
            ERROR("cannot remove obj %ld\n", (long)obj_to_evict->obj_id);
            }
            evicted=true;
        }
    }
}
```

Figure 8: Implementation of the main cache eviction function.

6. Evaluation of the proposed Improvements

After implementing the algorithms on the libCacheSim[2] simulator, we had to modify some parts of the code on the simulator so we can do the build with our proposed eviction schemes. The parts we modified were the Cmakelist.txt on the cache eviction folder where we included the path to our proposed improvements. Then we modified the cacheObj.h file so we can define our own metadata, that were used to either store the information about promotion or the frequencies depending on the scheme. We also added some tests to make sure our algorithm worked as intended and finally we modified cache_init.h to recognize our own implementation and accept it and run the simulation on them. After changing those settings, we did the build again and we evaluated our proposed improvements over other algorithms and S3FIFO.

6.1. Evaluation of the proposed improvements vs FIFO vs Random vs RandomTwo vs S3FIFO.

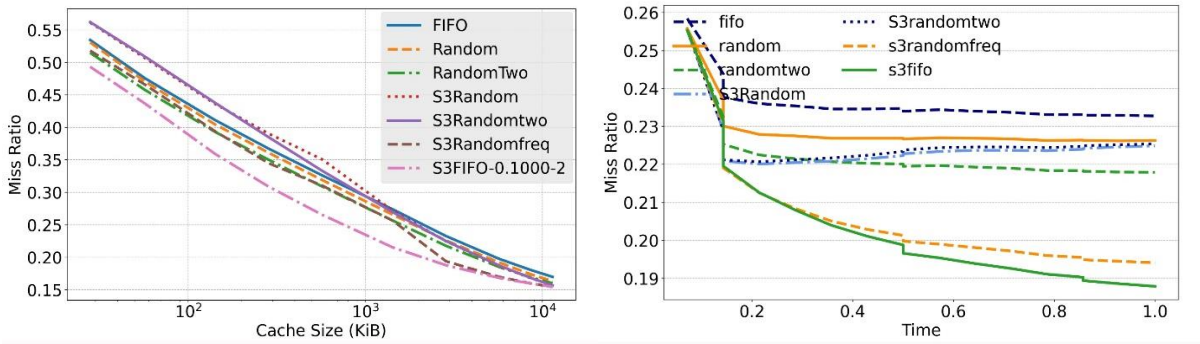
After doing the performance evaluation of our proposed improvements we obtained the graphs 3 and 4.

On the graph 3 we observe the miss rates vs the cache sizes. On this graph, we see that our proposed eviction schemes didn't perform better from the baseline paper, S3FIFO. And the proposed schemes were performing similarly as the algorithms FIFO, Random and RandomTwo. So,

in regards of the sizes our algorithms don't improve, so we can conclude that our improvements are not as scalable as S3FIFO, as this eviction scheme performs much better.

On the graph 4 shows a graph where it contrasts the miss rate over time in hours. In this graph we learn that S3Random and S3RandomTwo, perform better than the FIFO and Random, however they perform worse than RandomTwo algorithm after 0.3 hours of the simulation. Both schemes they have a tendency as times passes to have a miss rate like the Random eviction scheme. With this conclude that S3Random and S3RandomTwo are not good eviction algorithms as they don't improve and even perform worse than simple algorithms. The reason behind these results is that we are trying to use a complex structure with a very simple algorithm, and we don't obtain the expected results.

However, on graph4, we see that S3RandomFreq doesn't perform as good as S3FIFO, but they get very closed results comparing it with all the other algorithms. This happens thanks to the implementation of the 2-bit counter. With this we conclude that by the implementation of metadata to each object of 2 bits we improve significantly.



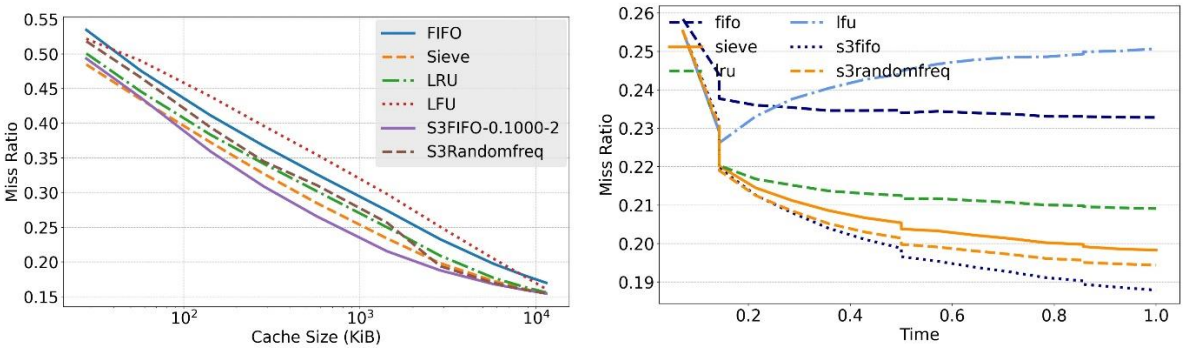
Graphs 3 and 4: Cache miss rate vs Cache size vs time with the twitter cluster 52, using a zipf workload using the eviction algorithms FIFO, Random and RandomTwo, S3Random,S3randomTwo , S3RandomFreq and S3FIFO (graphs obtained from the simulator libCacheSim[2])

6.2. Evaluation of S3RandomFreq vs other popular eviction algorithms.

From the observed results obtained on the previous section, we decided to evaluate the S3RandomFreq with other popular eviction algorithm such as fifo, sieve [3], lfu, lru.

On graph 5, we observe that our implemented S3RandomFreq algorithm doesn't scale in terms of cache size as good as other algorithms such as LRU and Sieve[3]. But is also important to say that compared to these 2 algorithms, S3RandomFreq is easier to implement and has less metadata.

On the graph 6, we observe that our designs perform much better over time compared to the other schemes and it tends to stay stable around a 0.195 miss rate. With these results can be a good reason to choose this algorithm compared to other schemes. However, S3FIFO outperforms S3RandomFreq in both cache sizes and time, with the expense of a little bit more complexity on the eviction functions.



Graph 5 and 6: : Cache miss rate vs Cache size vs time with the twitter cluster 52, using a zipf workload using the eviction algorithms FIFO, Sieve, LRU, LFU, S3RandomFreq and S3Random. (graphs generated by the libCacheSim[2] simulator)

7. Conclusion

The design of more efficient cache eviction algorithms is crucial in modern computing and for that reason we were motivated to do research on some of the latest improvements. After the successful and innovative S3FIFO algorithm, we found that although it excelled in terms of efficacy, its implementation complexity could perhaps be beaten. Consequently, we developed 3 algorithms, which aimed to preserve a relatively similar accuracy as to S3FIFO but to become simpler and more affordable in terms of storage costs and complexity. All of them have the same fundamental structure as S3FIFO but use randomness to simplify decision making. Although complete randomness was proven to be unsuccessful, our last design performance resulted in arguably comparable results as S3FIFO. Also, aside from the analysis results of the algorithms, we would also like to point out that we significantly expanded our knowledge on caches and strengthened earlier notions learned throughout the course, which is nevertheless also of great value from our point of view. More generally, we are glad to have been given the time opportunity to get outside of the conventional learning environment and looked at the field of research on a computer architecture subject, which not only provided insight but also solidified in our minds the fact that everything is human made and one should always think critically and as mentioned in the first day of the course: the role of the architect is to look backward (examine old code), look forward (listen to dreamers) , look up (nature of the problems) and look down (predict the future of technology).

References

[1] FIFO Queues are All You Need for Cache Eviction https://jasony.me/publication/sosp23-s3fifo.pdf

[2] LibCacheSim a high-performance simulator https://github.com/1a1a11a/libCacheSim

[3] Sieve is simple than LRU : https://junchengyang.com/publication/nsdi24-SIEVE.pdf