EE5364 Project Report,
Addressing Aging Set Issues with LFU Cache Replacement

Jack Hanson; hans5075
Tianming Cui, cuixx327

December 12, 2019

1 Abstract

As server based computing becomes more prevalent, it's worth examining the memory access patterns typical of server workloads and optimizing cache performance to reflect these patterns. One existing cache replacement policy, which fails to address changes to the working set (which is typical of server workloads) is LFU.

In this paper, we introduce LFU Replacement with Consecutive Miss Counter (CMCLFU), which is a low cost implementation designed to address the aging set issues of LFU replacement policy.

Our simulation results show that the CMCLFU can reduce the LLC MPKI by about 11% on gcc benchmark, which is a significant improvement. It achieves this while having comparable performance to existing replacement policies on other tested benchmarks.

We also introduced an alternative solution termed as LFU Replacement with Block Expiration. This policy should have even lower MPKI than CMCLFU, but will also introduce a higher overhead.

2 Introduction

Choice of replacement policy is an important part of the associative cache system. With an optimal replacement policy (usually a policy fit for the corresponding workload) the cache system will be more likely to retain the data which will be referenced again in the immediate future, and choose the data which will not be referenced again as the victim. Choosing the ideal replacement policy is a balance between cost and performance; a policy that slightly reduces average miss rate, but incurs significant overhead, is not likely to be implemented.

LFU (Least Frequently Used) replacement policy is a common replacement policy. With the assumption that the least frequently used data will also not likely be referenced in the near future, LFU records the number of references to each block and picks the block with the lowest counter to be the victim. The word LFU is a misleading term, in that the original LFU design only tracks the number of references instead of the real frequency of references for each block. LFU uses reference number to simulate the reference frequency since the cost of tracking the real (time-weighted) frequency would be unacceptable high: it would require extra hardware to record the age of the data, and would cause computational and memory-access overhead every time it tried to compute the access frequency of a block from its reference counter and its age. Tracking the reference number instead can greatly reduce the overhead, but it leads to another issue: since LFU does not reduce/reset the number of reference in stagnated sets, over time, the cache will be bloated by older blocks which have high number of references (even if they have not been referenced recently), and the new blocks are likely to be kicked out before they accumulate a high reference counter. It is called "the aging issue of LFU".

A toy example here shows how this aging issue will influence the cache performance:

```
//Simple Toy Example

for (int i = 0; i<10000; ++i){
    refer A[1];
    refer A[2];
    refer A[3];
    refer A[4];
}

for (int i = 0; i<1000; ++i){
    refer B[1];
    refer B[2];
    refer B[3];
    refer B[4];
}
```

Figure 1: Toy Exapmle Shows the Aging Issue

In this example, assume that the cache is 4-way set associative, and that (for the sake of example) each block can only hold one int. Also assume that each reference maps to the same set. Since each A[X] is referenced 10000 times, they are not likely to be chosen as the victim. So the actual progress of LFU policy for the second FOR loop is:

- Refer B[1]: kick A[1] out, insert B[1], B[1] counter equals to 1;
- Refer B[2]: since B[1] has the lowest counter, kick B[1] out, insert B[2], B[2] counter equals to 1;
- Refer B[3]: since B[2] has the lowest counter, kick B[2] out, insert B[3], B[3] counter equals to 1;
- Refer B[4]: since B[3] has the lowest counter, kick B[3] out, insert B[3], B[3] counter equals to 1;
- Refer B[1]: since B[4] has the lowest counter, kick B[4] out, insert B[1], B[1] counter equals to 1;
- This pattern continues

Figure 2: Behavior of LFU on the Toy Example

From this simple example, we observe that 3/4 of the cache memory is polluted by the aged data and a total of 1004 cache misses occur. It's apparent that in certain situations, the aging set issue will significantly reduce the performance of the cache system.

3 Background and Related work

Former researchers have tried to solve this aging issue. Among their designs, Frequency Based Replacement (FBR), Least Frequent Recently Used (LFRU) and LFU with Dynamic Aging (LFUDA) are relatively popular ones.

LFU with Dynamic Aging (LFUDA) solves the aging issue of LFU by introducing a real "aging" factor into the traditional LFU policy. While LFU is actually picking the least used data (in total) as the victim, LFUDA is picking the least

frequently (time-weighted) used data. This design adds an aging factor to each block so the system can track the real frequency of the data instead of using reference counter to approximate it. LFUDA can completely solve the aging issue of LFU, but the cost and overhead of LFUDA is extremely high, and that is the reason why traditional LFU does not introduce the aging factor.

Least frequent recently used (LFRU) replacement policy designed a new cache architecture. LFRU divides the entire cache into two individual partitions called privileged and unprivileged partitions. The frequently used data (with high reference counter) will be pushed into the privileged partition. When replacement happens, LFRU moves a block from the privileged partition into the unprivileged partition, removes one block from unprivileged partition and inserts the new block into the privileged partition. By applying an approximated LFU policy on the unprivileged partition, and LRU on the privileged partition, LFRU is able to filter out the aged data. [3]

The FBR design shares a similar idea like LFRU, but with a more complex design. FBR is a hybrid replacement policy which tries to combine the benefits of LRU and LFU. It divides the entire cache into three partitions, these three partitions are ordered from most recently used to least recently used. FBR picks the block with the lowest reference count as the victim like LFU, but only picks the victim from the least recently used partition. This implementation can solve the aging issue by pushing the aged blocks into the least recently used partition and finally pick them as the victim. [1]

LFRU and FBR policies can correctly solve the aging issue of traditional LFU policy by dividing the entire cache into several partitions and using different policies on different partitions. But since these implementations need to partition the cache and move blocks among those partitions, there will be significant memory overhead.

4 Implementation

4.1 LFU Replacement with Consecutive Miss Counter

As mentioned in the introduction part of this report, the cost (or overhead) of a replacement policy is not ignorable. So we will first introduce LFU Replacement with Consecutive Miss Counter (CMCLFU) in this paper, which has a relatively low hardware and computational cost. CMCLFU is designed to have similar behaviour to LFRU, but with lower cost and overhead, since it dynamically modifies its victim selection method based on cache statistics instead of the cache partition which is implemented by LFRU.

CMCLFU tracks the overall performance of the entire cache system with a global miss counter. It assumes that the overall performance of a cache system can be partly represented by the number of consecutive misses that have occurred. If the memory system observes a high number of consecutive misses, then it means the LFU replacement policy is not working well: it tends to pick non-optimal victims so that data which may still be useful in the future cannot be correctly held in the memory.

Based on this idea, we can dynamically adjust the replacement policy to make it fit for complex workloads. When the consecutive miss counter is low (which suggests traditional LFU currently has an acceptable performance), CMCLFU has same behavior as traditional LFU. When it detects the consecutive miss counter exceeds a certain threshold (termed as RandomRPThreshold), this suggests that traditional LFU is experiencing a performance decrease that might caused by the aging issue. In this case CMCLFU begins to behave like Random replacement policy and makes it possible to evict aged cache blocks. If the consecutive miss counter continues to increase and reaches a higher threshold (termed as FlushThreshold, fixed as 1024 in our implementation), CMCLFU will flush all data out and restart completely. We designed the RestartThreshold to be very high and usually cannot be reached with a common workload, but if the system is switching from one work-set into another (e.g. a server finishes responding to one client and starts to respond to another client), since all the aged data from the old work-set is not likely to be referenced again, the consecutive miss counter can be increased very quickly and will potentially reach the RestartThreshold. During this kind of situation, the algorithm flushes all aged data and restarts the entire cache in order to help the system quickly adapt to the new work-set.

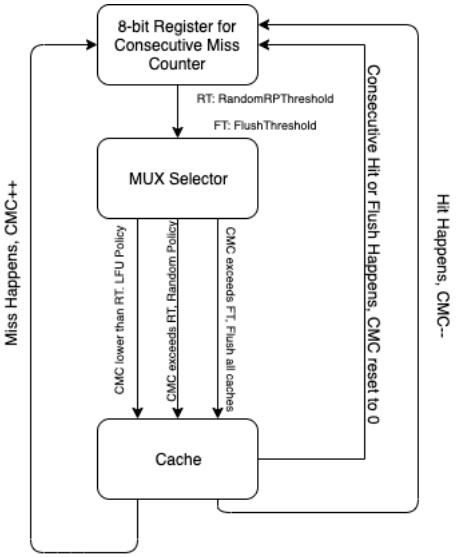

Figure 3: Design Diagram of CMCLFU

This implementation introduces almost no computational and hardware overhead into the system, only an 8-bit register for storing the consecutive miss counter and a MUX for picking corresponding replacement policy is needed.

4.2 LFU Replacement with Block Expiration

One way to address the issue of aging cache sets is to force blocks within the set to be accessed semi-regularly in order to preserve their place in the set. The majority of evictions will be determined by the traditional LFU replacement policy, with the exception that blocks which aren't accessed within a predefined period will be considered "expired", and will be evicted regardless of the value of their LFU counter.

This prevents blocks which are no longer in high demand from polluting the set, simply because they've accumulated a high LFU counter in the past. This allows newer blocks (which have been accessed more recently than the expired blocks) to remain in the set longer, giving them the chance to build up their own LFU counters.

Under LFU Replacement with Block Expiration, each block has an LFU counter, which serves the same purpose as the counter for traditional LFU replacement. Each set in the cache has its own bit field, the "tracking field", the purpose of which will be explained shortly. Each set also exists within one of two states: the Tracking State and the Expiration State. When a block is inserted into a set, the algorithm used to determine the victim changes depending on the state of the set.

Tracking State: All sets start out in the Tracking State, with each bit in their associated tracking field set to zero. The set will remain in the Tracking State for a period of N-accesses, after which the set will transition to the Expiration State. Each block in the set corresponds to a bit in the tracking field, with a value of zero indicating that block hasn't been accessed since the start of the current period (that is, since the set entered the Tracking State). When a block is accessed, its corresponding bit in the tracking field is set to one, and its LFU counter is incremented. All evictions are determined based on traditional LRU eviction, that is, the block with the lowest LFU counter is always chosen for eviction.

Expiration State: At the end of a period (a series of N-accesses while in the Tracking State), a set transitions to the Expiration State. If there are still zero-bits in the tracking field for that set, the next eviction will be a block corresponding to one of these zero-bits (after which the bit for that block is set to one). This continues until all bits in the tracking field have been set to one, at which point the set transitions back to the Tracking State, and the tracking field is reset to all zeros.

State Diagram:

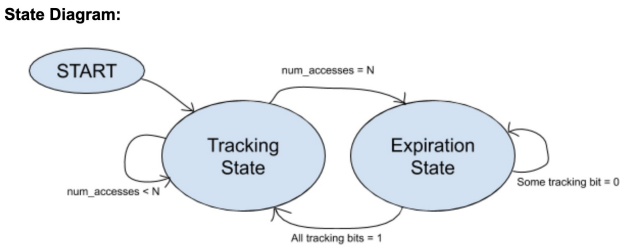

Figure 4: State Diagram of LFU with Block Expiration

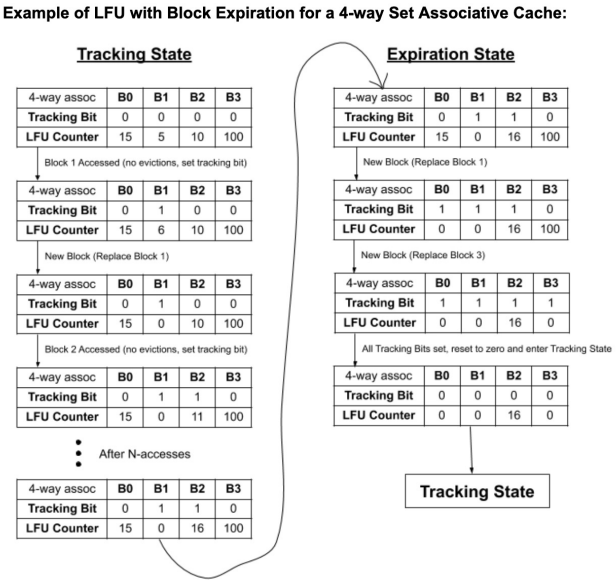

Figure 5: Example of LFU with Block Expiration

Issues with Implementation in gem5 After experiencing extensive issues with implementing our custom replacement policies, we eventually discovered that the actual implementation of the gem5 simulator has heavily diverged from its official documentation, which hasn't been updated in the past 20 months, despite undergoing extensive changes during this time. It seems that around March/April of 2018, the interface for implementing custom replacement policies was completely revamped, while around the same time updates ceased for the (doxygen based) gem5.org/docs/ site. The current implementation (hosted on github) is not only undocumented, but also heavily contradicted by the past source code hosted on gem5.org.

Using the (unbeknownst to us) depreciated doxygen site as a reference, we implemented our replacement policies in C++. Under the former gem5 implementation, this was achieved by inheriting from the AbstractReplacementPolicy class. In both past and current iterations of gem5, the main functionality of custom replacement algorithms is achieved by overriding the methods touch() and getVictim(). With AbstractReplacementPolicy, the prototype for these methods is as follows:

```
/* touch a block. a.k.a. update timestamp */
virtual void touch(int64_t set, int64_t way, Tick time) = 0;

/* returns the way to replace */
virtual int64_t getVictim(int64_t set) const = 0;
```

In this implementation, the programmer is responsible for allocating and maintaining the data structures used to store metadata for each cache set/block. The benefit is that this interface allows for a lot more control, with these functions specifying exactly which set/block is being updated. Compare this to the new interface (whose base class is now called BaseReplacementPolicy) where touch() only supplies the user

with metadata about the block being accessed:

```
virtual void touch(const std::shared_ptr<ReplacementData>&
                                        replacement_data) const = 0;
```

While this removes a lot of burden from the programmer in terms of managing data structures used to store block metadata, it completely removes the ability to update information about the set being accessed. Even if the user were inclined to maintain data about the set themselves, they wouldn't know which set a block being "touched" belonged to, nor would they have information about the cache as a whole, such as set associativity (formerly given by the data-member "m_assoc") or the number of sets (formerly "m_num_sets").

After realizing this mistake, we attempted to modify our replacement algorithms to comply with the new interface. For our LFU Replacement with Consecutive Miss Counter algorithm, this process was relatively straightforward, as the LFU counter information can be included in the new ReplacementData class, and the miss counter can be implemented as global data member, which doesn't need set-specific information to be updated/accessed.

LFU Replacement with Block Expiration, on the other hand, was harder to adapt to the new interface. The LFU counters and expiration bits (now implemented as individual booleans rather that a bit field) could be encapsulated by the ReplacementData class. However, the new system doesn't provide a straightforward means of implementing the finite-state behavior for sets. Specifically, it would be challenging to implement and update the per-set counter used to determine when a period has ended, triggering the transition from the Tracking State to the Expiration State.

Of course, implementing LFU with Block Expiration isn't impossible under the new gem5 implementation. The algorithm could be modified so that the condition for transitioning between states were more conducive under the new interface. Alternatively, it would be possible to implement our existing design by modifying the simulator itself, given that gem5 is open source. In the future, it would be worth exploring these options so that we could test the efficacy of our algorithm.

So at the current stage, we will focus more on the simulation of CMCLFU.

4.3 Algorithm Parameters

For LFU with Block Expiration, a set remains in the Tracking State for N accesses to that set before transitioning to the Expiration State. This is an important detail in our implementation, as our choice of period length could greatly affect the performance of our replacement algorithm. Setting this parameter too low could inhibit performance by causing blocks to expire prematurely. Intuitively, it seems that the period length should be well above the associativity of the cache. It's worth experimenting with different period lengths to determine the optimal value for this parameter.

For LFU with Consecutive Miss Counter, an important parameter is the number of consecutive misses that trigger random replacement, the RandomRPThreshold. Higher values of this threshold will cause the algorithm to behave more similar to LFU; lower values will cause the algorithm to behave more similar to Random Replacement. For our experiment, we will be testing different values for this threshold. We will refer to LFU with Consecutive Miss counter as CMCLFU-64 in our results when the threshold is set to 64; similarly CMCLFU-X refers to a threshold of X. The actual influence of this parameter will be discussed in the performance results section.

5 Performance Results

As mentioned in the introduction, the traditional LFU will meet a significant performance reduction on the toy workload where the aging issue is heavy.

The CMCLFU with RandomRPThreshold of 4 can obviously overperform traditional LFU on the toy workload(shown in introduction section) with only 12 cache misses.

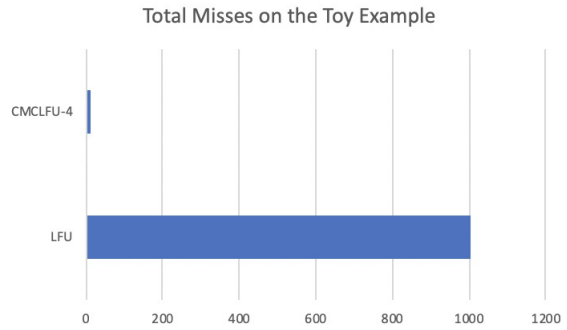

Figure 6: Total Misses on the Toy Example

But this toy workload is a perfect testbench designed for the CMCLFU implementation, therefore cannot represent the actual performance of our implementation.

After implementing the CMCLFU design with the gem5 simulator, we are able to run common testbenches with our implementation and compare it to other solutions.

The CMCLFU is designed for LLC replacement, so our implementation is based on the L2 cache controller of the MESI two level structure (for L1 cache, it is likely to have similar performance on the L1 instruction miss rate). As mentioned in the Algorithm Parameters section, CMCLFU-X refers to the CMCLFU policy with a RandomRPThreshold of X.
The tests are done with following configures:
–ruby –caches –l2cache –l1d_assoc=8 –l2_assoc=16 –l1i_assoc=4

–cpu-type=TimingSimpleCPU –num-cpus=1
–mem-type=SimpleMemory
–maxinsts=250000000 -F 1000000000
–mem-size=4GB –network=simple –topology=Mesh_XY
–mesh-rows=1 –num-dirs=1 –cacheline_size=64

The following figure shows the L2 MPKI of corresponding replacement policies (the lower the better):

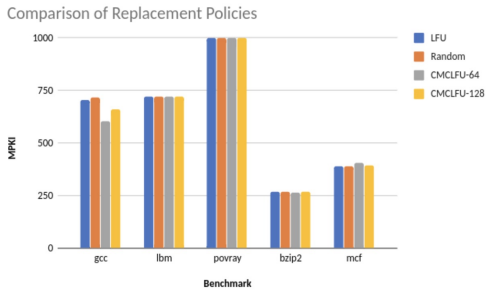

Figure 7: L2 MPKI of replacement policies

The following figure shows the cycles per instruction (CPI) of corresponding replacement policies, the lower the better:

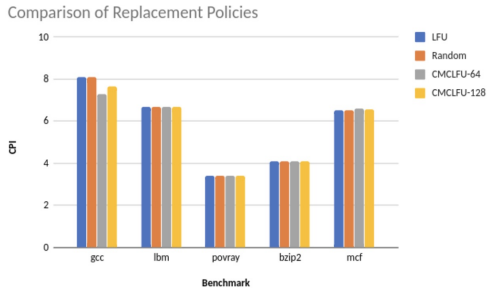

Figure 8: CPI of replacement policies

The simulation results meet our expectations: on those worksets where the aging issue is heavy, CMCLFU can outperform traditional LFU with lower CPI and MPKI. For most of the worksets where aging issue is not serious, the performance of CMCLFU is close to LFU (without incurring significant overhead). On some particular worksets like mcf, CMCLFU performs a little bit worse than traditional LFU but is still acceptable.

Picking the RandomRPThreshold for CMCLFU is a tradeoff for performance on different workloads. The following figures will show the L2 MPKI and the CPI of a CMCLFU based cache system with different RandomRPThresholds.

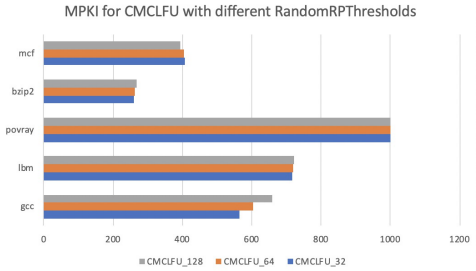

Figure 9: L2 MPKI for different thresholds

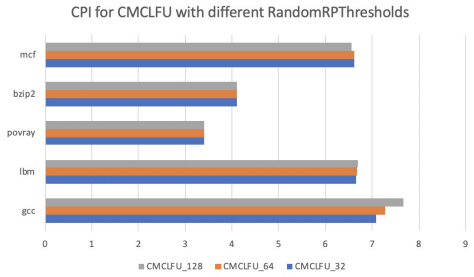

Figure 10: CPI for different thresholds

A lower RandomRPThreshold will make CMCLFU become more aggressive (since it will be more easily triggered), trying to solve the aging issue sooner, so it will work better on the workloads where the aging issue is significant while performing worse on the workloads where traditional LFU has a good performance. Based on the simulation results shown above, we recommend setting RandomRPThreshold as 64 to obtain a balanced performance.

When analyzing the simulation result in detail, we found that when testing gcc, the consecutive miss counter is relatively high compared to other testbenches, which may suggest the aging issue on gcc testbench is heavier than other testbenches. This can also explain why CMCLFU has significantly better performance than LFU on the gcc benchmark

6 Conclusion

With the emergence of cloud computing and other web based technologies, it becomes increasingly desirable to optimize cache performance for typical server workloads.[2] The issue of aging sets is particularly prevalent for servers, where the working set may be constantly evolving.

We have devised two novel replacement policies that expand upon traditional LFU, a replacement algorithm that completely fails to combat the issue of aging sets. Both of these algorithms combat this issue, with less overhead that existing solutions, such as LFU with Dynamic Aging. LFU with Consecutive Miss Counter in particular incurs negligible memory overhead compared to traditional LFU, and only requires the addition of one global counter used to track consecutive cache misses.

Our results show that for most benchmarks, the performance of LFU with Consecutive Miss Counter is comparable to LFU and Random Replacement, with minimal overhead. In the gcc benchmark, CMCLFU performs significantly better than these existing policies, which we believe reflects the aging set issue in the gcc memory access pattern.

We designed the LFU with Block Expiration as an alternative solution to address the aging issue in a more robust way than CMCLFU. Due to inconsistencies between the implementation of gem5 and its documentation, we were unable to implement LFU with Block Expiration for simulation. In the future we would be interested in evaluating the performance

of this algorithm in comparison to that of existing replacement policies.

References

[1] Muhammad; et al. Bilal. A cache management scheme for efficient content eviction and replication in cache networks. *IEEE Access. 5*, 2017.

[2] J Hajiakhondi-Meybodi, Z.; Abouei. Cache replacement schemes based on adaptive time window for video on demand services in femtocell networks. 2017.

[3] T Jayarekha, P.; Nair. An adaptive dynamic replacement approach for a multicast based popularity aware prefix cache memory system. 2010.