EC513 Computer Architecture
Project Report

Comparison of Cache Replacement Policies using Gem5 Simulator

Team Members

Hafsah Shahzad
Farhan Tanvir Khan
Farbin Fayza

1 Abstract

As modern processors include multiple levels of hierarchy and as the cache associativity increases, the importance of selecting the optimal cache replacement policy also rises. Different cache replacement policies evict differently based on the data items' recency, reference rate, frequency and size. In this project, we have implemented three cache replacement policies from academic publications and compared them against the common policies already implemented inside gem5. Our results demonstrate that the performance of a policy depends strongly on the application, complexity, and cache configurations.

2 Introduction

As the gap between processor and memory speed increases, the role of memory hierarchy is becoming increasingly crucial. Caches improve the efficiency of data retrieval by reducing memory latency and hence memory traffic. Its effectiveness is generally constrained in terms of the hit rate: ratio of memory requests serviced by the cache to the total memory requests made. This raises another question: "When a new line is to be inserted into the cache, which line must be evicted to make space for the new line?"

Cache Replacement Policies answer precisely this. An optimal replacement policy tries to replace a cache line whose next reference is furthest in the future. However, having a perfect knowledge of future cache references and their implementation is virtually impossible. Instead, several cache replacement policies employ different heuristics to determine the cache line most suitable for eviction.

In this project, we have implemented three cache replacement policies and evaluated their performance with already implemented replacement policies in gem5. The analysis is made using parsec benchmarks and metrics such as instructions per cycle and miss rate for varying sizes of L1 and L2 caches.

3 Replacement Policies' Description

3.1 Least Frequent Recently Used (LFRU)

Least Frequently Recently Used cache replacement policy combines the benefits of both Least Frequently Used (LFU) and Least Recently Used (LRU) replacement policies. It does so by approximating the LFU, calling it ALFU and a smaller privileged section running LRU.Contents are evicted from the ALFU partition based on their popularity, but it is brought down if the item is not referred to within a certain time window (Wt). If the content is highly popular, it is pushed into the privileged partition. Replacement of the privileged partition is achieved as follows: LFRU evicts content from the unprivileged partition, pushes content from privileged partition to unprivileged partition, and finally inserts new content into the privileged partition.

3.2 LFU with Dynamic Aging (LFUDA)

Least Frequently Used (LFU) policies evict items based on its overall usage i.e. items that are less popular in the long run[**?**]. However, LFU policies have limitations. In some cases, if an item has a popularity boost and then is no longer used anymore, LFU would still rate it high based on its spurt. The cache is said to be populated. LFU with Aging incorporates the idea of aging to circumvent this. However, these generally have higher management overhead since they require tuning parameters in order to operate well. LFU with Dynamic Aging (LFUDA), replaces the tunable aging with a dynamic mechanism that is parameter-less and hence less complex and easier to manage. In LFUDA, an item's priority key is set to that item's frequency plus the cache age (L). In this way, through Dynamic Aging, the priority key of items that were very popular previously but now unpopular, is reduced relative to the rest of the cache thus eventually making it eligible for eviction.

3.3 Adaptive Replacement Cache (ARC)

The motivation of this policy is to have a cache replacement system that can adapt to the dynamically evolving data access patterns of real-life workloads. It is implemented with two Least Recently Used (LRU) lists of blocks, each of capacity c. The first list $L1$ is used to store the recently used blocks, whereas the second list $L2$ tracks the frequently used blocks. That is, if a block is accessed more than once, it is moved from $L1$ to $L2$, because it has an access frequency of at least two. ARC attempts to maintain a target number of Most Recently Used (MRU) blocks in $L1$ and $L2$. Precisely, it tries to have p number of MRU blocks in L1 and $c - p$ number of MRU blocks in $L2$. The speciality of ARC is that it tunes the parameter p based on the history of access patterns. It learns if a workload is prioritizing recency or frequency on accessing data and maintains the sizes of $L1$ and $L2$ accordingly by changing p. This allows ARC to respond dynamically to the variations of the access patterns of the workloads.

4 Implementation

LFRU, LFUDA and ARC are implemented by Farhan, Hafsah, and Farbin respectively. The source codes of LFUDA are kept in a separate branch named "Hafsah_Project_Implementation" because it has changes in some config files that hamper the other two replacement policies. All the outputs and the source codes of LFRU and ARC are kept in the master branch.

4.1 LFRU

The implementation of the policy was done in gem5, working with the constraint of the few methods available. The first task for me was to identify the size for each of the partitions while keeping the privileged partition smaller than the unprivileged partition. After tweaking with the parameters, I decided to set a 3 items max to run the LRU section, in which each item has a privilege parameter set to 1 and they would be safe from eviction in that particular eviction cycle, but the worst of it's kind loses it's privilege and is replaced by an item with a very high frequency of request from the unprivileged section. All these checks are done in the getVictim() method because that is when I could do a sweep over all the candidates[] elements. Also, in this method, the unprivileged section having it's refCount variable set to the lowest is the target victim to be evicted. The problem with the getVictim() however, was that it was a const method. I needed to keep track of the number of items that were operating in the privileged section so I had to remove the const keyword from the files. The touch() method was used to update the refCount and the lastTick variables. If the time interval between currentTick an item's lastTick was greater than a certain time window (which I defined as time window = 100 in my code), the refCount of the item went back to 1. This ensured that the popularity of the item does not stay up indefinitely when they are not referenced anymore.

4.2 LFUDA

An initial challenge was to understand how cache sizes are set and then be able to specify the replacement policy within the build command. To achieve the first goal was relatively simple: we were able to understand cache configuration from the gem5 documentation and set the individual cache sizes. To achieve the second goal required considerable effort and research. We figured out that the replacement policy is explicitly specified inside *src/mem/cache/Cache.py*. Inorder to be able to specify it inside the build command requires creating a new attribute in cache options inside gem5. We first created a new flag for *l1d_replacement_policy*, *l2_replacement_policy* and *l1i_replacement_policy* inside *configs/common/Options.py*. Next we made a new replacement_policy_attr inside def_get_cache_opts() in *configs/common/CacheConfig.py*. With these two

changes we can add the - -l2_replacement_policy, - -l1i_replacement_policy, - -l1d_replacement_policy options inside our build command.

The original code implementation for LFUDA is done inside the folder *src/mem/cache/replacement-policies*. We first edited the SConscript to add the option for new replacement policy LFUDA. Next we edited the ReplacementPolicies.py to add LFUDA. In gem5, replacement policies are implemented in terms of reset(), touch(), invalidate() and getVictim() methods. The priority key for LFUDA is set/reset every time that item is accessed and also when it is first added to the cache. First, cache age (L), is initialized to 0 in the global space. Whenever an item is evicted using a call to invalidate(), L is set to that item's priority key. On reset(), whenever a replacement data is initialized, priority key is set equal to 1 + L. Then subsequently wherever touch() is called, the replacement entry is updated by incrementing the item's priority key. After source codes for replacement policy are implemented and all changes are made, we then run the scons build/RISCV/gem5.opt -j4 command. Once everything compiles without errors and smoothly, we then run the build command for each benchmark by varying the flags for cache sizes and replacement policy.

4.3 ARC

The infrastructure of gem5's replacement policy class is not favourable for ARC to be implemented the way it has been proposed in the paper. The major reason is that, the getVictim function does not contain the information of the newly requested block, which is necessary for choosing the victim in ARC. It is not feasible to successfully change the getVictim funcion and modify everywhere it has been called from within a short time because this method has been used in a lot if places in gem5 for various reasons. Making a change in it and making it error-free is a very extensive process. For this reason, we use the reset function, which contains the newly requested block information, to save the victim for future eviction. This anomaly hampers the natural flow of ARC algorithm and we had to make several adjustments to deal with it.

Initialization: Firstly, we implement the top and bottom parts of L1 and L2, which are referred to as T1, B1, T2, and B2 respectively, using four vectors. To create the replacement data for ARC, we declare ARCReplData that inherits ReplacementData class with an extra variable called "id". Initially we set the id to -1. For touch and reset method, we use the second type of declaration in base.hh that includes the packet information as parameter.

FUTURE_REPLACE: We define a function called FUTURE_REPLACE (int blockID) and use it instead of REPLACE in the original paper, because it is not possible to make a real eviction in any other function other than getVictim. FUTURE_REPLACE takes the id of a block as the parameter, and performs similar to the REPLACE method defined in the paper. However, the difference is that it does not perform the actual replacement. Rather, it stores which block is to be replaced in the future getVictim call and in which vector (T1 or T2) it can be found.

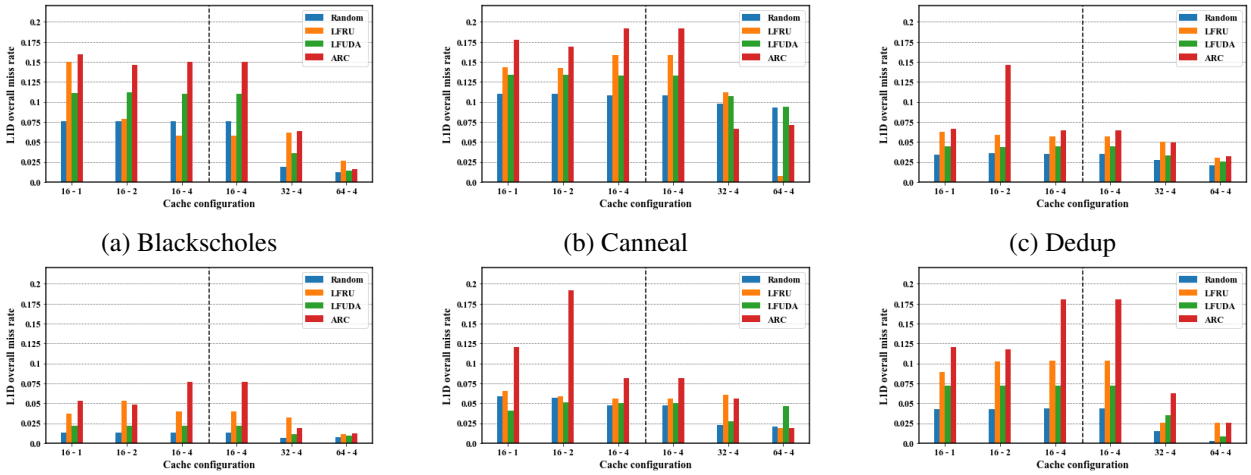
Reset: In every reset call, we first extract the address of the packet request by calling getAddr method. We right-shift this address by 8 bits to get the tag. This is based on our observation of the logs, that shifting the address by 8 bits should be the tag. We set tag as the id of the ReplacementData instance, so that it can be uniquely identified when needed, even after eviction. After setting the id, if a block has been evicted recently, we handle case II and III, where we check if the newly requested block is in B1 or B2. While handling case II and III, FUTURE_REPLACE is also called as necessary to store the future victim information.

Touch: We handle case I in touch method. In case I, we check if the block is in T1 or T2 and based on that, we move the block to the top of T2.

GetVictim: If there is no previously set information about the victim, then we preform case IV, as it evicts a block without the newly requested block information. Otherwise, we evict a block based on the previously saved information from reset. We also set a flag to indicate that a block has been evicted, so that the immediate reset call for the new block can handle case II and III and save the information about future replacement.

5 Testing, Evaluation, Results

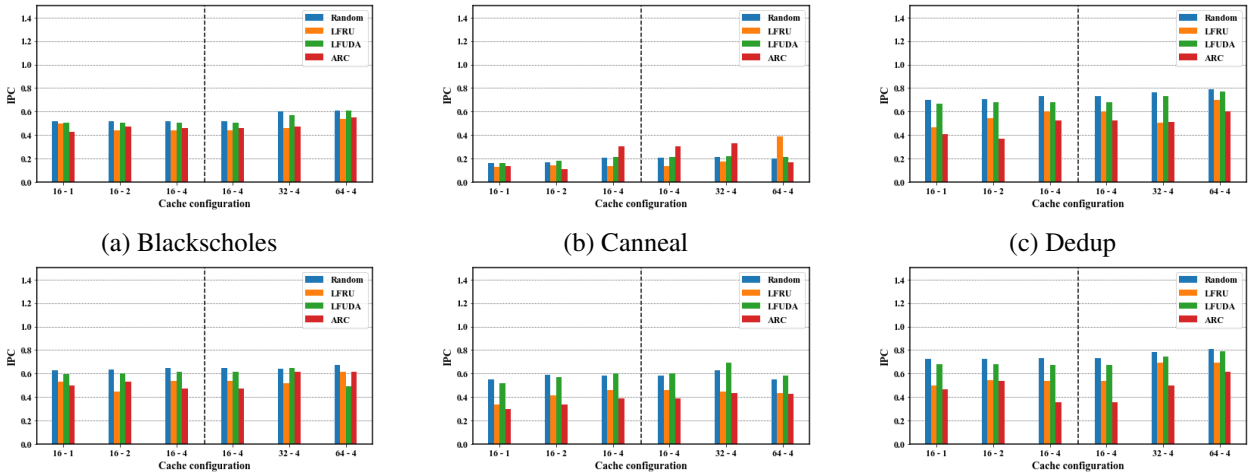
We tested our three replacement policies and Random for all the parsec benchamrks except freqmine. The left half of the subfigures in Figure 1 show the L1D overall miss rates when L1D size is fixed to 16 kB and L2 sizes are 1 MB, 2 MB and 4 MB. On the right half the L2 size is fixed to 4 MB and the L1D sizes are 16 kB, 32 kB and 64 kB. Figure 2 shows the IPC for the same cache configurations in a similar manner.



(a) Blackscholes (b) Canneal (c) Dedup

(d) Fluidanimate (e) Streamcluster (f) Swaptions

Figure 1: L1D Overall Miss Rates for Different Cache Configurations



(a) Blackscholes (b) Canneal (c) Dedup

(d) Fluidanimate (e) Streamcluster (f) Swaptions

Figure 2: IPC for Different Cache Configurations

Our first observation is that for any replacement policy, when the miss rate is high, the IPC is low. This is obvious because for higher misses we require more time to bring data L2 cache or the main memory. That causes more stalls in the pipeline and IPC goes low. Next, we observe that the miss rate drops significantly as we increase the L1D size but changing the L2 size has no effect. This is also obvious because changing increasing L2 size has nothing to do with L1 miss rate.

Random performs the best of all in almost all the cases. This could be because for parsec benchmarks,

possibly it is important for the complexity of the replacement algorithms to be low. Random has the simplest complexity among all the algorithms. Also, the complexity of ARC implementation is the highest, for which we can see that the miss rate for ARC is the notably higher than all the replacement policies.

LFRU performs better in blackscholes. It performs better than Random for cache configuration 16 kB - 4 MB in blackscholes. For canneal when the cahce configuration is 64 kB - 4 MB, it performs significantly well.

Overall among the three replacement policies that we implemented, LFUDA runs the best. The miss rate for LFUDA is close to Random in most of the cases.

Table 1: IPC, L1D Miss Rate, L1I Miss Rate, and L2 Miss Rate for All Benchmarks

	Conf.	IPC				L1D miss rate				L1I miss rate				L2 miss rate			
		Rand	LFRU	LFUDA	ARC	Rand	LFRU	LFUDA	ARC	Rand	LFRU	LFUDA	ARC	Rand	LFRU	LFUDA	ARC
blackscholes	16_1	0.518	0.497	0.505	0.43	0.076	0.15	0.111	0.16	0.002	0.002	0.003	0.01	0.003	0.005	0.002	0.03
	16_2	0.516	0.439	0.505	0.47	0.076	0.079	0.112	0.15	0.002	0.026	0.003	0.01	0.002	0.009	0.001	0.02
	16_4	0.517	0.438	0.506	0.46	0.076	0.058	0.11	0.15	0.001	0.028	0.003	0.01	0.001	0.0001	0.001	0.01
	32_4	0.6	0.458	0.568	0.47	0.019	0.062	0.036	0.06	0.001	0.025	0.002	0.02	0.006	0.008	0.003	0.01
	62_4	0.61	0.539	0.607	0.55	0.012	0.026	0.014	0.02	0.001	0.022	0.001	0.01	0.008	0.031	0.006	0.04
canneal	16_1	0.161	0.128	0.159	0.14	0.111	0.143	0.134	0.18	0.014	0.023	0.033	0.02	0.37	0.353	0.283	0.28
	16_2	0.168	0.14	0.178	0.11	0.11	0.143	0.133	0.17	0.013	0.023	0.032	0.02	0.237	0.298	0.211	0.34
	16_4	0.207	0.133	0.21	0.3	0.108	0.159	0.133	0.19	0.005	0.013	0.029	0.03	0.227	0.29	0.18	0.02
	32_4	0.210	0.174	0.222	0.33	0.098	0.112	0.107	0.07	0.006	0.024	0.009	0.03	0.241	0.262	0.216	0.09
	62_4	0.198	0.39	0.21	0.17	0.093	0.008	0.094	0.07	0.006	0.029	0.004	0.02	0.196	0.078	0.182	0.31
dedup	16_1	0.702	0.463	0.666	0.41	0.035	0.063	0.044	0.07	0.013	0.021	0.018	0.02	0.06	0.159	0.04	0.27
	16_2	0.708	0.544	0.678	0.37	0.036	0.058	0.044	0.15	0.014	0.020	0.018	0.03	0.041	0.066	0.03	0.03
	16_4	0.731	0.604	0.679	0.52	0.035	0.057	0.045	0.06	0.008	0.018	0.018	0.02	0.038	0.022	0.024	0.1
	32_4	0.763	0.504	0.733	0.51	0.027	0.05	0.033	0.05	0.007	0.026	0.011	0.02	0.049	0.01	0.035	0.06
	62_4	0.788	0.701	0.772	0.60	0.021	0.03	0.025	0.03	0.008	0.016	0.006	0.02	0.060	0.045	0.048	0.17
swaptions	16_1	0.727	0.499	0.679	0.47	0.042	0.9	0.072	0.12	0.006	0.028	0.006	0.02	0.0005	0.013	0.000	0.07
	16_2	0.727	0.544	0.681	0.53	0.042	0.103	0.072	0.12	0.006	0.027	0.006	0.02	0.0004	0.008	0.000	0.02
	16_4	0.729	0.537	0.675	0.35	0.043	0.104	0.072	0.18	0.004	0.025	0.006	0.03	0.001	0.000	0.000	0.00
	32_4	0.782	0.693	0.743	0.5	0.015	0.026	0.035	0.06	0.003	0.024	0.004	0.03	0.001	0.002	0.000	0.00
	62_4	0.807	0.693	0.789	0.61	0.002	0.026	0.009	0.03	0.003	0.024	0.002	0.02	0.003	0.002	0.002	0.01
streamcluster	16_1	0.552	0.334	0.517	0.29	0.058	0.066	0.04	0.12	0.005	0.034	0.009	0.02	0.042	0.204	0.188	0.15
	16_2	0.588	0.414	0.571	0.33	0.057	0.058	0.051	0.19	0.004	0.03	0.012	0.03	0.014	0.033	0.038	0.00
	16_4	0.582	0.458	0.598	0.39	0.047	0.056	0.05	0.08	0.002	0.027	0.011	0.02	0.003	0.013	0.011	0.14
	32_4	0.625	0.443	0.695	0.43	0.023	0.06	0.027	0.06	0.001	0.031	0.004	0.03	0.011	0.044	0.012	0.03
	62_4	0.55	0.431	0.582	0.43	0.021	0.019	0.047	0.02	0.001	0.032	0.003	0.03	0.004	0.044	0.005	0.09
fluidanimate	16_1	0.625	0.532	0.598	0.5	0.013	0.037	0.022	0.05	0.002	0.008	0.004	0.01	0.27	0.16	0.156	0.11
	16_2	0.631	0.448	0.604	0.53	0.013	0.053	0.022	0.05	0.002	0.027	0.004	0.01	0.24	0.007	0.141	0.1
	16_4	0.645	0.536	0.614	0.47	0.013	0.04	0.022	0.08	0.001	0.014	0.004	0.01	0.203	0.063	0.112	0.04
	32_4	0.637	0.517	0.645	0.61	0.007	0.033	0.011	0.02	0.001	0.02	0.002	0.00	0.333	0.041	0.224	0.22
	62_4	0.674	0.613	0.493	0.61	0.007	0.011	0.009	0.01	0.001	0.009	0.002	0.01	0.387	0.199	0.309	0.25

Table 1 contains all the data that we have collected from our benchmarks to three decimal places.

Overall, the IPCs remained fairly similar with all the three implemented replacement policies for the different cache configurations under the six shown benchmarks. When compared against the Random replacement policy for miss rates in L2 cache, our three policies fared quite well, sometimes even out-performing it for some of the benchmarks such as fluidanimate and canneal.

Parsec benchmark comparison. Random vs. other replacement policies in gem5: Based off of parsec benchmark results each of the teams simulated, the random replacement policy performed better than FIFO for smaller cache configurations and is on par with it for larger sizes. LRU shows better overall results than almost all the other replacement policies, which is not surprising.

6 Conclusion

Despite the constraints of gem5's replacement policy classes and methods, there were times we had to work our way around it, the implementations of the three cache replacement policies have been demonstrated well when compared to an already built version of Random replacement policy in gem5. There were slight ambiguities when deciding some of the parameters but our team was hard at work and eventually found optimal grounds. This was a great, hands-on experience with gem5 and all of us learned a lot.