GHOST-CACHE CRFP ALGORITHM SIMULATOR BASED ON TRACE AND FILE SYSTEM

Wahyu Suadi
Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia
wsuadi@if.its.ac.id

Supeno Djanali
Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia
supeno@its.ac.id

Radityo Anggoro
Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia
onggo@if.its.ac.id

Abstract

Solid State Drive (SSD) is fast media that has become popular recently. Because it has a smaller size, it acts on larger and slower media as a cache. There are researches on SSD as a cache on Hard Disk (HDD) media. This paper present GCRFP algorithm and test its performance using two simulators. The first simulator is implemented by using trace input, the second using user-space file-system. The purpose of these simulators is to study the algorithm performance behavior. The findings demonstrate that GCRFP gives LARC comparable performance in both simulators.

***Keywords*: SSD, cache algorithm, Linux, FUSE.**

1. Introduction

In latest decade, flash memory media, particularly Solid State Disk (SSD), has changed the mindset of mass storage media due to low power consumption, shock resistance, and higher read-write capability. Regardless of these preferences, in term of its size, SSD is the smaller compared to a hard disk (HDD). This smaller size and relatively higher cost, rouse to utilize SSD as a cache for HDD. Most of the cache algorithm load a frequently used data in faster memory Rather than a slower memory access. Thus it able to increase system performance [1].

Due to its limitations on the writing counter, SSD will experience a rapid decrease of its lifetime. Moreover, Cache algorithms that commonly applied for disk and memory cannot be directly implemented on SSD since, Cache algorithm does not consider writing counter that will decrease the media life. For instance, the LRU algorithm only counts the latest access and quickly transfer the data from HDD to SSD. The old data is replaced by the latest data in very fast cycle thus it creates a high number of writing. Thus it does not compatible for SSD cache.

Recent researches [2] implement LARC, Lazy Adaptive Replacement Cache, on applying SSD as a cache for HDD. This algorithm separate the blocks that are rarely accessed and keeps them from being cached. the Ghost-cache in LARC is act as an LRU queue, which stores an information of the block rather than an actual block data. GCRFP [3] also implement ghost-cache. It combined the LRU (Least Recently Used) and LFU (Least Frequently Used) policy. Different method in utilizing ghost-cache is used in popular block selection [4].

2. Related Works

SSD usage model showed in Fig.1(a), is used as a memory system extension which, RAM and SSD are used as a cache tier against the HDD. On the other hand, SSD can be used as an extension disk that accessed using a standard interface which is transparent to other components as it shown in Fig.1 (b). Our research utilized SSD as a cache of HDD as it adopted on the second model.

Recently there have been a lot of variation on cache algorithm. In order to maximize cache efficiency, it store its popular blocks to cache. The concept of temporal locality is used where, a block that just previously used, is having a higher probability of reuse than pre-accessed blocks. On the other hand, several blocks are accessed more than many others as known as the skewed popularity. Two algorithms adopted this concept such as LRU (Least Recently Used) and LFU (Least Frequently Used). The LRU algorithm implements temporal locality to increase the hit ratio. It keep a popular block in the past and expect that it will be re-accessed. LRU-K [5], CRFP [6], and ARC [7] improves LRU in many aspects, such as the improvement of LFU historical count into LRU. It considers the access number in a block that increase the respective hit counter. However, LFU originally has an aging issue that is a popular block never leaves the cache even though it never getting access. Thus, the space is always occupied. FBR [8] has been proposed to solve that problem.

A cache memory compensate the speed gap between register (fast memory) and main memory (slow memory). It reduces significantly the speed difference between registers, main memory and HDD. In this paper, a cache is utilized to solve the speed issues by storing potential data on the SSD.

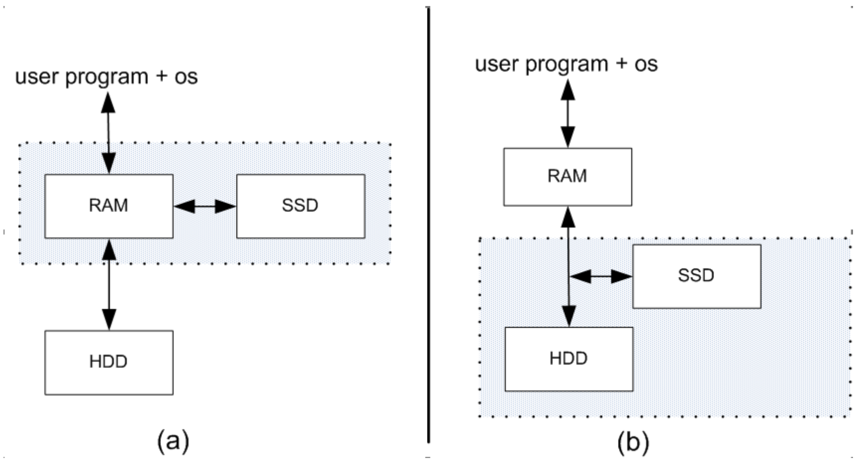

Fig. 1 SSD cache and HDD

Since a cache has limited space, it only store the block that has potential reuse in the future by using Page Replacement Policy. The Lazy Adaptive Replacement Cache (LARC) algorithm identifies rarely accessed blocks and keeps them away from cache. It implement ghost cache as an LRU queue, which is only stores the information of the block rather than the actual block data. Ghost-cache store the block at the first time used. If the block is being reused, it will be considered as a popular block then it moved into SSD and LRU rules is applied. Every access to the block, will check it availability on SSD. If it exist, the block is moved to MRU (Most Recently Used). Otherwise, it will access ghost-cache and the new block will enter LRU on SSD. However, when the block is unavailable, the request will be fetched from the HDD and it will be stored in the ghost-cache. In case that SSD and ghost-cache do not have enough space, the LRU blocks will be discarded. This method tries to reduce the changes in SSD, thus reducing the usage of write cycle. Paper [9] approach a different method to reduce it.

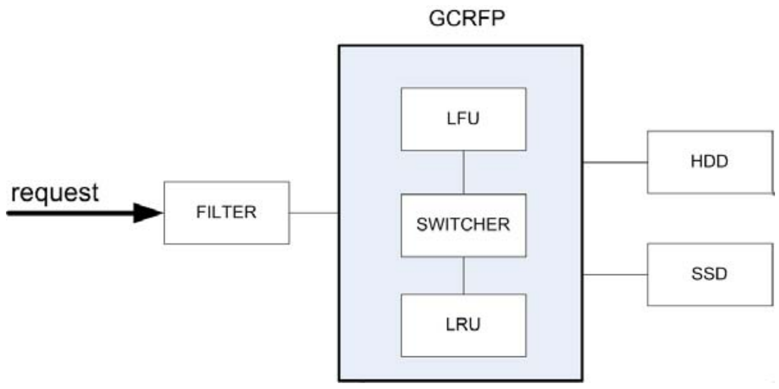

Fig. 2 Data flows, beginning from application requests to the respective data source

3. Ghost-cache CRFP Algorithm

Ghost-cache CRFP (GCRFP) has two ghost-cache method. The first, ghost-cache is situated in the front station as a filter, as in LARC. The second, ghost-cache is act as a record of frequencies from the released block, as similar as in CRFP. Since the distribution of blocks is usually unsymmetrical, and only a few block is being reuse, thus only a few block allows to enter SSD. So, it reduce writing frequency. The second ghost-cache in CRFP also record the changes in access patterns. This algorithm allows to switch its mode into LRU or LFU. It turn into LFU when the frequency-based access pattern is dominant. In contrast, it turn back into LRU when the number of missed access beyond the threshold (miss/hit). Fig 2 shows GCRFP and CRFP data movement.

In this paper, file-system based simulator is used to simulate GCRP. The modification to our previous work [3] is needed to adjust the trace-based into file-system based simulator. The other methods such as LFU, LRU, and LARC are also simulated. The benchmark that we refer for this user space file-system cache simulator by using database, disk and file-system

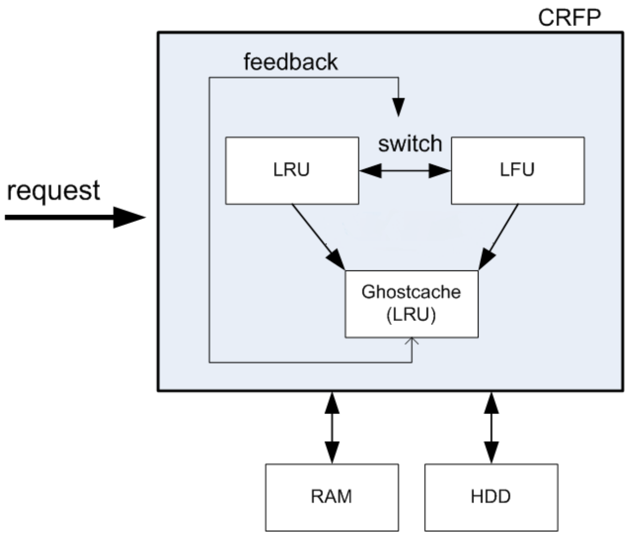

Fig. 3 CRPF Algorithm

In this paper, we focus on using filebench [10]. This application is suitable for our experiment since it already has several standard workloads. LRU algorithm allows to move list of element into MRU each time it being accessed. If the space of data structure is overloaded, The LRU will be removed. Both LRU and LFU are widely combined with many replacement policy algorithms. However, both have their advantages and disadvantages. LRU is able to catch the current aspects of popular blocks based on temporal locality. Meanwhile LFU store the information of the most popular block. ARC [7] has combine those two algorithms and adopt each of its advantages. However, in case that the access pattern is change, the issue is occur since either LRU or LFU cannot always give a good result.

```
reset  H & O values                /* H=O=0 */
if request in LRUout then
  move it main LRU
  inc(H)
else
  inc(O)

if (SWITCH == 0) then              /* LRU
policy */
  if  H>SWITCH_TIMES and
              (H / O) > SWITCH_RATIO then
      reset  H & O values
      SWITCH = 1       /* switch to LFU */
if (SWITCH == 1) then             /* LFU
policy */
   if O>SWITCH_TIMES and
             (O / H) > SWITCH_RATIO then
      reset  H & O values
      SWITCH = 0       /* switch to RU */
```

Fig. 4 CRFP and Ghost-cache Algorithm

CRFP able to solve that problem by adaptively switching LRU and LFU simultaneously as it shows in Fig. 3. However, switching mechanism is needed to facilitate the work flow of CRFP. Fig. 4 describe how switching mechanism is applied to CRFP. Fig. 4 describes the algorithm. H is a hit ghost-cache out. O is if there is a miss. SWITCH_TIMES is the number of times the number of hit / miss. SWITCH_RATIO is the ratio between H / O or O / H.

4. FUSE Simulator Implementation

The simulator is Linux based operating system using the FUSE library (Filesystem in Userspace) go programing language [10][11] as it shown in Fig. 5. The compiled programs enable an application to run faster than using an interpreter like python. Data and request move from user application, through fuse kernel module, into fuse userspace.

The system consists of cache algorithm modules, filesystem modules, and inode modules. Cache library uses interface feature of go programing language, so each algorithm must follow the same methods. The filesystem uses a specified cache algorithm for both reading and writing. While all the basic operations are handled by inode module. In order to run this application, the cache library needs modification. In the filesystem, there is no direct block access, all accesses are go through inode number, offset, and operation type. For that, the identity mapped to a tuple. These tuples are keys in the data structure, in red-black trees, linked-lists, and arrays.

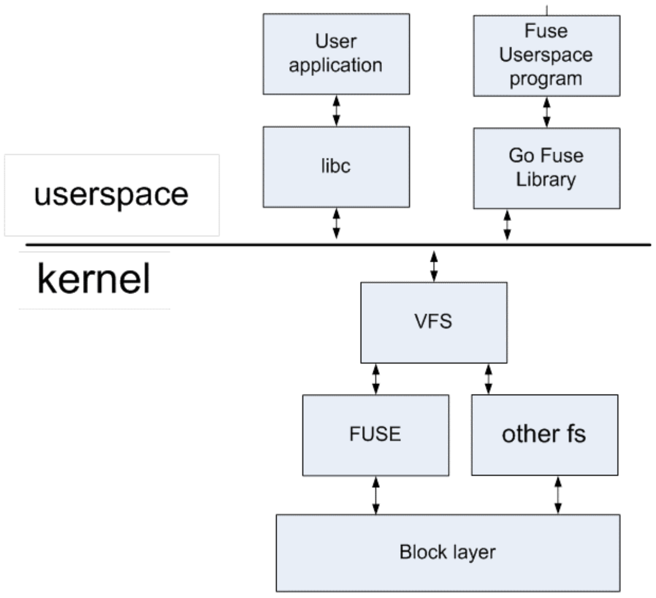

Fig. 5 Fuse library, filesystem implementation, and user application reside in userspace

The standard block size, 4096 bytes, is applied to simplify the design. Offset and request size are also adjusted to the page size. The inode module is built on top of the memory filesystem. This module is taken from the fuse distribution [11] and has implemented the functions required for a standard filesystem.

5. Simulation and Results

Simulation is implemented on a Linux machine with an Intel i5-7400 processor with 8GB of memory. Memory size limit to the number and file size in the filesystem. Filebench [11] is used as a benchmark for our experiment. It has scenarios that include on how to create a specific workload based on an application type. This scenario able to create workloads with several parameters for instance, I/O size, number of concurrent threads, and read/write ratio. There are many predefined scenarios included in the distribution of the application. The workload used in this testing is OLTP, fileserver, and webserver.

We show on two table for each workload experiment, the hit and write ratio table. The hit ratio represents the number of hits on the SSD divided by the total I/O operations. On the other hand, write ratio shows the number of write operations that occur on the SSD divided by the total I/O operations. A write operation is happening on a file system write operation or a page replacement at SSD cache. Each experiment runs three times and it averages from the results.

Table 1 show that LRU obtains the best hit ratio, but with a high number of data writes to the cache. Since that dataset is small, the LRU cache could contain many hot blocks, but with a high cost of many SSD cache replacement. LARC gets a smaller hit ratio but with the smallest number of write ratios. LARC ghost cache could filter much recent access that is not happening again fast enough. LFU is performing worst because it could not detect a change of access pattern fast enough and contain data that was once popular but not popular in the future. GCRP hit ratio and write ratio is comparable to LARC, within 1-3% for hit ratio and 1-2 % for write ratio.

Hit Ratio (%)	**Cache Size**				
	10,000	**20,000**	**30,000**	**40,000**	**50,000**
LARC	27.89	29.98	30.14	29.99	30.07
LFU	1.78	3.39	5.17	6.84	8.58
LRU	45.57	53.52	54.08	54.21	54.20
GCRFP	28.54	29.93	30.03	30.06	30.04
Write Ratio (%)	**Cache Size**				
	10,000	**20,000**	**30,000**	**40,000**	**50,000**
LARC	27.66	28.48	28.51	28.59	28.56
LFU	98.51	97.04	95.40	93.86	92.27
LRU	58.92	50.96	50.40	50.27	50.28
GCRFP	28.69	28.62	28.59	28.54	28.60

Tabel 1 File Server Experiment

It shows that LRU still has the best hit ratio, but with a highest write ratio. LARC has lesser hit ratio but with the best write ratio. The LFU performs worse than the OLTP workload. In this workload, GCRP comparable with LARC, with small spread, within 1-2% in both hit ratio and write ratio.

Table II shows that in this workload describes the sharing of many files from a networked machine to many clients. LARC has the best hit ratio and the best write ratio. On the other hand, LRU has the worst ratio, both in a hit and write. It shows that this workload is the worst match for LRU. LFU performance is better than LRU. GCRFP also comparable to LARC, especially with bigger cache size.

Hit Ratio (%)	**Cache Size**				
	3,000	**3,250**	**3,500**	**3,750**	**4,000**
LARC	50.99	57.73	65.31	73.67	82.80
LFU	34.38	38.52	46.81	57.52	69.72
LRU	0.05	0.07	0.07	0.08	0.10
GCRFP	39.14	50.08	61.19	72.12	83.40
Write Ratio (%)	**Cache Size**				
	3,000	**3,250**	**3,500**	**3,750**	**4,000**
LARC	22.56	19.43	15.74	11.52	7.15
LFU	65.64	61.50	53.22	42.49	30.29
LRU	99.96	99.94	99.93	99.92	99.90
GCRFP	28.07	22.66	17.12	11.76	6.10

Table 2. Web Server Experiment

6. Conclusions

Experiments show that, for most cases, with this simulator GCRFP gives comparable performance to LARC. Combined with complex logic and codes, GCRP gives small to no benefit to LARC. Other conclusions, GCRFP and LARC do not always provide the best hit ratio but consistently bring good results in a write ratio. Hence, it gives more prolonged usage for SSD but still good hit ratio and fast execution. Since a bad write ratio not only diminishes the writing cycle and it also slows down SSD since its internal cache is often full. It also shows that LRU and LFU are not a good candidate cache algorithm for SSD caching, although sometimes they could give a good hit ratio, it upset with a big write ratio.

This work also shows that the simulation of cache algorithm could be done in userspace filesystem and give a more practical working environment for researcher.

In the future, more thorough research can be explored on optimizing parameters from CRFP algorithm to be used in SSD caching. Different methods for setting switch ratio value such as cache_hit / outlru_hit or out_hit / cache_hit) and switch_times are potential alternatives.

References

[1] S. S. Rizvi and T. Chung, "Flash SSD vs HDD: High performance oriented modern embedded and multimedia storage systems," *2010 2nd International Conference on Computer Engineering and Technology*, 2010, pp. V7-297-V7-299, doi: 10.1109/ICCET.2010.5485421.

[2] S. Huang, Q. Wei, D. Feng, J. Chen and C. Chen, "Improving flash-based disk cache with lazy adaptive replacement", Trans. Storage, vol. 12, no. 2, pp. 8:1-8:24, Feb. 2016, [online] Available: http://doi.acm.org/10.1145/2737832. Suadi, W. (2020). GCRFP - Page Replacement for Solid State Drive using Ghost-Cache. *JUTI: Jurnal Ilmiah Teknologi Informasi, 18*(2), 85-93.

[3] Y. Ryou, B. Lee, S. Yoo and H. Y. Youn, "Considering block popularity in disk cache replacement for enhancing hit ratio with solid state drive," 2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015, pp. 1-6, doi: 10.1109/SNPD.2015.7176246.

[4] E. J. O'Neil, P. E. O'Neil and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering", ACM SIGMOD Rec., vol. 22, no. 2, pp. 297-306, Jun. 1993.

[5] Z. Li, D. Liu and H. Bi, "CRFP: A Novel Adaptive Replacement Policy Combined the LRU and LFU Policies," 2008 IEEE 8th International Conference on Computer and Information Technology Workshops, 2008, pp. 72-79, doi: 10.1109/CIT.2008.Workshops.22.

[6] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In Proceedings of the 2nd USENIX Conference on File and Storage Technologies(FAST'03). San Francisco, CA, 115-130

[7] T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement", Proc. ACM SIGMETRICS Conf. Meas. Modeling Comput. Syst., pp. 134-142, 1990

[8] Y. Chai, Z. Du, X. Qin and D. A. Bader, "WEC: Improving Durability of SSD Cache Drives by Caching Write-Efficient Data," in IEEE Transactions on Computers, vol. 64, no. 11, pp. 3304-3316, 1 Nov. 2015, doi: 10.1109/TC.2015.2401029.

[9] GolangDev. (n.d.). *Go Programming Language*. (Google) Retrieved from https://golang.org/

[10] Jacobs, A. (n.d.). *Package fuse enables writing and mounting user-space file systems*. Retrieved from https://godoc.org/github.com/jacobsa/fuse

[11] Tarasov, V. (Spring 2016). Filebench: A Flexible Framework for File System Benchmarking. *;login: usenix magazine*.