International Journal of Advanced Smart Convergence Vol.11 No.2 1-6 (2022)
http://dx.doi.org/10.7236/IJASC.2022.11.2.1

IJASC 22-2-1

CPC: A File I/O Cache Management Policy for Compute-Bound Workloads

Hyokyung Bahn

Professor, Department of Computer Engineering, Ewha University, Korea
bahn@ewha.ac.kr

Abstract

With the emergence of the new era of the 4th industrial revolution, compute-bound workloads with large memory footprint like big data processing increase dramatically. Even in such compute-bound workloads, however, we observe bulky I/Os while loading big data from storage to memory. Although file I/O cache plays a role of accelerating the performance of storage I/O, we found out that the cache hit rate in such environments is not improved even though we increase the file I/O cache capacity because of some special I/O references generated by compute-bound workloads. To cope with this situation, we propose a new file I/O cache management policy that improves the cache hit rate for compute-bound workloads significantly. Trace-driven simulations by replaying file I/O reference logs of compute-bound workloads show that the proposed cache management policy improves the cache hit rate compared to the well-acknowledged CLOCK algorithm by a large margin.

Keywords: *Compute-bound workload, file I/O cache, cache hit rate, storage, CLOCK.*

1. Introduction

Due to the recent technology enhancement in many-core computing technologies, large footprint workloads like graphic rendering and deep learning grow rapidly [1]. This is in line with the emerging era of the 4th Industrial Revolution, where compute-bound applications or generating high quality contents is not the domain of the specialized industries any longer [1, 2].

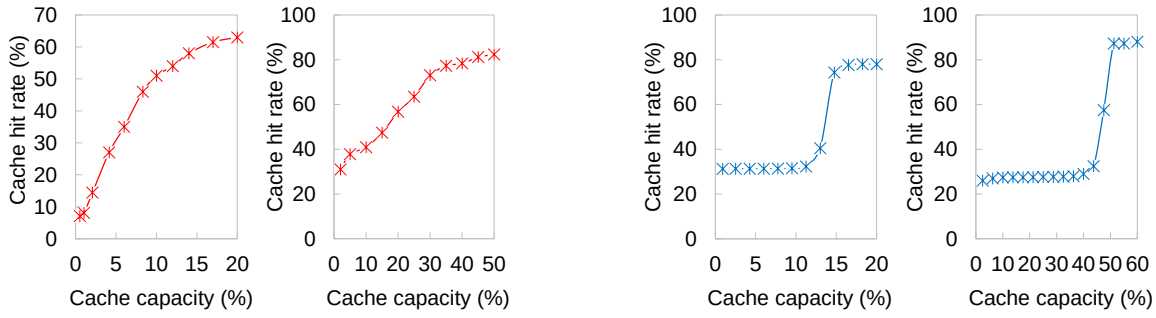
Even in such compute-bound workloads, file I/O references for reading input files from data storage or writing output files to storage may be a performance bottleneck [3, 4]. This is because accessing data storage via I/O is about five to six orders of magnitude slower than computing in processor cores [3, 5]. File I/O cache is a well-known technique to buffer the speed gap between data storage and computing cores. In file I/O cache, data files accessed from the data storage are stored in a portion of DRAM memory called file I/O cache, which can be reused without storage accesses in case the same files are used again in the future [6, 7]. As the size of the file I/O cache is not infinite, a certain amount of data files in the file I/O cache should be discarded in order to maintain new data files in case the cache space is exhausted. This article makes an observation that file I/O

Manuscript Received: March. 3, 2022 / Revised: March. 7, 2022 / Accepted: March. 9, 2022
Corresponding Author: bahn@ewha.ac.kr
Tel: +82-2-3277-2368, Fax: +82-2-3277-2306
Professor, Department of Computer Engineering, Ewha University, Korea

cache for conventional systems does not behave efficiently for compute-bound workloads, and proposes a new file I/O cache management policy customized for compute-bound workloads.

To this end, this article gathers file I/O logs while running a certain number of compute-bound workloads and analyzes their characteristics. By this analysis, we categorize file I/O patterns in the logs into 3 types. The 1st is the sequential read references that happens when the workload begins its launch. The 2nd reference type is the multiple read references including loop and temporally co-related. As compute-bound workloads generally execute long computations after a bulk of storage I/O, this type of file references is utilized as input files for computations. The 3rd reference type is sequential write references. This occurs due to the file writes after the completion of computing.

Our observations exhibit that the aforementioned file I/O reference patterns of compute-bound workloads deteriorate the performance of the file I/O cache significantly. To resolve this issue, this article proposes a new file I/O cache policy to enhance the file I/O performance of compute-bound workloads by making use of the observation results of I/O logs. Our experimental results based on log replaying simulations show that the presented file I/O cache policy enhances the file I/O performance of compute-bound workloads by an average of 96% in comparison with the current system adopting the CLOCK algorithm.



(a) Traditional workloads **(b) Compute-bound workloads**

Figure 1. The cache hit rate of the file I/O cache.

2. File I/O Cache Performance in Compute-bound Workloads

In this section, we depict the performance of traditional file I/O cache when running conventional workloads in comparison with compute-bound workloads, and argue that the traditional file I/O cache does not perform well for compute-bound workloads.

Figures 1(a) shows the cache hit rate of conventional workloads as the cache capacity increases. Note that this experiment has been performed with the conventional file I/O cache settings by making use of the CLOCK algorithm. In this experiment, the x-axis is scaled relative to the total referenced file capacity of workloads. That is, 100% in the x-axis shows the configuration that the entire referenced files of the workload can be maintained in the file I/O cache at the same time, without incurring eviction of cached items. This is an identical condition with the infinite capacity cache, where any cache management policies result in the same cache hit rate, not being possible in real world situations. Usually, the capacity of the file I/O cache in real environments is often configured to smaller than 50%, where removal happens essentially after the warm-up period of the

system. As we see from Figure 1(a), the cache hit rate of the file I/O cache increases as the cache capacity becomes large in conventional workloads.

From now on, let us discuss the cache hit rate of compute-bound workloads. Figure 1(b) shows the cache hit rate for similar settings of Figure 1(a). As we see, the curves form almost horizontal lines when the cache capacity is not large enough. The performance improvement can be seen only after the enough capacity cache is provided at the right side of the figures. This implies that the file I/O cache performance in compute-bound workloads does not improve if the capacity of the file I/O cache is not large enough to preserve a certain portion of hot referenced files of the workload. For this reason, traditional file I/O cache policies are not efficient to use in compute-bound workload environments. This article aims to design an efficient file I/O cache management policy for compute-bound workloads, thereby improving the cache hit rate gradually as the capacity of the file I/O cache grows. To do so, we anatomize the file I/O reference logs of compute-bound workloads and suggest a file I/O cache management policy considering the observation results.

3. File I/O Cache Management for Compute-bound Workloads

To see the effectiveness of file I/O cache in compute-bound workloads, this article anatomizes the file I/O reference logs of two compute-bound workloads. Through this analysis, the three types of reference characteristics have been observed. First, the sequential read references while launching the workloads have been classified. Actually, these references cannot contribute to improving file I/O cache performances since they are referenced only once. Note that caching is effective only when data files are referenced again in the future. However, we can consider these reference patterns as long looping references because the same data files will be re-referenced in case the workload will run again in future. We anatomize these references and observe that they are files for configuring workload launching. The second reference type is multiple read references including loop and temporally co-related. As compute-bound workloads are typically composed of short file I/O references and long computation periods, this type of file references will be used as input files for computations. Note that these multiple read references are composed of short looping references and temporally co-related references. The third reference type is sequential write references. This occurs due to the file writing after the completion of computations, thereby flushing to storage. Since the three reference types we anatomized can be classified based on the attributes of files, this article aim to manage the file I/O cache customized for compute-bound workloads through the reference types of data files.

When various reference types are mixed in file I/O references, partitioning the file I/O cache regions for each reference type and utilizing proper policies will be efficient. This article divides the file I/O cache regions by making use of the expected performance benefit of each reference type while workload runs and makes use of suitable caching policies.

First, if a data file has been referenced and is not used again, we call this type of data the single-reference data, and we do not maintain such type of data in the file I/O cache. This is because expected performance benefit is 0 when we store such data in the file I/O cache. Second, if a looping reference type is classified, we assign a new cache region based on the period of the references considering the expected performance benefit. To this end, some literatures detect reference types, but this causes large overhead of online management [8]. Thus, instead of instant detection, this article makes use of the file I/O reference characteristics already anatomized in compute-bound workloads for managing the file I/O cache with low overhead.

Our file I/O cache is composed of a configuring region C for sequential read references that happen while launching the workload, and a short looping reference region S, and a temporally co-related reference region

T. It does not allocate a file I/O cache region for sequential write references as they are not effective in improving cache performances. Instead, we assign a certain size of buffer area for temporarily buffering the data files to be flushed to storage and remove them right after written to storage. The capacities of the cache regions C, S, and T are periodically adjusted based on the contribution of each region to the cache hit rate.

Our policy utilizes the history buffers C', S', and T' to estimate the effectiveness of the corresponding regions C, S, and T, respectively, and resizes each region. The history buffers predict the performance benefit of each region when the capacity of the cache region grows. In particular, history buffers only maintain the attributes of files discarded recently from the corresponding regions. By utilizing the attributes of the discarded file data, our policy can evaluate the performance benefit of the region in case its capacity grows. If frequent references are monitored in the history buffers of a certain region, our policy increases the capacity of the corresponding region to enhance the cache hit rate of that region. As the total cache capacity is limited, the capacity of other regions need to be adjusted together when the capacity of a certain region grows due to frequent references in its ghost buffers. Assume that data files in C' are referenced frequently. Then, our policy increases the capacity of C to store more configuring files in the cache, and then decreases the capacities of S and T. As a result, the capacity of the history buffer C' is reduced. This is because the cache hit rate of the total file I/O cache can be predicted if the total capacity of the history buffer and the corresponding region will be the full capacity of the file I/O cache. Previous studies also used some types of history buffers and the overhead of history buffers is known to be small enough since they need only the attributes of data files, which is smaller than tens of bytes whereas an actual data file is at least 4 kilobytes [9, 10].

Now, let us explain the details of the proposed policy. If a data file is retrieved and needs to be stored in the file I/O cache, our policy checks the attributes of the file and the reference type of the file referenced. If it is a sequential write reference for storing the computation output, our policy adds it to the buffer area. As aforementioned, data files maintained in the buffer area will be flushed to storage soon, so they exist temporarily in the buffer and will be discarded. Otherwise, data files are added to the configuring region C, short looping reference region S, or a temporally co-related reference region T based on their file attributes. As data files in C and S generate looping references, the most recently used data is discarded first if removal is necessary. The reason is that the least recently used data will be reused first in the looping reference patterns [8]. In contrast, when removal is necessary in the T region, we evict the least recently used item as it is known to perform well in workloads with temporally co-related references [8].

If the requested data file already exists in the file I/O cache, it can be found either in C, S, or T region. In case the data file is in either C or S region, it is essentially a part of a looping reference pattern, so we decrease its priority to the lowest in that region as it will be re-referenced farthest. If the requested data file is found in the T region, we raise its priority to the highest in that region. Even if the requested data file does not exist in the file I/O cache, the attribute of the data file can be found in the history buffer. In such a case, the capacity of the history buffer as well as the actual region are adjusted appropriately. For re-balancing the total file I/O cache, our policy adjusts the capacity of the other regions as well. There are three cases that the attribute of a data file is found in the history buffer. First, in case the requested data file exists in C' history buffer, the capacity of C increases, and the capacity of C' is reduced by the same amount. The capacity of S or T is decreased and the capacities of their corresponding history buffers are adjusted accordingly. Second, when the requested data file exists in the S' ghost buffer, the capacity of S increases and the capacity of S' is reduced by the same amount. Then, the capacities of other regions and their history buffers are adjusted accordingly. Third, when the requested data file exists in the T' history buffer, the capacity of T increases, and its history buffer decreases by the same amount. Our policy also adjusts the capacities of other regions and their history

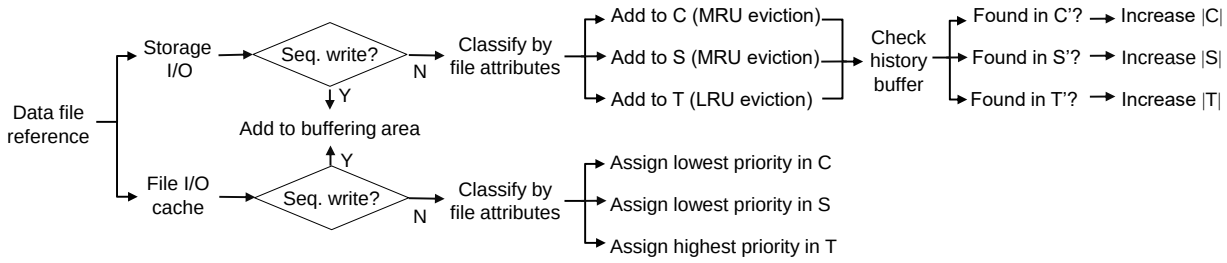
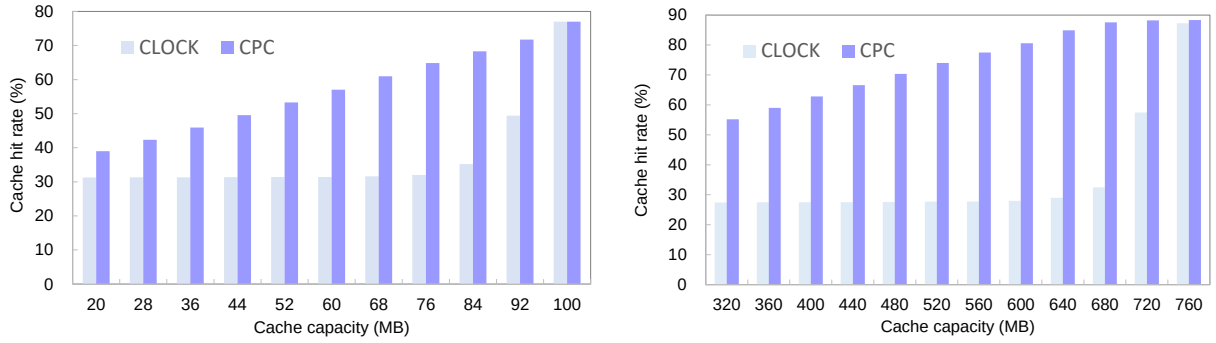


Figure 2. A conceptual flow of our file I/O cache.



(a) workload 1 **(b) workload 2**

Figure 3. The cache hit rate of CIC compared to CLOCK.

buffers accordingly. Figure 2 depicts the conceptual flow of our file I/O cache.

4. Simulation Experiments

To validate the performance of the proposed policy, trace-driven simulations have been conducted with file I/O workload logs. The system we executed the workloads is a desktop consisting of Intel i7-7700 4-core CPU with 8KB cache memory, 4GB DRAM, and 192MB/s SATA HDD. The size of file I/O cache assigned to the workloads is about 1GB. We used Linux Ubuntu 16.04 as our OS. The file I/O workload logs were collected at the system call layer while the workloads ran. We compare our policy with the CLOCK replacement policy. Figure 3 shows the cache hit rate of CLOCK and the proposed policy that we call CPC (Caching Policy for Compute-bound workloads) as the file I/O cache capacity changes. As we see from this figure, CPC exhibits better cache hit rate than CLOCK for all workload cases. As compute-bound workloads have a lot of short looping references, CLOCK does not exhibit good results when the file I/O cache is small for maintaining the looping references. This is because file data are removed before referenced again if the file I/O cache is smaller than the loop size. CLOCK shows good performances only when the file I/O cache is large enough to maintain the entire looping references. For this reason, growing the file I/O cache size does not result in the performance gain until a certain large capacity.

Specifically, the performance improvement of the proposed CPC policy is 96% on average and up to 192% against CLOCK. Our simulation result has shown that CLOCK also performs reasonably well if the file I/O cache is large enough. However, this may not be the case in real system environments because of the evolution of workloads as time goes on. Please, note that the performance benefit of CPC is gradual as the file I/O cache capacity increases and this is independent of the size of workloads to be executed in the system.

5. Conclusion

This article proposed CPC, a file I/O cache management policy customized for compute-bound workloads. In our preliminary analysis, we found out that compute-bound workloads show a certain special file reference characteristics, which deteriorates the cache hit rate of the file I/O cache seriously. To cope with this situation, this article suggested a new file I/O caching policy, which partitions the cache regions for each reference type according to the contribution of each type of references to the performance enhancement, and monitors the assigned cache regions for resizing them based on workload evolutions. Trace-driven simulations showed that CIC performs better than the well-known CLOCK policy with respect to the cache hit rate by 96% on average and up to 192%.

Acknowledgement

This work was supported by the IITP grant funded by the Korea government (MSIT) (No.2021-0-02068, Artificial Intelligence Innovation Hub) and the ICT R&D program of MSIT/IITP (2018-0-00549, Extremely Scalable Order Preserving OS for Manycore and Non-volatile Memory).

References

[1] G. Patil, S. Deshpande, “Distributed rendering system for 3D animations with Blender,” Proc. IEEE Conf. on Advances in Electronics, Communication and Computer Technology, pp.91-98, 2016.
DOI: https://doi.org/10.1109/ICAECCT.2016.7942562

[2] S. Yoo, Y. Jo, and H. Bahn, “Integrated scheduling of real-time and interactive tasks for configurable industrial systems,” IEEE Transactions on Industrial Informatics, vol. 18, no. 1, pp. 631-641, 2022.
DOI: https://doi.org/10.1109/TII.2021.3067714

[3] H. Bahn, J. Kim, “Separation of virtual machine I/O in cloud systems,” IEEE Access, vol. 8, pp. 223756-223764, 2020.
DOI: https://doi.org/10.1109/ACCESS.2020.3044172

[4] O. Kwon, H. Bahn, and K Koh, “Popularity and prefix aware interval caching for multimedia streaming servers,” Proc. IEEE CIT Conference, pp. 555-560, 2008.
DOI: http://doi.org/10.1109/CIT.2008.4594735

[5] J. Kim and H. Bahn, “Analysis of smartphone I/O characteristics — toward efficient swap in a smartphone,” IEEE Access, vol. 7, pp. 129930-129941, 2019.
DOI: https://doi.org/10.1109/ACCESS.2019.2937852

[6] S. Lim, H. Bahn, “Characterizing file accesses in android applications and caching implications,” IEEE Access, vol. 9, pp. 150292-150303, 2021.
DOI: https://doi.org/10.1109/ACCESS.2021.3125779

[7] H. Bahn, H. Lee, S. Noh, S. Min, and K. Koh, “Replica-aware caching for web proxies, Computer Communications, vol. 25, no. 3, pp. 183-188, 2002.
DOI: https://doi.org/10.1016/S0140-3664(01)00365-6

[8] J. Choi, S. Noh, S. Min, Y. Cho, “An implementation study of a detection-based adaptive block replacement scheme,” Proc. USENIX Annual Technical Conf., pp. 239-252, 1999.

[9] T. Johnson and D. Shasha, “2Q: a low overhead high performance buffer management replacement algorithm,” Proc. 20th ACM Conf. on Very Large Databases (VLDB), pp. 439-450, 1994.

[10] S. Bansal, D. S. Modha, “CAR: clock with adaptive replacement,” Proc. USENIX Conf. on File and Storage Technologies (FAST), 2004.