electronics

MDPI

Article

File Type and Access Pattern Aware Buffer Cache Management for Rendering Systems

Donghee Shin [1,†], Kyungwoon Cho [2,†] and Hyokyung Bahn [1,*]

[1] Department of Computer Engineering, Ewha University, Seoul 03760, Korea; veststar1@gmail.com
[2] Embedded Software Research Center, Ewha University, Seoul 03760, Korea; cezanne@ewha.ac.kr
* Correspondence: bahn@ewha.ac.kr; Tel.: +82-2-3277-4247
† These authors contributed equally to this work.

Received: 3 December 2019; Accepted: 12 January 2020; Published: 15 January 2020

Abstract: Rendering is the process of generating high-resolution images by software, which is widely used in animation, video games and visual effects in movies. Although rendering is a computation-intensive job, we observe that storage accesses may become another performance bottleneck in desktop-rendering systems. In this article, we present a new buffer cache management scheme specialized for rendering systems. Unlike general-purpose computing systems, rendering systems exhibit specific file access patterns, and we show that this results in significant performance degradation in the buffer cache system. To cope with this situation, we collect various file input/output (I/O) traces of rendering workloads and analyze their access patterns. The results of this analysis show that file I/Os in rendering processes consist of long loops for configuration, short loops for texture input, random reads for input, and single-writes for output. Based on this observation, we propose a new buffer cache management scheme for improving the storage performance of rendering systems. Experimental results show that the proposed scheme improves the storage I/O performance by an average of 19% and a maximum of 55% compared to the conventional buffer cache system.

Keywords: rendering; buffer cache; file I/O; loop; random access

1. Introduction

Rendering is the process of generating high-resolution photorealistic or non-photorealistic 2D images from 2D or 3D models [1–3]. Rendering is widely used in animations, video games and visual effects in movies, where more than 24 frames should be generated per second. Based on input files consisting of shape, contrast, texture, etc., rendering software performs a large amount of computation, which takes several hours or even a couple of days to generate a single frame. Thus, it is common to use a dedicated cluster machine rather than a desktop PC for rendering processes.

However, due to the recent advances in multi-core and many-core technologies, the computation unit is becoming increasingly faster. This allows new types of applications using interactive rendering or real-time volume rendering, where rendering for a single frame is completed within a few seconds. Also, rendering is increasingly performed in desktop PCs by installing open source software like Blender [2]. That is, users can utilize their desktop as a general-purpose computer system during usual days, but can also use it as a rendering machine if necessary. This is in line with recent trends, such as personalized YouTube broadcasting, where creating high quality contents through rendering is no longer the domain of the expert. As rendering is not a frequent job for non-experts, it is difficult to equip expensive dedicated machines, and thus desktop rendering will be increasingly useful.

Although rendering is a computation-intensive job, we observe that storage access may become another performance bottleneck in desktop rendering systems. This is because storage is five to six orders of magnitude slower than computation units, but the memory capacity in desktop is limited to

resolve this issue [4]. Unlike dedicated rendering machines that have the enough memory capacity for loading entire data files simultaneously, desktop rendering relies on the buffer cache provided by the operating system.

Buffer cache stores requested files read from storage in a certain part of memory area, which allows fast accesses of the same file data in subsequent requests [5–8]. As the size of the buffer cache is limited, we need to select an eviction victim in order to store new data. In this article, we find out that the buffer cache in general purpose computer systems is not efficient when executing rendering software, and propose a new buffer cache management technique to solve this problem.

This study extracted various file I/O traces while executing rendering software and analyzed them. By so doing, we observed three specific I/O access patterns that occur due to the characteristics of rendering procedures. The first is the sequential access pattern that appears while reading the configuration files at the launch time of the rendering software. Although the files are accessed only once throughout the execution, it forms long loop references as they are re-referenced when the program is executed again. The second appears while reading the texture input files for rendering, which is the mixture of short loops and random references. The third appears while writing the result of the rendering to output files, which forms a sequential write-once reference.

We show that such reference characteristics of rendering file accesses significantly degrade the performance of the buffer cache in general-purpose systems. Based on this observation, we propose a new buffer cache management scheme for improving the storage performance of rendering systems. Through experiments using various rendering traces, we show that the proposed buffer cache management scheme has the effect of improving the storage I/O performance by an average of 19% and a maximum of 55%, compared to the conventional system.

The remainder of this article is organized as follows. Section 2 summarizes related works of this research. Section 3 presents the motivation for this research. Section 4 describes a new buffer cache management scheme for rendering systems. In Section 5, we show the performance evaluation results to assess the effectiveness of the proposed scheme. Finally, we conclude this article in Section 6.

2. Related Works

As the wide speed gap between memory and storage has been the major performance bottleneck in computer systems, there have been various studies to improve the performance by making use of the buffer cache [9]. Because of the mechanical movement of the disk head, the access latency of HDD (hard disk drive) has been limited to tens of milliseconds, which is five to six orders of magnitude slower than DRAM (dynamic random access memory).

To buffer this speed gap, studies on buffer cache mainly consider the improvement of the hit ratio. The least recently used (LRU) replacement algorithm is the most commonly used algorithm to achieve this goal [10]. LRU sorts blocks in the buffer cache based on the last reference time and discards the block referenced longest time ago if a free cache block is needed. LRU considers the recency of references, which assumes that more-recently referenced data is more likely to be referenced again soon. As LRU is simple to implement and it is efficient in various domains, it is widely used. However, paging systems have difficulty in adopting pure LRU as monitoring every page reference is not an easy matter in virtual memory systems.

To relieve this problem, second-chance algorithms have been introduced for paging systems [11]. The second-chance algorithm maintains a reference bit for each page in memory; the reference bit of a page is set to 1 upon the reference of a page and periodically reset to 0 in order to reflect the aging of old references. If free memory space is needed, the second chance algorithm searches the circular list of pages and discards the first page whose reference bit is 0.

LRU and the second-chance algorithm consider the recency of the last reference time but they have weaknesses in that the frequency of the block/page is not considered. The least-frequently used (LFU) replacement algorithm considers the reference frequency of blocks in the cache rather than recency. Specifically, LFU evicts the block with the least number of references if free blocks are necessary.

To precisely predict references in the future, algorithms using both the frequency and the recency information have been introduced. For example, LRU-k selects the victim block by utilizing the recency of the k-th recent reference [12,13]. This has the effect of considering the frequency of references during the last k references. LRU-k also has the effect of aging since it can distinguish recent history and old history in terms of the reference frequency, and thus it is different from LFU.

The least recently/frequently used (LRFU) replacement algorithm also exploits both the frequency and the recency information of block references [14,15]. Each block in LRFU has the combined recency and frequency (CRF) value, and it is used to evaluate the value of each block for eviction.

The adaptive replacement cache (ARC) algorithm is another buffer cache replacement algorithm that considers both the frequency and the recency information, but it can adaptively control the effect of the two properties [16]. Specifically, ARC uses two block lists sorted by their reference time. The first list L1 maintains single-accessed blocks within a certain time window while the second list L2 maintains multiple-accessed blocks for considering frequency. For adaptive control of the two lists, ARC makes use of a control parameter, which puts weight either on frequency or recency. Similar to the concept of the shadow area that will be introduced in this article, ARC uses the shadow lists B1 and B2 for L1 and L2. Shadow lists maintain the metadata of recently evicted blocks from the buffer cache without maintaining their actual data. If there are frequent references in the shadow list, ARC increases the corresponding real list by adjusting the control parameter.

The clock with adaptive replacement (CAR) algorithm has the same idea of ARC for controlling the effect of the frequency and the recency adaptively, but its overhead is less than ARC [17]. Actually, CAR is the second chance algorithm version of ARC, implemented by simplifying the list manipulation. CAR uses two circular lists T1 and T2. T1 maintains single-accessed blocks within a certain time window while the T2 maintains multiple-accessed blocks for considering frequency. CAR adaptively controls the capacity of T1 and T2 by making use of the shadow areas of T1 and T2.

The two queue (2Q) replacement algorithm considers both the frequency and the recency information similar to LRU-k, but it evicts cold blocks faster than LRU-k by making use of two priority lists [18]. 2Q inserts the first-referenced blocks in a special area $A1_{in}$, which is managed by the FIFO algorithm. When a block in $A1_{in}$ is referenced, it moves to the main area A_m, which is managed by the LRU algorithm. If a block in $A1_{in}$ is not referenced again, it is considered as a cold block, and thus removed from $A1_{in}$ and its metadata is maintained in the shadow area $A1_{out}$. If a block in $A1_{out}$ is referenced, it moves directly to A_m. The time overhead of the 2Q algorithm is constant for each operation, which is lower than the LRU-k algorithm.

The multi-queue (MQ) replacement algorithm is proposed for the multi-level hierarchical cache systems [19]. Specifically, the authors show that the second level buffer cache exhibits significantly different reference patterns from the original non-hierarchical buffer cache. By considering this, multiple LRU lists are employed to manage blocks in the cache and a shadow area is adopted for blocks discarded recently. When a block in the buffer cache is requested, it is eliminated from the present LRU list and is inserted to a higher priority LRU list based on the reference frequency. When a block not existing in the buffer cache is requested, MQ removes the first block in the lowest LRU list to make free space. When the requested block is in the shadow area, the frequency information of the block is loaded, and the block is inserted to an appropriate LRU list by considering its frequency information.

The dual locality (DULO) cache-replacement algorithm considers the spatial locality as well as the temporal locality [20]. In particular, DULO manages blocks from adjacent disk positions as the sequence concept, and victim candidates are determined by sequence units. As seek time is dominant in HDD accesses, reading a large number of adjacent blocks together does not incur significant additional cost. DULO determines the eviction victims by making use of the size of the sequence as well as the reference recency information.

The unified buffer management (UBM) scheme performs on-line detection of block reference patterns and partitions the buffer cache space for different reference patterns [21]. UBM detects the reference patterns of blocks as sequential, loop, and others. Then, it allocates the cache space to the

detected reference patterns by considering the contribution of the allocated space. Once the cache spaces are allocated, the replacement algorithm is determined for each allocated area by considering the reference patterns.

The low inter-reference recency set (LIRS) replacement algorithm adopts the inter-reference recency concept to determine the eviction victim [22]. In particular, LIRS categories blocks in the cache as high inter-reference recency (HIR) blocks and low inter-reference recency (LIR) blocks. LIR implies that blocks are frequently referenced, whereas HIR implies not frequently referenced. Therefore, a block requested in the LIR category still remains in the LIR category, whereas a block in the HIR category can be promoted to the LIR category if it is frequently referenced. If free space is needed, blocks in the HIR category is preferentially discarded because they are relatively infrequently referenced blocks.

The clock for read and write (CRAW) policy analyzes the characteristics of read and write operations separately in memory page references and allocates memory space for each operation based on the effectiveness of the allocation [23,24]. CRAW considers the characteristics of asymmetric operation costs in read and write operations and also workload density of each operation.

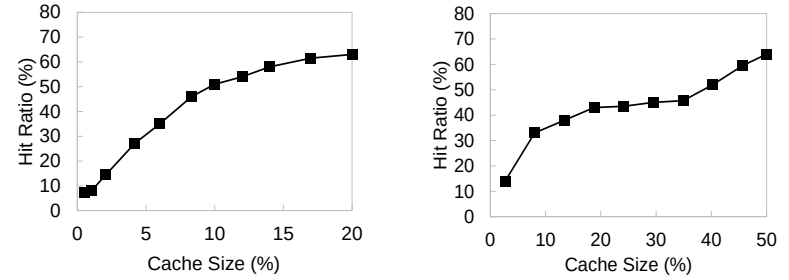
3. Motivations

3.1. Buffer Cache Performance of Rendering Systems

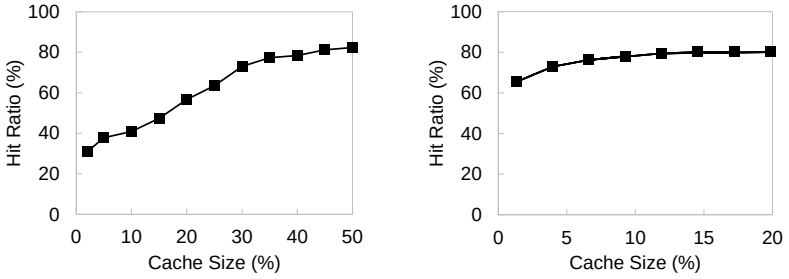
In this section, we compare the buffer cache performances of general-purpose computer systems and rendering systems, and show that traditional buffer cache designed for general-purpose systems does not work well for rendering workloads.

Figure 1 shows the performance of the buffer cache as a function of the cache size when executing general-purpose workloads on a typical buffer cache system using the least recently used (LRU) algorithm. We present the *x*-axis of the graphs in a relative scale in order to consider the different density of each workload appropriately. Note that a cache size of 100% in the relative scale is an unrealistic condition where the complete footprint of file I/Os can be cached at the same time, not incurring any cache replacement. This is equivalent to an infinite cache capacity, where all algorithms perform the same. Thus, the cache capacity in a practical configuration is usually the size less than 50% as replacement essentially happens once the system has been warmed up in any kinds of computer systems except real-time systems. The workloads used in this experiment are obtained from previous buffer cache studies: OLTP [16], Multi3 [21], Sprite [14], and DB2 [18]. Specifically, OLTP contains references to a CODASYL database, which consists of 13,208,930 references to 3,153,310 unique blocks [16]. Multi3 is a mixed workload obtained by concurrently executing cpp, gnuplot, glimpse, and postgres, which has 30,241 references to 7,453 unique blocks [21]. Sprite is a file system trace from the Sprite network file system and we use the workstation client 53 trace, which has 141,223 references to 19,990 unique blocks [14]. DB2 is a database trace obtained from a commercial installation of DB2 and contains 500,000 references to 75,514 unique blocks [18]. As shown in the figure, the buffer cache performance of general-purpose workloads is gradually improved as the cache size increases.

Now, let us examine the results for the rendering workloads. We execute Blender, a popular open-source rendering software, and capture the file access traces for 4 rendering workloads. Then, we perform trace-driven simulations by replaying these traces. The characteristics of the traces captured are summarized in Table 1, and will be explained later in Section 5. Figure 2 shows the results for this experiment with the same configurations of Figure 1. Unlike the curves in Figure 1 that gradually increase, the graphs of Figure 2 form almost horizontal lines consistently before the steep slope appears at the right-end side of the graphs. That is, buffer cache in rendering systems cannot improve its performance until the cache size becomes sufficient to accommodate all the requested data.

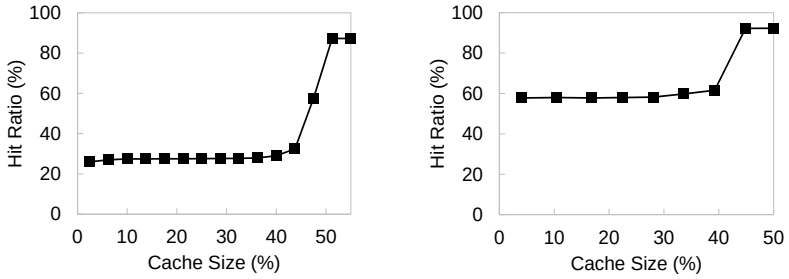


(**a**) OLTP (**b**) Multi3

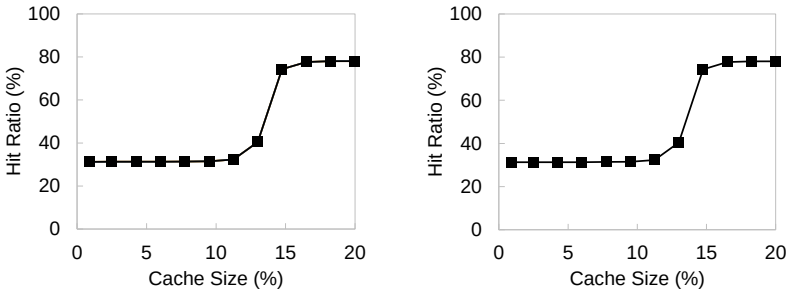


(**c**) Sprite (**d**) DB2

Figure 1. Buffer cache performance in general purpose systems.



(**a**) Classroom (**b**) Wallslam



(**c**) Caminandes (**d**) Dweep

Figure 2. Buffer cache performance in rendering systems.

Table 1. Characteristics of traces used in experiments.

	Total # of References	Total # of Blocks
Caminandes	1,562,471	168,515
Dweep	1,939,659	576,752
Classroom	3,308,248	361,951
Wallslam	8,876,286	356,598

This implies that it is not appropriate to use the buffer cache of general-purpose computer systems as it is for rendering systems. This article aims to design a buffer cache management scheme suitable for rendering systems so that the performance is gradually improved as the cache size increases. To do so, we analyze the file access characteristics of the rendering systems and manage the buffer cache by considering the effectiveness of the management algorithm on the given workload characteristics.

Previous studies also tried to consider the workload characteristics in buffer cache management. The unified buffer management (UBM) scheme performs on-line detection of the file access patterns and allocates the cache space based on the performance improvement effect of each pattern [21]. However, the target of UBM is for general purpose systems, and thus it has overhead in on-line detection of reference patterns and re-arrangement of the buffer cache space for various mixed workloads as time progresses. Specifically, in order to allocate the buffer cache space, UBM maintains a detection table in memory and updating the information, and then changes the allocation of the buffer cache space according to the evolution of workloads, which incurs additional time and space overhead.

Other algorithms such as LRU-k [13], LRFU [14], ARC [16], CAR [17], 2Q [18], and MQ [19] also consider the file access characteristics. They examine whether the workloads are more affected by recency or frequency characteristics, and adjust the cache management based on the workload characteristics. DULO [20] considers the temporal locality and spatial locality characteristics, whereas CRAW [23] considers read and write characteristics. However, all of these algorithms only consider the probabilistic characteristics, not taking into account regular patterns like repeated references as observed in rendering systems. For this reason, if a workload larger than the buffer cache size is executed, thrashing happens as shown in Figure 2 and the I/O performances cannot be improved at all until the cache size becomes sufficient to accommodate total workload footprint.

Unlike previous studies that target all kinds of workloads, in which probabilistic reference characteristics like recency and frequency are mainly exploited, this study focuses on the rendering workloads and thus analyzes the distinct characteristics of rendering file I/Os, such as long loops, short loops, and single writes for each file type in advance, then exploits them in judicious buffer cache management. By so doing, our algorithm improves the I/O performance of rendering systems regardless of the workload size and the memory size of the target machine, which is not possible in previous solutions. Although UBM considers loop patterns, it also targets all kinds of workloads, and thus requires the overhead of heavy on-line detection in terms of space and time.

3.2. I/O Time in Desktop Rendering

In this section, we analyze the relation between the computing time and the I/O time while our rendering is conducted. As shown in Figure 3a, CPU and I/O bursts appear repeatedly in the rendering process. That is, for each period, interleaved executions of CPU and I/O first happen, which we call the interleaving phase, and then a long CPU burst follows, which we call the computation phase.

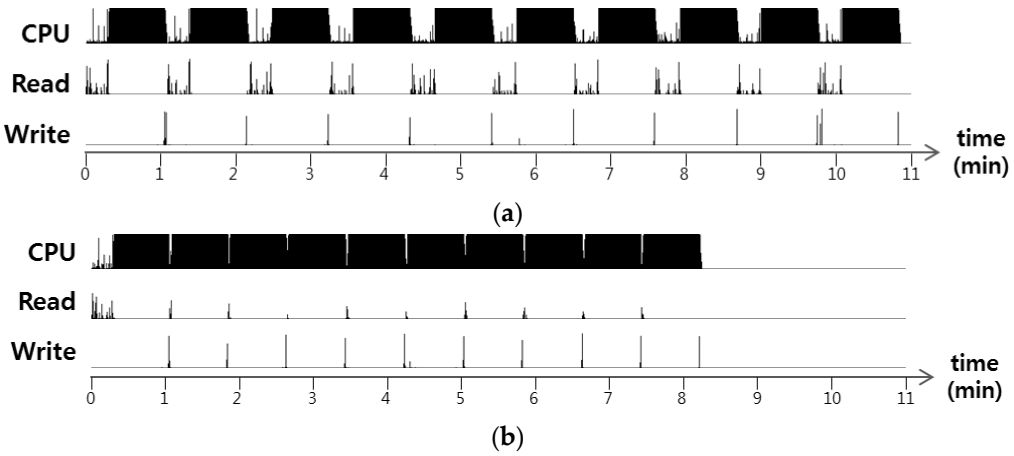


Figure 3. CPU and I/O activities in desktop rendering. (**a**) Results for original system configurations. (**b**) Results when all file data are simultaneously loaded in memory.

In traditional photorealistic rendering, the time scale for the computation phase is 100× or more than that of the interleaving phase. However, our desktop rendering shows that the measured ratio of the computation phase is only 2.5× that of the interleaving phase. This is because our configurations are set to generate rendered images quickly as we focus on the interactive rendering application. Another important observation is that the interleaving phase consistently appears in every period. This is because buffer cache uses the LRU replacement algorithm, which evicts all loop data before they are used again. To assess the necessity of efficient buffer cache management further, we conducted the same rendering experiment again by increasing the memory size enough to load all file requests simultaneously. As shown in Figure 3b, the interleaving phase almost disappears from the second period, reducing the execution time by 26%. One may think that this can be realized simply by increasing the memory size. However, this procedure will not work if a workload larger than the given memory size is executed. In that case, the interleaving phase still remains as the same time scale, and we cannot benefit from the buffer cache at all. Thus, our conclusion is that judicious buffer cache management is necessary for desktop rendering systems.

4. Buffer Cache Management for Rendering Systems

In order to design the buffer cache of rendering systems, this article first extracts the file block reference traces while rendering is performed, and analyzes them. Figure 4 shows the blocks referenced over time in the extracted trace, by file type. In the figure, the x-axis is the logical time, which is incremented by one each time a block is referenced, and the y-axis represents the block number.

When we analyze the block reference patterns based on file types, the configuration files are first read during the launch time of the rendering software as shown in Figure 4. Then, the texture input files are read for the rendering of each frame. Finally, the rendering results are written to output files each time rendering a frame is completed. We observe that these characteristics are commonly found in any rendering tasks.

Now, let us see the detailed reference patterns for each file types. As shown in Figure 4, the configuration files are read-only once in every program execution, so it forms sequential accesses but can also be considered as a long loop reference. Reading texture input files shown in the figure is mostly composed of short loop references, but also have some random references. Writing rendering result to output files forms a sequential write-once reference pattern. Since these reference patterns can be identified through the file format, this article tries to design a buffer cache management scheme specialized for rendering systems based on the reference patterns of file types.

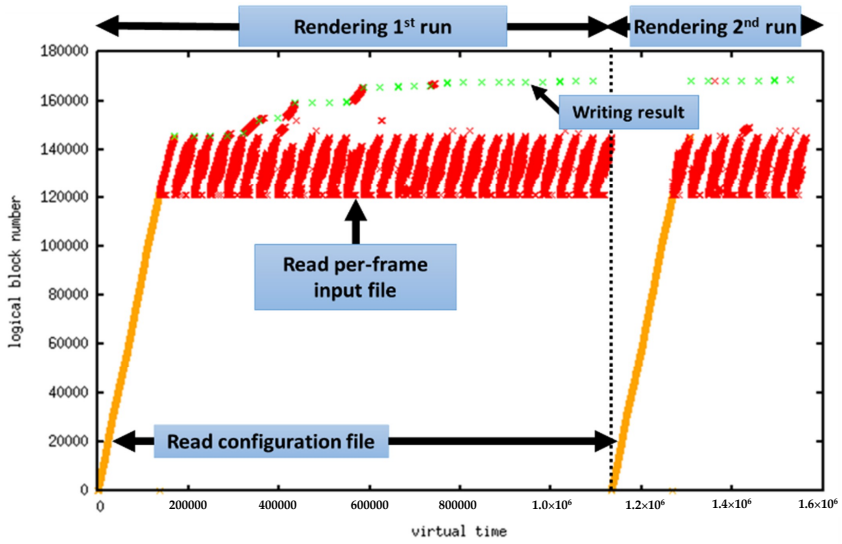


Figure 4. Block reference patterns in rendering process.

Meanwhile, when various reference patterns are mixed, it is effective to separate the buffer cache space for each reference pattern and use an appropriate management technique. This article allocates cache space based on the performance improvement effect of each reference pattern during rendering processes and uses appropriate management techniques. First, blocks that are referenced only once and never accessed again, which we call single-reference blocks, are not cached in our buffer cache because there is no performance improvement effect when caching them. In the case of loop references, we allocate cache space based on the period of the loops, which reflects the gain of the loop per unit time when it is cached. Unlike previous studies that have large overhead in the on-line detection of file access patterns [9], this article makes use of the file-access characteristics of the rendering processes ascertained in advance for allocating and managing cache spaces.

Figure 5 shows the data structure of the proposed buffer cache. The cache space is composed of a configuration files area C1 for long loop references, a texture files area T1 for short loop references, and a random reference area O1. We do not allocate cache area for output files as they are not used again. Instead, we buffer them for writing to storage and evict promptly after being flushed to storage. The allocated areas C1, T1, and O1 are resized according to the performance contribution of each area as the density of block references are varied for different rendering cases.

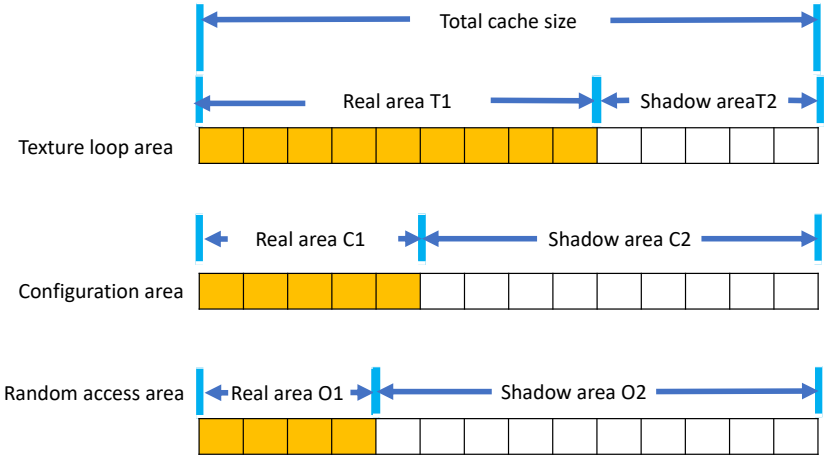


Figure 5. The data structure of the proposed buffer cache.

Our scheme employs shadow areas (C2, T2, and O2) to evaluate and adjust the size of real areas (C1, T1, and O1) as shown in Figure 5. The shadow area indicates how much performance improvement can be expected when the size of the real area is increased. In particular, shadow areas only maintain the metadata of recently evicted blocks from the corresponding real areas without maintaining their actual data. By maintaining information that the block has been recently removed from the real area due to space limitations, we can predict the effect that extending the area would have on performance. If there are frequent references in the shadow area, our scheme extends the corresponding real area to reduce the number of storage accesses. As the total cache size is fixed, the size of some other area whose relative contribution is small is decreased. For example, if blocks in T2 are frequently referenced, our scheme extends the T1 area to accommodate more blocks, and then shrinks C1 area. Also, the size of the shadow area T2 is decreased. This is because the hit ratio for the entire buffer cache can be predicted if the sum of the real areas and the shadow areas is equal to the total cache size. The overhead of maintaining shadow areas is known to be very small as it only maintains the metadata of blocks, which is less than 64 bytes of information including pointers and a block identifier, whereas each of the actual blocks has 4 KB of data [14,18,22,25].

Now, let us see the details of the proposed algorithm. When a new file block needs to be cached, our scheme first checks the file type of the referenced block. If it is the output file, we insert it into the buffering area, which will later flushed to storage and then eliminated. Otherwise, blocks are stored in the texture area (T1) or the configuration area (C1) based on the extension of the file. As blocks in T1 and C1 exhibit loop reference patterns, if free space is necessary, the most recently used (MRU) block is evicted from the area as the MRU replacement algorithm is known to the optimal in loop references [21].

Meanwhile, if the evicted block does not exhibit loop reference but random reference patterns, early eviction by MRU may cause performance degradation. Therefore, in this study, instead of immediately removing it from the buffer cache, we moved it to the random reference area O1, but the metadata still remains in the shadow area T2. As the size of the shadow area is also limited, if the space in T2 is exhausted, the metadata of the block that has entered the shadow area the longest time ago is deleted. When replacement is necessary in the O1 area, we select the victim block by the LRU (least recently used) replacement algorithm. This is because LRU is the most commonly used algorithm that shows reasonably good performance for non-regular reference patterns [21].

If the requested block already exists in the buffer cache (or shadow area), it can be classified into three categories. The first is that it exists in either T1 or C1 area. In this case, the block is a part of the loop, which is maintained in the cache, and is moved to the lowest priority position in the area. Secondly, the requested block may exist in the O1 area. In this case, the block is moved to the highest priority position in the LRU list. The third case is that the metadata of the requested block exists in the shadow area. Note that this may occur simultaneously with other cases. For example, the requested block can be found in the O1 area, but its metadata can also be found in the T2 area. In this case, the size of the shadow area is decreased, and the size of the real area is increased. Then, in order to balance the cache size throughout the all areas, the size of another area is adjusted. We can classify this process into three cases. First, if the requested block is found in the T2 shadow area, the size of T1 is increased, and then the size of T2 is decreased accordingly. In addition, the size of C1 is decreased and the size of C2 is increased. Second, if the requested block is found in the C2 shadow area, the size of C1 is increased, and then the size of C2 is decreased. Then, the same size of T1 is decreased and T2 is increased accordingly. Third, if the requested block is found in the O2 shadow area, the size of O1 is increased, and then the size of O2 is decreased. As this is the case that some texture file not forming a loop is found, we decrease T1 and increase its shadow area T2. Figure 6 shows the conceptual flow of the proposed algorithm.

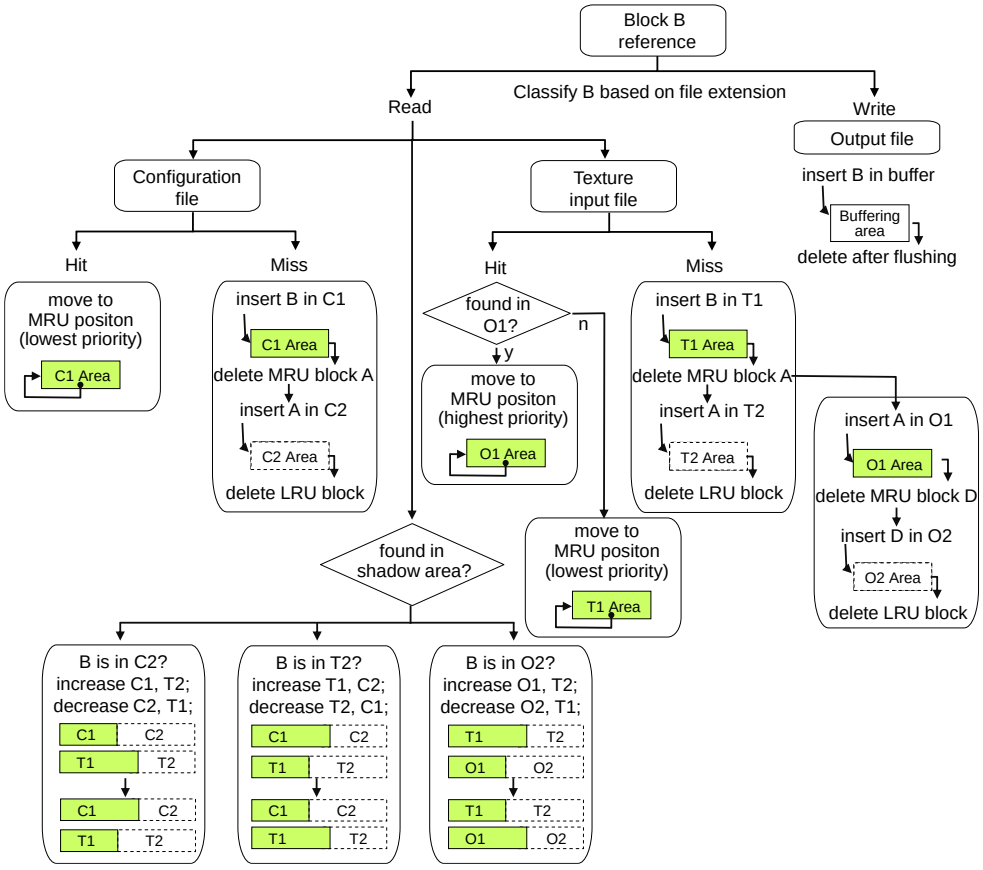


Figure 6. A conceptual flow of the proposed algorithm.

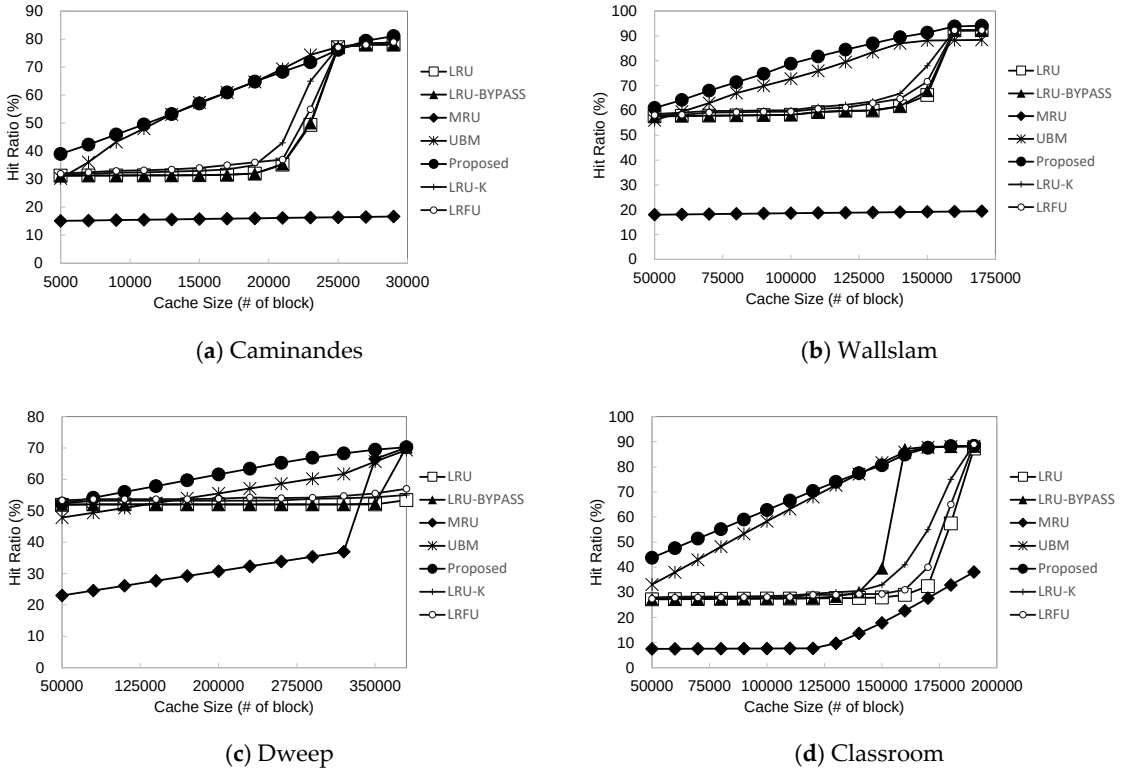
5. Experimental Results

In order to evaluate the performance of the proposed algorithm, we extracted I/O traces while executing rendering software and performed simulation experiments by replaying them. Trace extraction was performed by randomly selecting the four open rendering projects provided by Blender, which is open-source rendering software. The system we executed the rendering workload is a desktop PC consisting of Intel i7-7700 4 cores, 8 threads, 3.6 GH CPU with 8 KB of cache memory, 4 GB of main memory, and 192 MB/s SATA HDD. Note that less than 1 GB of the total main memory 4 GB can be purely used for application's buffer cache in this configuration. The operating system we used was Linux Ubuntu 16.04. Note that the file access traces were extracted with this machine at the system call layer while executing the rendering workloads. Table 1 shows the characteristics of the extracted traces. Classroom is a demonstration file that virtually navigates a school classroom [26]. All the others are animation clips as a part of production files of open movie projects. Wallslam [27], Caminandes [28], and Dweep [29] are from the films titled "The Daily Dweebs", "Caminandes Llamigos", and "Agent327", respectively. The time we executed rendering for extracting these traces is 363 s, 480 s, 628 s, and 711 s, for Caminandes, Dweep, Classroom, and Wallslam, respectively.

We compare the proposed algorithm with LRU, LRU + bypass, MRU, UBM, LRFU, and LRU-k. LRU + bypass uses LRU as its replacement algorithm but it does not cache the output file, which will never be accessed again. In the proposed algorithm, the initial configuration of the texture area and the configuration area is set by the ratio of 9 to 1 by considering the workload characteristics and the size of the random-access area to 64 KB. This size is adaptively changed when the contribution of each area varies due to the evolution of the workload.

Figure 7 shows the cache hit ratio of the proposed algorithm in comparison with the other algorithms as the size of the buffer cache is varied for the four workloads described in Table 1. As

shown in the figure, the proposed algorithm outperforms the others in most cases. As rendering I/O has many loop patterns, LRU does not show good performance if the cache size is not large enough to maintain the entire loops. This is because blocks are already evicted from the cache at the time of re-reference if the cache size is not large enough. LRU exhibits good performance only when the buffer cache size is sufficient to accommodate the entire loops. Thus, increasing the cache size is not effective until it exceeds a certain level.



(**a**) Caminandes (**b**) Wallslam

(**c**) Dweep (**d**) Classroom

Figure 7. Performance comparison with other buffer cache management schemes.

LRU + bypass performs better than LRU. In particular, as single-access blocks are prevented from entering the cache, it shows the effect of increasing the cache space, which subsequently improves the performance earlier than LRU as the cache size increases. However, LRU + bypass also does not consider the characteristics of loop reference patterns, and thus it does not show significant performance improvement.

When comparing our scheme with LRU-K and LRFU, our scheme performs consistently better than these two schemes. Although these two schemes perform slightly better than LRU, they do not resolve the issue of the performance thrashing problem when the cache size is not sufficiently large to accommodate the entire footprint of the workload.

MRU showed the worst performance among the algorithms we considered. The trace used in our experiments has some specific characteristics, such as the appearance of many short loops after long loops. In our traces, short loops account for 78% of all references, but MRU already exhausts the buffer cache space by long loops, and thus performance is degraded significantly by short loops that incur cache misses.

Since UBM quickly removes single-access blocks from the buffer cache and allocates the cache space based on the performance improvement effect of each reference pattern, it can lead to cache hits for short loops and random reference blocks. However, UBM incurs an on-line detection overhead for identifying and classifying reference patterns and calculating the expected gain of each reference pattern for adjusting the cache spaces allocated. When considering the overall experimental results in Figure 7a–d, the performance improvement of the proposed scheme is 19% on average and up to

55% in comparison with LRU, which is the most commonly used buffer cache replacement algorithm. When comparing with LRU + bypass, MRU, and UBM, our scheme improves the file I/O performances by 15%, 45%, and 3%, on average, and up to 47%, 74%, and 10%, respectively.

Our experiments showed that existing algorithms also perform well in large cache sizes. However, such results happen only when the cache size becomes excessively large, not reflecting practical situations. Specifically, the machine we experimented rendering has the memory size of 4 GB, of which less than 1 GB can be used as the buffer cache space for user applications. Except for the UBM scheme, existing algorithms did not perform well in this range. Note again that the performance gain of the proposed policy is proportional to the buffer cache size regardless of the workload size and the memory size of the target machine.

6. Conclusions

In this article, we designed a buffer-cache-management scheme specialized for desktop rendering systems. Unlike general-purpose computing systems, the rendering system shows unique file I/O characteristics, and this results in a significant performance degradation under the existing buffer cache management scheme. To cope with this situation, this article analyzed the access patterns of file I/O operations in rendering systems. We found that file I/Os in rendering processes consist of long loops, short loops, random accesses, and sequential writes. Based on these observations, we proposed a cache space allocation and management scheme, which achieves the performance improvement of 19% on average and up to 55% compared with the existing buffer cache system.

Author Contributions: D.S. designed the architecture and algorithm, K.C. performed the experiments. H.B. supervised the work and provided expertise. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2016R1A6A3A11930295 and IITP 2018-0-00549).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Patil, G.V.; Deshpande, S.L. Distributed rendering system for 3D animations with Blender. In Proceedings of the IEEE International Conference on Advances in Electronics, Communication and Computer Technology, Pune, India, 2–3 December 2016; pp. 91–98.
2. Kent, B. *3D Scientific Visualization with Blender*; Morgan & Claypool Publishers: San Rafael, CA, USA, 2015.
3. Rossl, C.; Kobbelt, L. Line-art rendering of 3D-models. In Proceedings of the 8th IEEE Pacific Conference on Computer Graphics and Applications, Hong Kong, China, 5 October 2000; pp. 87–96.
4. Lee, E.; Bahn, H. Caching Strategies for High Performance Storage Media. *ACM Trans. Storage* **2014**, *10*, 11. [CrossRef]
5. Kim, D.; Bahn, H. Exploiting Write-Only-Once Characteristics of File Data in Smartphone Buffer Cache Management. *Pervasive Mob. Comput.* **2017**, *40*, 528–540. [CrossRef]
6. Lee, E.; Kang, H.; Bahn, H.; Shin, K. Eliminating Periodic Flush Overhead of File I/O with Non-volatile Buffer Cache. *IEEE Trans. Comput.* **2016**, *65*, 1145–1157. [CrossRef]
7. Lee, E.; Bahn, H.; Noh, S. A Unified Buffer Cache Architecture that Subsumes Journaling Functionality via Non-volatile Memory. *ACM Trans. Storage* **2014**, *10*, 1.
8. Park, J.; Lee, E.; Bahn, H. DABC-NV: A buffer cache architecture for mobile systems with heterogeneous flash memories. *IEEE Trans. Consum. Electron.* **2012**, *58*, 1237–1245. [CrossRef]
9. Pai, V.S.; Druschel, P.; Zwaenepoel, W. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Trans. Comput. Syst. TOCS* **2000**, *18*, 37–66. [CrossRef]
10. Denning, P.J. The Working Set Model for Program Behavior. *Commun. ACM* **1967**, *15*, 323–333.
11. Corbato, F.J. A paging experiment with the multics system. In *Honor of P. M. Morse*; MIT Press: Cambridge, MA, USA, 1969; pp. 217–228.

12. O'Neil, E.J.; O'Neil, P.E.; Weikum, G. An optimality proof of the LRU-K page replacement algorithm. *J. ACM* **1999**, *46*, 92–112. [CrossRef]
13. O'Neil, E.J.; O'Neil, P.E.; Weikum, G. The LRU-K page replacement algorithm for database disk buffering. *ACM Sigmod Rec.* **1993**, *22*, 297–306. [CrossRef]
14. Lee, D.; Choi, J.; Kim, J.H.; Noh, S.H.; Min, S.L.; Cho, Y.; Kim, C.S. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Comput.* **2001**, *50*, 1352–1361.
15. Lee, D.; Choi, J.; Noh, S.H.; Min, S.L.; Cho, Y.; Kim, C.S. On the existence of a spectrum of policies that subsumes the LRU and LFU policies. In Proceedings of the ACM SIGMETRICS Conference, Atlanta, GA, USA, 1–4 May 1999.
16. Megiddo, N.; Modha, D.S. ARC: A Self-tuning, low overhead replacement cache. In Proceedings of the USENIX International Conference on File and Storage Technologies (FAST), San Francisco, CA, USA, 31 March–2 April 2003.
17. Bansal, S.; Modha, D.S. CAR: Clock with adaptive replacement. In Proceedings of the 3rd USENIX Conference File and Storage Technologies (FAST), San Francisco, CA, USA, 31 March–2 April 2004; pp. 187–200.
18. Johnson, T.; Shasha, D. 2Q: A low overhead high performance buffer management replacement algorithm. In Proceedings of the VLDB Conferences, Santiago, Chile, 12–15 September 1994.
19. Zhou, Y.; Philbin, J.F. The multi-queue replacement algorithm for second level buffer caches. In Proceedings of the USENIX Annual Technical Conference (ATC), Princeton, NJ, USA, 29 April 2001; pp. 90–104.
20. Jiang, S.; Ding, X.; Chen, F.; Tan, E.; Zhang, X. DULO: An Effective Buffer Cache Management Scheme to Exploit both temporal and Spatial Locality. In Proceedings of the 4th Conference on USENIX Conference File and Storage Technologies (FAST), Columbus, OH, USA, 6–9 June 2005; pp. 101–114.
21. Kim, J.M.; Choi, J.; Kim, J.; Noh, S.H.; Min, S.L.; Cho, Y.; Kim, C.S. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI), San Diego, CA, USA, 23–25 October 2000.
22. Jiang, S.; Zhang, X. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM Sigmetrics Perform. Eval. Rev.* **2002**, *30*, 31–42. [CrossRef]
23. Lee, H.; Bahn, H. Characterizing virtual memory write references for efficient page replacement in NAND flash memory. In Proceedings of the IEEE MASCOTS Conferences, London, UK, 21–23 September 2009; pp. 1–10.
24. Lee, H.; Bahn, H.; Shin, K.G. Page Replacement for Write References in NAND Flash Based Virtual Memory Systems. *J. Comput. Sci. Eng.* **2014**, *8*, 157–172. [CrossRef]
25. Bahn, H.; Noh, S.H. Characterization of Web Reference Behavior Revisited: Evidence for Dichotomized Cache Management. *Lect. Notes Comput. Sci.* **2003**, *2662*, 1018–1027.
26. Available online: https://download.blender.org/demo/test/classroom.zip (accessed on 28 November 2019).
27. Assets & Production Files. 2017. Available online: https://cloud.blender.org/p/dailydweebs/59e7c623f4885514c231defb (accessed on 28 November 2019).
28. Available online: https://cloud.blender.org/p/caminandes-3/56b474adc379cf4aeeb79a0f (accessed on 28 November 2019).
29. Shot File Selection. 2017. Available online: https://cloud.blender.org/p/agent-327/591aab07bb3ea14149532ed2 (accessed on 28 November 2019).