IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received August 23, 2021, accepted October 6, 2021, date of publication October 8, 2021, date of current version October 15, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3119077

Design and Implementation of a Distributed Versioning File System for Cloud Rendering

KYUNGWOON CHO AND HYOKYUNG BAHN, (Member, IEEE)
Department of Computer Engineering, Ewha University, Seoul 120-750, South Korea

Corresponding author: Hyokyung Bahn (bahn@ewha.ac.kr)

This work was supported in part by ICT Research and Development Program of MSIP/IITP (Developing System Software Technologies for Emerging New Memory That Adaptively Learn Workload Characteristics) under Grant 2019-0-00074, in part by (Development of Data Improvement and Dataset Correction Technology Based on Data Quality Assessment) under Grant 2020-0-00121, and in part by RP-Grant 2020 of Ewha Womans University.

ABSTRACT Rendering is widely used for visual effects in animations, video games, and movies. As the computational load in rendering workloads fluctuates greatly over time, it is attractive to use cloud infrastructures for cost-effective rendering. However, we analyze that cloud rendering has several technical challenges involved in the handling of rendering input data. In this article, we analyze the workload characteristics of popular rendering projects, and find out the following three observations. First, total size of rendering input files reaches tens to hundreds of gigabytes, and uploading these large data to cloud increases the startup latency of rendering significantly. Second, the consistency requirement of file systems in cloud rendering is complicated compared to that of traditional distributed file systems. Third, file accesses in rendering are highly skewed such that the top 20% files account for 60-80% of total accesses, whereas 40-70% are never used or used only once. Based on these observations, we design and implement a new file system for cloud rendering, which has the function of version control, on-demand fetch, and distributed cooperative caching for rendering data. This allows for minimizing data transmission overhead caused by the large input data of rendering and satisfying the rendering data consistency. Measurement studies under synthetic and real workloads show that the proposed file system performs better than the conventional uploading scheme and NFS by 55.4% and 29.5% on average, respectively.

INDEX TERMS Cloud, rendering, file system, cooperative caching, versioning file system.

I. INTRODUCTION

Rendering is the process of creating high-resolution photo-realistic or non-photorealistic images from a 2D or 3D geometric model by making use of computer software [1]–[3]. Rendering is widely used to generate scenes for animations, video games, simulations, and visual effects in movies, where a scene consists of more than 24 frames per second. Based on graphical inputs such as textures, colors, materials, lighting, and shading, rendering software performs a large number of computations that take several hours or even more than a full day to create a single scene. Thus, rendering is commonly performed on high performance dedicated machines.

However, as the computational load in rendering fluctuates greatly over time, it may exceed the capacity of the on-premise facilities as the work due date approaches. To cope with this situation, public clouds such as AWS [4] and Azure [5] can be utilized as temporary infrastructures for rendering [6]. That is, rendering users make use of their own on-premise facilities in usual time, but temporarily rent public cloud infrastructures in case the rendering workloads become instantly heavy. This has the advantage of adapting quickly to the change of computing load over time without owning expensive equipment. As rendering of each scene is an independent work, it is also possible to perform rendering on multiple cloud nodes simultaneously. Concurrent rendering on multiple virtual machines can further speed up the rendering operations.

Although cloud is being adopted in a variety of computing platforms such as virtual desktops, web hosting, and big data processing, there are some challenges when using clouds for rendering. Rendering is a computing-intensive task, but it has an I/O intensive pre-processing phase, where several co-workers generate a large amount of rendering input data. In cloud rendering, the input data created at the on-premise machine should be sent to cloud before rendering starts. There are two important issues with this input data in cloud rendering. First, the input data for rendering a single scene

The associate editor coordinating the review of this manuscript and approving it for publication was Songwen Pei.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

reaches tens to hundreds of gigabytes, and it causes heavy overhead to send them to the cloud instantly. Second, multiple versions of the rendering input data coexist due to frequent design modifications, and maintaining the consistency of these versions is complicated compared to the requirement of traditional distributed file systems.

Let us discuss the details of these two issues. As there are separate file systems in the on-premise local machine and remote cloud machines, the rendering administrator needs to manually upload the input files for cloud rendering. In this process, a smart administrator may check the files that will be actually used during the current rendering process and upload them selectively. However, in general, all input files in the on-premise machine should be transferred as it is not known in advance which files will be actually used. Thus, despite its simplicity, uploading input files may cause unnecessary data transfers and is also a burden to the rendering administrator [7]. Our analysis shows that there are 8.1-61.4% of rendering input files, which are not actually used while the rendering is performed in popular animation rendering projects [8].

A distributed file system such as NFS (network file system) can be alternatively used to resolve the aforementioned issue [9]. This is because only data to be actually used can be fetched on-demand when NFS is adopted. However, it also has much overhead in instant validation and transmission of rendering input data, which are frequently updated. Note that modifications to local source data should also be reflected on remote nodes in distributed file systems, and thus validation requests to on-premise storage is necessary even if the data have already been uploaded to the cloud.

In reality, the consistency requirement of file systems in cloud rendering is even more complicated compared to that of traditional distributed file systems. That is, although the input file in the on-premise storage has been modified, any rendering that is already in progress should use the input files generated before starting this rendering. This makes it difficult to use traditional distributed file systems such as NFS as it is for cloud rendering, because the version of the file to use depends on the start time of the rendering process even if a new version has been created. As a result, either uploading input files by the administrator or using existing distributed file systems has problems of large overhead and/or correctness problems.

Caching is an effective way to relieve the overhead of data delivery in cloud rendering. In caching, once the data has been fetched, a copy is stored and it will be reused later if the same data is requested again [10]–[12]. However, caching in rendering has distinct characteristics and is different from traditional buffer caching [13] or web caching [14]. Specifically, unlike file system buffer caching (including NFS) that makes use of per block-based caching, per file-based caching will be efficient in cloud rendering from the perspective of validation and version control. Also, caching for cloud rendering is different from web caching in that data synchronization is important in rendering as input files are frequently updated.

In this article, we analyze the workload characteristics of popular rendering projects, and present a new file system for cloud rendering. Our file system is designed as a distributed file system that synchronizes original files in on-premise storage and cached files in cloud nodes, while satisfying the semantics of rendering input. We have implemented our file system as a user-level FUSE file system [15], and the rendering software does not need to be aware of the existence of our file system. It has the function of version control, on-demand fetch, and distributed cooperative caching.

In the on-premise system side, we have implemented a file server that manages versions of rendering input files and performs the on-demand synchronization of the input files requested by cloud nodes based on the file versions maintained. In the cloud side, we have implemented a rendering cache manager, which checks the versions of the rendering input files and fetches the files necessary for the current rendering process either from on-premise storage or from one of the peer cloud nodes.

Our rendering cache manager makes use of two caching strategies specialized for cloud rendering: version-aware caching for input data consistency and cooperative caching for minimizing the data transmission overhead. The version-aware caching maintains the rendering input files based on their versions and validates the current version from on-premise storage when the rendering process requests the files. As we use cooperative caching, the rendering cache manager first tries to fetch an input file from one of the peer cloud nodes in case a valid version is cached there. If not, the file is retrieved from the on-premise storage. After fetching the input file, our rendering cache manager maintains it in the storage cache with the version information. Note that all caching process including fetching and validation in our rendering cache manager is performed in a file unit.

To assess the effectiveness of our file system implementation, we have performed measurement experiments under both synthetic and real workloads. For synthetic workload, we extract the characteristics of rendering activities from the popular animation rendering projects and generate I/O workloads. For real workload, we execute a well-known open source rendering software Blender on AWS. Our experimental results show that the proposed file system performs better than the uploading scheme and NFS by 55.4% and 29.5% on average, respectively.

The remainder of this article is organized as follows. In Section II, we explain the rendering process and the workload characteristics of rendering. Section III presents the details of the propose file system for cloud rendering. Experimental results based on measurement for evaluating the proposed file system is described in Section IV. Finally, Section V concludes this article.

II. RENDERING PROCESS AND WORKLOAD CHARACTERISTICS

A rendering process consists of a certain pipelined phases: pre-processing, rendering, and post-processing. As shown

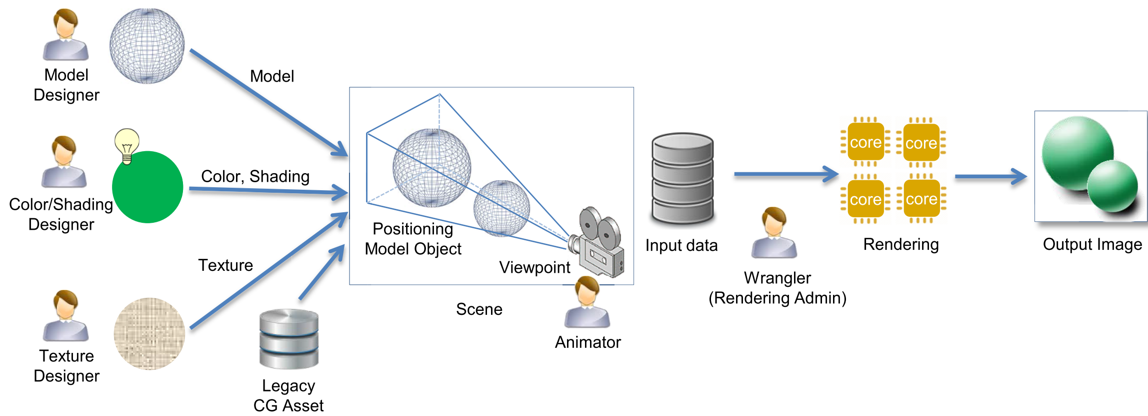


FIGURE 1. An example of a rendering process.

in Figure 1, the rendering input data are created during the pre-processing phase by a number of co-working graphic designers. In this phase, modeling, shading, texturing, and animating input data are generated by each designer making use of existing rendering assets or other designer's output. By integrating their produced works, rendering input files are generated. Then, the rendering administrator called the Wrangler collects the input files and runs the rendering software. After the long computation process of rendering, a single image file is generated as the rendering output. In the post-processing phase, the rendering administrator and the graphic designers check the created image, and if necessary, change the input files to perform the rendering again.

As rendering is performed by several co-working designers and frequent updates are essential until the final version is prepared, a lot of input file versions coexist. Rolling back to a previous version is usually performed during the rendering process, and thus it is important to keep the old version of the input files undeleted. For example, let us consider an animation scene in which a black-haired woman appears. If the rendering designer changes the color of the woman's hair from black to blonde but the original black looks better, the input files should be rolled back to perform the rendering again.

To generate a photo realistic image via rendering, the size of input files reaches hundreds of gigabytes even for a single scene. In cloud rendering, pre-processing and post-processing are performed on an on-premise machine for efficient collaboration and I/O work, and main rendering is performed on the cloud as it has heavy computations. Thus, the large input data created at the on-premise machine should be sent to cloud, which incurs heavy network overhead.

In this article, we analyze the characteristics of the rendering workloads specially focusing on the large input data by executing a representative open source rendering software Blender [16].

Figure 2 plots the CPU and I/O usage patterns while rendering a single image by Blender. As shown in the figure, rendering exhibits computing-intensive characteristics, which has the peak usage of CPU during almost all rendering

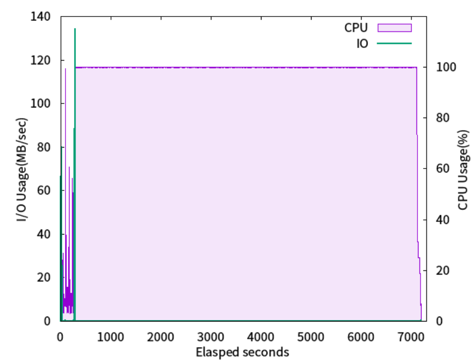
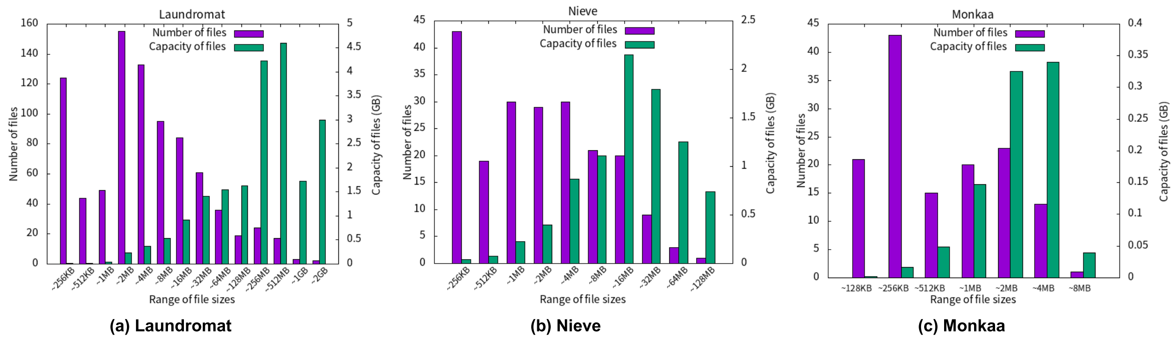


FIGURE 2. Resource utilizations of CPU and I/O devices while rendering a scene.

periods. In contrast, I/O usage is bulky only at the beginning phase of rendering. Most of the remaining process does not require I/O at all and incurs a small fraction of I/O at the end of the process for generating output. Based on such resource usage patterns of rendering, it is clear that cloud rendering will be effective if the overhead of large I/O in the early stage can be resolved.

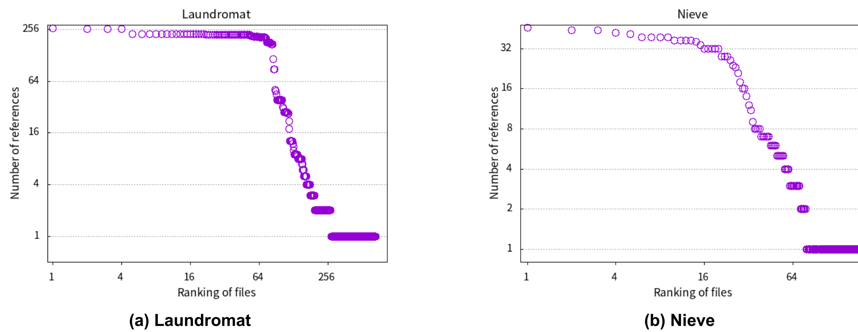
Rendering materials are valuable assets for post-production and animation studios, so it is not easy to obtain experimental data sets. Open movie projects are useful for researchers who need to analyze rendering materials at a commercial level [8]. We analyze the rendering workload characteristics of three computer animation projects: Laundromat, Nieve, and Monkaa [8]. Table 1 summarizes the rendering assets of the three movies to be analyzed. Even though these assets have been well-organized after the projects were completed, only a small portion of the rendering input files are involved in producing final output images. It is also noticeable that there exists quite a large amount of data that are never used at all in these assets. This implies that upload-based data synchronization may result in too much useless data to be transferred.

Figure 3 shows the distributions of rendering input data as the file size is varied. Although the details are varied depending on the project size and characteristics, the overall trends of the distributions are similar. Specifically, a majority



(a) Laundromat (b) Nieve (c) Monkaa

FIGURE 3. File size distributions of rendering input data for popular animation rendering projects.



(a) Laundromat (b) Nieve

FIGURE 4. Number of references that occurs on the given rankings of input files sorted by their reference count.

TABLE 1. Brief characteristics of the rendering workloads analyzed in this article.

	Laundromat	Nieve	Monkaa
# of files	841	205	139
Asset size	20.2GB	8.7GB	0.9GB
Size of rendering input	7.6GB	8.0GB	0.6GB
Size of unused files	12.6GB	0.7GB	0.3GB

of input files are relatively small, but a few large files account for the most of the total capacity.

To analyze the characteristics of the rendering input files further, we investigate the popularity distributions. Figure 4 shows the distributions of input file accesses as the popularity rankings are varied. In the figure, the x-axis represents the ranking of files sorted by their total reference count and the y-axis represents the number of references on that ranking. As shown in the figure, file accesses made by rendering input data are extremely biased, and some of the top ranking files account for most of the references. Also, there exists an inflection point where the reference count drops steeply as the file ranking is over that point. This implies that caching will be effective by maintaining hot files selectively. Also, from Figure 4 and Table 1, it is worth noting that the files of rendering asset never used or accessed only once account for 40.1-69.5% of the total capacity.

To analyze the reference bias of rendering input files more clearly, we plot the cumulative ratio of file references for the given ratio of top ranking files in Figure 5. Note that the x-axis is the ratio of referenced files sorted by their reference count and the y-axis represents the ratio of references for the given fraction of top ranked files. For example, 20% in the x-axis implies the top 20% input files, and the corresponding value in the y-axis represents the ratio of file references they made. As shown in the figure, 20% of the top ranked files account for about 80% of total references in Nieve, implying that file references in rendering mostly result from some hot files. In case of Laundromat, 20% of the top ranked files account for 60% of total references. When compared to general workloads, the skewness of rendering file accesses is very strong as it is known that 50-60% of the top ranking files account for about 80% of file accesses in general workloads of server systems [17], [18].

Analysis in this section will be used later in Section IV for generating synthetic workload models and evaluating the performance of our file system.

III. THE PROPOSED FILE SYSTEM FOR CLOUD RENDING

Two types of storage architectures for cloud rendering can be considered. The first consists of separate storage for on-premise and cloud systems, and the rendering administrator uploads input files from local to cloud for rendering, and downloads the result after rendering is finished. In this case, data synchronization should be managed by the administrator [19]. Although this approach is simple to

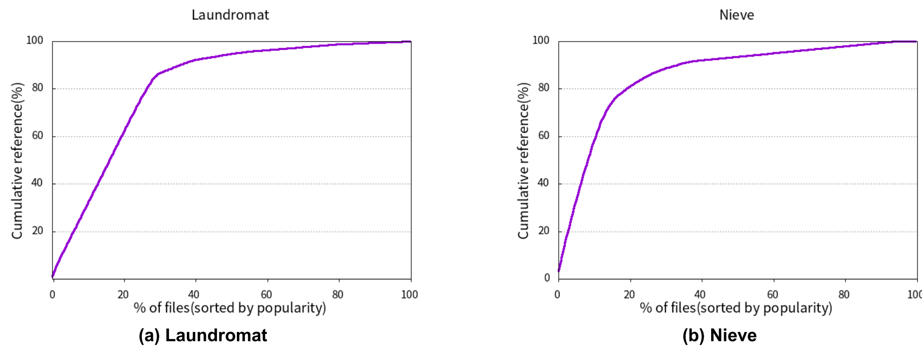


FIGURE 5. Cumulative ratio of file accesses for the given ratio of top ranked files sorted by their reference count.

implement, it needs an additional cluster file system to share data among cloud rendering nodes. Also, it is not easy for the administrator to distinguish files to be uploaded as there are too many files involved in rendering and some of them are duplicated or not used at all, which may incur unnecessary network transmission. Until the uploading of large data is completed, rendering cannot start, which incurs additional latency.

The second storage architecture for cloud rendering is to make the rendering repository as a virtualized storage. In this architecture, a virtual file system interface can be used for cloud nodes to access the on-premise rendering storage. A simple way to implement the virtualized storage is to make use of NFS (network file system) [9], [20]. In NFS, an NFS client on the cloud requests rendering input files to the on-premise system when they are actually requested during the rendering process, and the NFS server at the on-premise system finds the files from its storage and sends them to the cloud node. Although virtualized storage eliminates the unnecessary data transmission and long startup latency, it still has the problem of version control and validation of rendering data [21].

In this section, we present the file system design and implementation for cloud rendering that resolves the aforementioned problems of existing storage architectures. Our file system is designed as a user-level distributed file system that synchronizes original files in on-premise storage and cached files in cloud nodes. It has the function of version control, on-demand fetch, and distributed cooperative caching of rendering input data. Figure 6 shows a brief architecture of the proposed file system. Sections III-A and III-B will explain the on-premise side file server and the cloud side version-aware cooperative caching, respectively.

A. ON-PREMISE FILE SERVER

In our file system, on-premise storage is used as the data repository, which controls file-based retrieval and commits the input files via check-in and check-out functions. Specifically, a file server residing at the on-premise storage manages rendering input files. That is, rendering designers generate input files on the on-premise storage, and the file server synchronizes these files to cloud nodes when the rendering software requests the input files.

Meanwhile, as the file server needs to check the modified time of the input files before sending them, our file system manages the input files by assigning versions of files based on the modified time. Specifically, a new version is created whenever a rendering designer modifies an input file. Then, the file server sends the last generated version of the input file just before the current rendering begins on the cloud node, rather than the up-to-date version of the file. However, in reality, as rolling back to previous versions is possible, the default setting of our file server is to use the file versions of the rendering administrator's current working directory rather than the last generated version. Thus, when submitting a rendering task, the versions of all files at that time are snapshotted as a tree structure and delivered together to the cloud node.

B. VERSION-AWARE COOPERATIVE CACHING IN CLOUD

In order to reduce the data transmission overhead caused by large input files, a rendering cache manager resides on the cloud side of our file system. If all cloud nodes try to fetch original files from on-premise storage, it would be the performance bottleneck. Furthermore, it takes much time to transfer data between the on-premise storage and cloud nodes as they are connected by the wide area network (WAN). To cope with this situation, our rendering cache manager provides cooperative caching between peer cloud nodes, and also supports version-aware caching for input data consistency specialized for cloud rendering.

When rendering starts on a cloud node and input files are requested, the rendering cache manager first checks whether the files for the current rendering exist in the cache. If so, the cache manager checks whether the existing version is valid. If the cached version is valid, the cache manager returns it. Otherwise, the rendering cache manager tries to fetch the valid version of the input files from the nearest peer cloud node based on the cooperative caching [22], [23]. If the cooperative caching fails to find the valid version, the cache manager fetches the files from on-premise storage.

After fetching the requested input files, our rendering cache manager maintains them in its storage cache with the

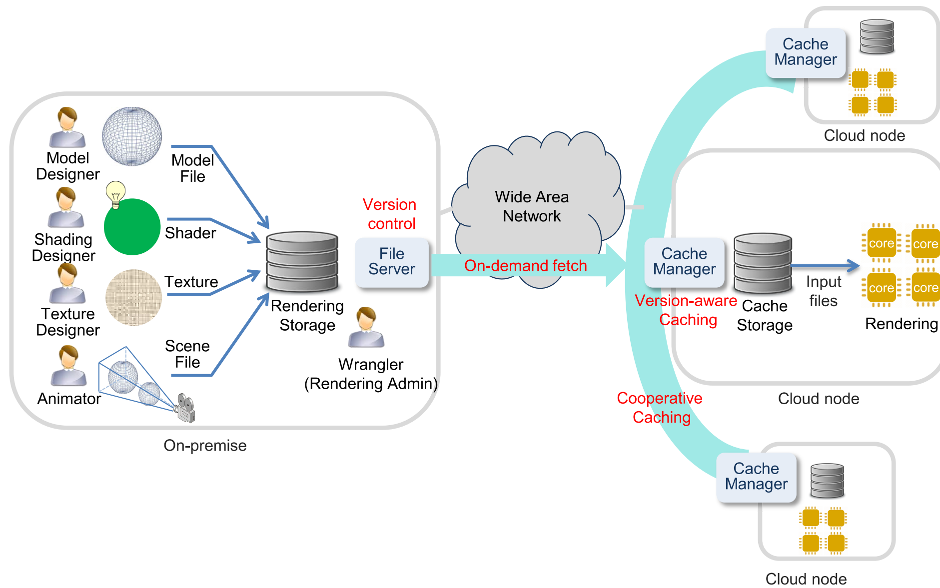


FIGURE 6. Our architecture for cloud rendering.

version information. Note that our rendering cache manager performs caching on a per-file basis, not per-block, which makes caching and validation more efficient by utilizing the rendering workload characteristics. A cached file is identified with its version as well as the path. Note also that the rendering software does not need to recognize the versions of files as our file system internally manages them based on the rendering data consistency, and the virtual file system interface is used.

IV. EXPERIMENTAL RESULTS

To evaluate the effectiveness of our file system implementation, we perform measurement experiments under both synthetic and real workloads.

Synthetic workloads are generated by the rendering workload characteristics analyzed in Section II. The number of input files are set to 1,000, and their average sizes are 45 MB. Popularities of the files are distributed such that the top 20% files account for 60% of total accesses. The total size of files is 53GB and the number of rendering tasks is set to 80. 646 files are accessed at least once out of the 1,000 files by the rendering tasks. Total size of the accessed files is 35GB out of the 53GB. For rendering nodes, we use 4 Citrix Xen virtual machines, and each node consists of E5-2620 8 Cores and 4GB memory. For the on-premise file server node, we use E5-2620 1 Core and 1GB memory. For a fair comparison, we experiment both cold start and warm start scenarios. In the cold start scenario, we measure the performance with the empty cache pool, whereas the warm start scenario measures the performance after filling the cache pool of the rendering nodes. Specifically, for measuring the performance of the warm start scenario, we first perform the rendering with the empty cache of the rendering nodes, and 20% of the input files are modified at the on-premise file server. Then, we measure the performance by executing the rendering again.

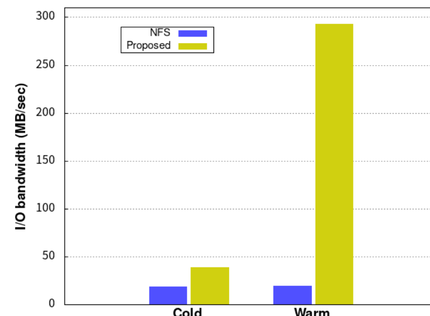


FIGURE 7. Comparison of the proposed file system and NFS with respect to the I/O bandwidth (synthetic workload).

Figure 7 shows the I/O bandwidth of the proposed file system in comparison with NFS. As shown in the figure, our file system performs significantly better than NFS in both cold start and warm start scenarios. Specifically, our file system exhibits 2.1x and 14.8x improved bandwidth for cold and warm start scenarios, respectively, compared to NFS. The performance improvement of the warm start scenario is large as we perform judicious caching by making use of version-aware caching and cooperative caching. Also, NFS has a problem in that it cannot provide the exact data consistency for rendering semantics.

Figure 8 compares the proposed file system with the upload scheme in terms of the total elapsed time. For the upload scheme, we measure the elapsed time of the two experimental configurations: $\text{Upload}_{\text{ALL}}$ and $\text{Upload}_{\text{SMART}}$. $\text{Upload}_{\text{ALL}}$ uploads all input files to the rendering nodes before starting the rendering process, whereas $\text{Upload}_{\text{SMART}}$ uploads only the files accessed during the rendering phase. As mentioned

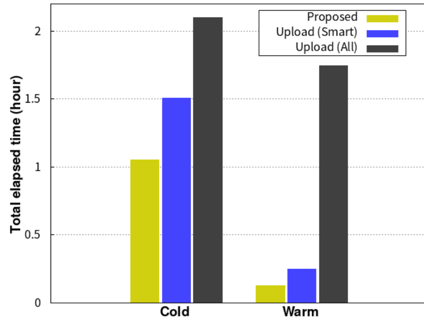


FIGURE 8. Comparison of the proposed file system and the upload scheme with respect to the elapsed time (synthetic workload).

in Section III, the rendering administrator performs the uploading of input files manually, and it is difficult to identify the files that will be actually used beforehand. As shown in the figure, our file system performs better than the upload scheme for all configurations. Specifically, the performance improvement of our file system against $\text{Upload}_{\text{ALL}}$ is 49.9% and 92.8%, respectively, for the cold and warm start scenarios. The performance gap is wider in the warm start scenario, and this shows the effectiveness of our judicious caching. When compared to $\text{Upload}_{\text{SMART}}$, the improvement of our file system is 30.3% and 48.9%, respectively, for the cold and warm start scenarios. This is a significant result as $\text{Upload}_{\text{SMART}}$ indicates an ideal case of uploading only the files that will be used, which needs to be decided by the administrator beforehand. Even in case of the cold start scenario, our file system exhibits significant improvement as we make use of the cooperative caching among peer rendering nodes.

To assess the effectiveness of the proposed file system under more realistic scenarios, we measure the performance while executing real workloads on AWS [4]. Specifically, we execute the open source rendering software Blender for rendering the Laundromat movie with 60 files [8], [16]. The network bandwidth between the on-premise file server and the cloud nodes is 1.8MB/s on average.

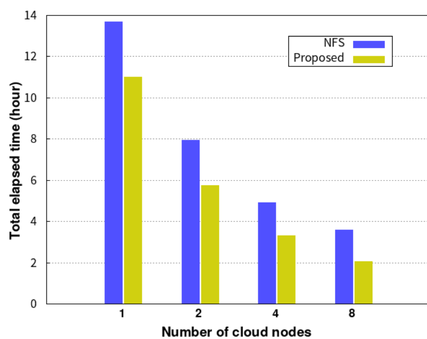


FIGURE 9. Comparison of the proposed file system and NFS with respect to the elapsed time as the number of cloud nodes is varied (real workload).

Figure 9 shows the total elapsed time of the proposed file system in comparison with NFS as the number of rendering nodes on cloud is varied. As shown in the figure, the elapsed time decreases almost in proportion to the number of rendering nodes. This is possible as rendering is an independent task, which can be performed concurrently on multiple cloud nodes. When comparing the two schemes, our file system performs better than NFS in all cases, with a consistent performance gap. The performance improvement of our file system is 30.5% on average and up to 42.3%. There are several reasons behind this result. First, NFS protocols have a certain overhead as it performs per-block cache validation, whereas our file system manages the cache based on the version information of the files. Second, as the on-premise file server and the cloud are connected via slow wide area network, the network and I/O bandwidths of the on-premise file server become the performance bottleneck in NFS. Third, the performance gap becomes wider as the number of cloud nodes increases because cooperative caching increases the likelihood of fetching data from nearby peer nodes.

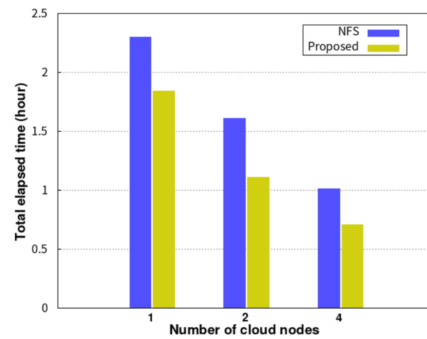


FIGURE 10. Comparison of the proposed file system and NFS with respect to the elapsed time as the same scene is rendered multiple times (real workload).

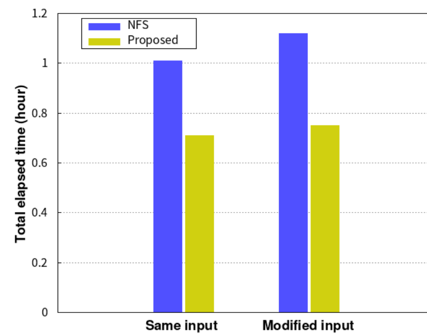


FIGURE 11. Comparison of the elapsed time for the proposed file system and NFS as the same scene is rendered multiple times (with the same input and the modified input files).

To see the effectiveness of the proposed file system under warm start scenarios, we choose 4 frames and perform rendering multiple times. Before rendering the same frames, which are already rendered, we modify 3.3% of the input files, and measure the elapsed time of rendering. Figure 10 plots the elapsed time of the proposed file system in comparison with NFS. As shown in the figure, our file system performs consistently better than NFS regardless of the number of rendering nodes. The performance improvement is

26.9% on average and up to 31.1%. Figure 11 compares the elapsed time of rendering with the same input files and the modified input files. As shown in the figure, our file system performs better than NFS by 29.7% and 33.0%, respectively, for the same and modified input files. Also, it is noticeable that the elapsed time of NFS increases by 11% with the modified input files whereas our file system degrades the performance by only 3%. This implies that the version-aware cooperative caching proposed in our file system is effective as it makes use of the cached items within all cache pools across the peer nodes, whereas NFS cannot do so because it should request all modified data to the on-premise file server.

V. CONCLUSION

Post-production and animation studios have a growing demand for rendering on public clouds to cope with the transient explosion of computational loads. Rendering workload is commonly known as computing-intensive but our analysis showed that it incurs heavy I/O due to the large input files, which makes technical hurdles for implementing cloud rendering efficiently. Specifically, 1) uploading large input files to cloud increases the latency of rendering significantly, and 2) the consistency requirement of rendering input files is complicated compared to that of traditional distributed file systems. In order to upload rendering workloads to cloud nodes seamlessly, this article analyzed the workload characteristics of popular rendering projects, and implemented a new file system for cloud rendering. Our file system is designed as a distributed file system that synchronizes original files in on-premise storage and cached files in cloud nodes, while satisfying the semantics of rendering input. We devised the function of version control, on-demand fetch, and distributed cooperative caching of rendering input files in our file system design, which allows for minimizing the data transmission overhead and satisfying the rendering data consistency. Measurement studies under synthetic and real workloads showed that the proposed file system outperforms the conventional upload scheme and NFS by 55.4% and 29.5% on average, respectively.

REFERENCES

[1] G. V. Patil and S. L. Deshpande, "Distributed rendering system for 3D animations with blender," in *Proc. IEEE Int. Conf. Adv. Electron., Commun. Comput. Technol. (ICAECCT)*, Dec. 2016, pp. 91–98.
[2] D. Shin, K. Cho, and H. Bahn, "File type and access pattern aware buffer cache management for rendering systems," *Electronics*, vol. 9, no. 1, p. 164, Jan. 2020.
[3] C. Rossl and L. Kobbelt, "Line-art rendering of 3D-models," in *Proc. 8th Pacific Conf. Comput. Graph. Appl.*, 2000, pp. 87–96.
[4] *Amazon Web Services*. Accessed: Aug. 23, 2021. [Online]. Available: https://aws.amazon.com/
[5] *Microsoft Azure*. Accessed: Aug. 23, 2021. [Online]. Available: https://azure.microsoft.com/
[6] J. R. Annette, W. A. Banu, and P. S. Chandran, "Rendering-as-a-service: Taxonomy and comparison," *Proc. Comput. Sci.*, vol. 50, pp. 276–281, Jan. 2015.
[7] K. Cho, J. Seo, J. Kang, J. Lee, S. Kim, J. Park, J. Song, J. Kim, and D. Kwon, "Render verse: Hybrid render farm for cluster and cloud environments," in *Proc. 7th Int. Conf. Control Autom.*, Dec. 2014, pp. 6–11.
[8] *Open Movie Projects*. Accessed: Aug. 23, 2021. [Online]. Available: https://www.blender.org/about/projects/
[9] B. Nowicki, *NFS: Network File System Protocol Specification*, document RFC1094, 1989.
[10] E. Lee and H. Bahn, "Caching strategies for high-performance storage media," *ACM Trans. Storage*, vol. 10, no. 3, pp. 1–22, Jul. 2014.
[11] T. Kim, H. Bahn, and K. Koh, "Popularity-aware interval caching for multimedia streaming servers," *IET Electron. Lett.*, vol. 39, no. 21, pp. 1555–1557, Oct. 2003.
[12] E. Lee, H. Bahn, and S. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2013, pp. 73–80.
[13] E. Lee, H. Kang, H. Bahn, and K. G. Shin, "Eliminating periodic flush overhead of file I/O with non-volatile buffer cache," *IEEE Trans. Comput.*, vol. 65, no. 4, pp. 1145–1157, Apr. 2016.
[14] H. Bahn, H. Lee, S. H. Noh, S. Lyul Min, and K. Koh, "Replica-aware caching for web proxies," *Comput. Commun.*, vol. 25, no. 3, pp. 183–188, Feb. 2002.
[15] *FUSE Filesystem in Userspace*. Accessed: Aug. 23, 2021. [Online]. Available: http://fuse.sourceforge.net/
[16] *Blender*. Accessed: Aug. 23, 2021. [Online]. Available: https://www.blender.org/
[17] M. F. Arlitt and C. L. Williamson, "Web server workload characterization: The search for invariants," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 24, no. 1, pp. 126–137, May 1996.
[18] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *Proc. 18th Annu. Joint Conf. IEEE Comput. Commun. Soc. Future*, 1999, pp. 126–134.
[19] D. Dobre, P. Viotti, and M. Vukolić, "Hybris: Robust hybrid cloud storage," in *Proc. ACM Symp. Cloud Comput.*, Nov. 2014, pp. 1–14.
[20] J. Slawinski, T. Passerini, U. Villa, A. Veneziani, and V. Sunderam, "Experiences with target-platform heterogeneity in clouds, grids, and on-premises resources," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp. Workshops PhD Forum*, May 2012, pp. 41–52.
[21] E. Anderson, "Capture, conversion, and analysis of an intense NFS workload," in *Proc. 7th USENIX Conf. File Storage Technol. (FAST)*, 2009, pp. 139–152.
[22] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson, "Cooperative caching: Using remote client memory to improve file system performance," in *Proc. 1st USENIX Conf. Oper. Syst. Design Implement. (OSDI)*, 1994, pp. 1–14.
[23] Y. H. Shin, H. Bahn, and K. Koh, "Directory-based coordinated caching in shared web proxies," in *Information Networking* (Lecture Notes in Computer Science), vol. 2662. Berlin, Germany: Springer, 2003, pp. 1010–1017.

KYUNGWOON CHO received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Seoul National University, in 1995, 1997, and 2012, respectively. He is currently a Senior Researcher with the Embedded Software Research Center, Ewha University, Seoul, Republic of Korea. Before joining Ewha, he was a Chief Officer with the Clunix Research and Development Center, Seoul. His research interests include multimedia systems, cloud computing, real-time systems, embedded systems, and operating systems.

HYOKYUNG BAHN (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Seoul National University, in 1997, 1999, and 2002, respectively. He is currently a Full Professor in computer science and engineering with Ewha University, Seoul, Republic of Korea. He has published more than 100 papers in leading conferences and journals including USENIX FAST, IEEE TRANSACTIONS ON COMPUTERS, IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, and *ACM Transactions on Storage*. His research interests include operating systems, caching algorithms, storage systems, embedded systems, system optimizations, and real-time systems. He also received the Best Paper Awards at the USENIX Conference on File and Storage Technologies, in 2013.