HSAStore: A Hierarchical Storage Architecture for Computing Systems Containing Large-Scale Intermediate Data

Zhoujie Zhang[1,2], Limin Xiao[1,2(✉)], Shubin Su[1,2], Li Ruan[1,2], Bing Wei[1,2], Nan Zhou[1,2], Xi Liu[1,2], Haitao Wang[3], and Zipeng Wei[3]

[1] State Key Laboratory of Software Development Environment, Beihang University, Beijing 100191, China
{jokerzhang, xiaolm, dreamsu}@buaa.edu.cn
[2] School of Computer Science and Engineering, Beihang University, Beijing 100191, China
[3] Space Star Technology Co., Ltd, Beijing 100086, China
wanghaitao@spacestar.com.cn, wei.zipeng@outlook.com

Abstract. This paper introduces HSAStore, a newly designed storage system whose goal goes to build an efficient storage system for computing systems that containing large-scale intermediate data. HSAStore involves three sub-systems to work together, a distributed file system which is in charge of the intermediate data, a centralized Network Attached Storage (NAS) which stores the raw input data and the results data, and a local file system that serves for the local data. The HSAStore takes full advantage of the network bandwidth of the computing cluster. Moreover, HSAStore adopts the distributed file system, so it is helpful for efficient execution of parallel programs. Experiments show that HSAStore has a significant improvement on efficiency in the computing systems that containing large-scale intermediate data.

Keywords: Hierarchical storage architecture · Bandwidth bottleneck
Large-scale intermediate data · Distributed file system

1 Introduction

Large-scale computer systems play an important role in scientific computing. A completely scientific computing task can be divided into three sub-steps: Input, Compute and Output [1, 2]. It has been a consensus that the development of storage devices is not as fast as the computing devices. As suggested by the "Bucket Effect", Input & Output component of the system, as know as storage system, deserves a special attention.

When the topic comes to storage system, the distributed solution has become a consensus. Google's distributed file system GFS (Google File System) [3] is a wonderful practice of this distributed idea. The core concept of Distributed File System (DFS) is constructing robust storage system with cheap disks. DFS has been widely applied by many companies. Three advantages make DFS to be an alternative to the conventional system. First, the distributed system is elastic, and system managers can

I. Romdhani et al. (Eds.): CollaborateCom 2017, LNICST 252, pp. 591–601, 2018.
https://doi.org/10.1007/978-3-030-00916-8_54

easily introduce a new storage node and need not to interrupt any of the running nodes. Second, DFS has a congenital advantage for parallel programs [4], such a feature can improve the performance of the scientific computing that can be parallelized. The last characteristic is that DFS is designed to be inexpensive and this goal is finally achieved by the engineers, users can adopt cheap hard disks and nodes to build it [5]. However, DFS is a complicated software system that just offers the necessary interface to accomplish its task, which makes the management job inflexible. On the other hand, the Centralized Storage System (CSS) is a traditional solution which can be tracked back down a very early age [6]. It is a straightforward thought to accomplish I/O requests in one place. The devices of a centralized system are always reliable. Unlike DFS, there is no extra steps are needed to maintain consistency [7], so the read and write requests are more efficient on the CSS. Furthermore, the CSS is better manageable than DFS.

This paper shows the design and implementation of HSAStore, a hierarchical storage system that is well designed for the computing systems containing large-scale intermediate data. Now, due to explosive growth of data, the data processing applications become more and more complex. These applications often involve multiple steps, many intermediate steps will produce a large amount of intermediate data. For example, the depth learning, each level of neural network will produce a lot of intermediate data. Another example is the seismic data processing application, general seismic data processing applications often read raw data from the storage system first, and then perform multiple iterations to complete the computing task. The whole task involves several procedures. Except the last procedure, each of them has its input and output. The output of the iterations is only useful for the next one or two steps, we call these output "intermediate data". These intermediate data are often several times as large as the raw data. While thousands of nodes coordinate to complete a task, those frequently read and write requests of intermediate data can result in considerable bandwidth consumption and competition.

The previous storage system deploys Network Attached Storage (NAS) devices to serve those huge I/O traffic from the compute component. However, the NAS is very expensive device, and its performance is limited by hardware manufacturing technology. Simply replace the centralized system with a distributed one is not a wise choice. We will introduce more details in Sect. 2 to explain this point. In this context, we designed HSAStore.

The rest of the paper is organized as follows. Section 2 introduces the traditional system and its problems. Section 3 shows how and why HSAStore performs well. We do evaluation jobs to verify our new system in Sect. 4. Section 5 displays some related works and the Sect. 6 is the conclusion of this paper.

2 The Previous System

In this section, we will make a brief description of NAS-based CSS. HSAStore is an improve version based on the NAS-based CSS, a general understanding of the CSS is necessary.

2.1 Background

When a computing task comes to the system, the cluster will distribute it to the available compute nodes. Every read/write request from those nodes will finally distribute to the storage system. Restricted by the performance of the hardware, if there are too many requests, the storage system cannot respond timely to each of them, especially when nodes come up to hundreds and thousands.

The CSS often employs Network Attached Storage (NAS) as the storage system. As Fig. 1 shows, since the bandwidth determines the maximum amount of data transmission per unit time, the bandwidth between the NAS and other components is much more important. In the CSS, data transmission bandwidth will be overloaded if a large number of nodes request the storage system at the same time. Here is a classic example, consider there is a cluster with 256 nodes, each of the nodes owns 2 CPU, and each CPU has 8 process cores and another 2096 GPU cores. Therefore, we have 4096 CPU cores (256 * 2 * 8) in our system. Suppose that the bandwidth of the storage system is 5 GB/s (1 GB = 1024 MB), the average bandwidth of each core (GPU cores and CPU cores) is just 0.83 MB/S. This figure will be even lower if we add more nodes in the cluster. As the Table 1 shows, the third row shows the average read bandwidth of applications, the fourth row tells us about the write bandwidth. Obviously, 0.83 MB/s is not a satisfactory count. Unfortunately, more nodes just make more traffic jams.

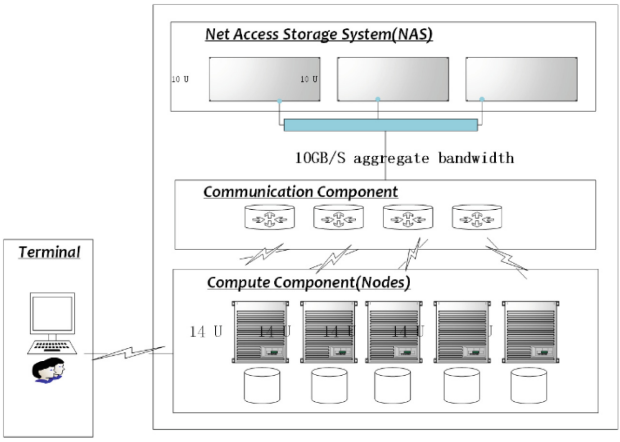

Fig. 1. NAS-based system

In other hand, there are two reasons that we cannot simply replace this NAS with a DFS to finish the optimization work of these computing systems. The first reason is about reliability. DFS has turned out to be a stable solution, but there is no sophisticated scheme for a scientific data processing system to work on DFS. Besides that, the original data should have flexibility to substitute. An example is that when doing geological exploration, we may replace an old storage device with a new one that has loaded new seismic data that collected from the field, the DFS cannot meet this demand.

Table 1. Typical features of a real-world computing system

Applications	Serial applications	Parallel application
Data size*	20	6 ~ 7
Storage space*	10	3
Read bandwidth /CPU core	12 MB/S	100 KB/S (with intermediate data) 750 KB/S (without intermediate data)
Write bandwidth/CPU core	4 MB/S	20 KB/S (with intermediate data) 450 KB/s (without intermediate data)

*Assume that the size of the raw input data is 1

2.2 Discussion

We have discussed about the advantages of the NAS-based version. In fact, NAS is a more efficient choice for some applications, especially under lightly loaded conditions. Due to those reasons we mentioned above, we cannot just simply displace the CSS by Distributed File System.

A wise man's question contains half the answer. We find two things deserve our attention. First is about the network of the cluster. The network's bandwidth can always reach 5.6 GB/s. But in the NAS-based storage system, it serves only the communication between nodes. 5.6 GB/s is a too large number to a communication service. This network has a lot of untapped potential. Second, as the Table 1 shows, some of the applications may process data that is 10 or more than 10 times than the original data, most of the data are intermediate data, that is, those data will be useless after the next step of the calculation. Take a consideration of where to direct those large-scale intermediate data is profitable.

3 Design and Implementation

HSAStore is an improvement of the NAS-based storage. As we talked about above, the capacity of the internal network is great more than the communication demands between those nodes. Therefore, we can easily build a distributed storage subsystem taking advantage of the internal network. This distributed storage subsystem can shares part of those I/O requests and reduce direct requests to the NAS. Now, the question is how to reach our target. In other words, in which way can we distribute the requests efficiently? Recall that those computing applications always generate large intermediate and very short-lived data. Obviously, any requests of these data, including read and write operations would be better not to direct to NAS. In this work, we put all of this data into the DFS. Thus, the DFS is just like a very huge cache for intermediate data, greatly relieve the load of NAS.

3.1 Overview

As the Fig. 2 shows, the architecture of HSAStore consists of three parts: the original storage component, the DFS which is used to be an intermediate data Cache, and the

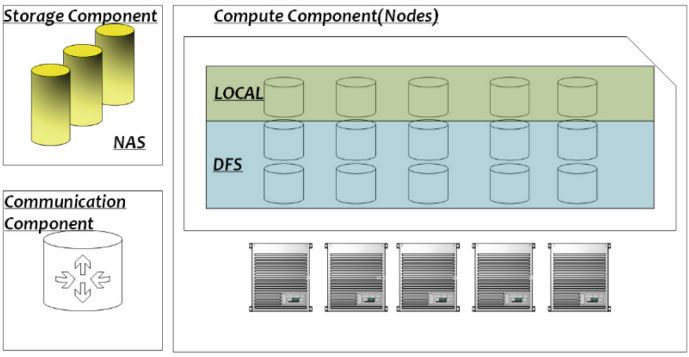

Fig. 2. HSAStore storage system

local storage. In addition, we have built a handy interface to make it more easily for applications programmers. The main idea is to provide a uniform access interface for the three subsystems of the storage system: the original storage component, the local storage, the DFS storage.

The DFS is the core component of HSAStore. Every node is equipped with one or more hard disks to make up this system. Besides requests of the intermediate data, other data can use the DFS if it is necessary. The interface will offer functions to tag useful data and these data will be transmitted to the NAS storage by a daemon process. The NAS subsystem offers the raw data and stores the result of the applications. Like the DFS, other data can be stored on this system, too. We build the local storage subsystem to store local data. Such as configuration information, it will be better to keep these data locally and this local storage system is responsible for handling these data. Local storage requires only another hard disk for each node.

3.2 Implementation

In this part, we will show more details about the interface of our new system. We employ HDFS [8] (Hadoop Distributed File System) as an instance of our DFS, and a unix-like file system as an instance of the local storage system. We will introduce the word "metadata" first. Metadata is a special data structure to store important information for a file [9, 10]. After that, we will explain basic file operations: read, write and delete.

Metadata

When a read/wirte request comes to the HSAStore, the first reaction is to read/write information from/to the metadata file. For example, if it is a request to write a new file on the DFS, the first is to record its location. This location string is a parameter of the requests. By recording this information into metadata file, a file can be accessed correctly in the future. In addition, metadata also includes some useful information such as the expiration time of a file.

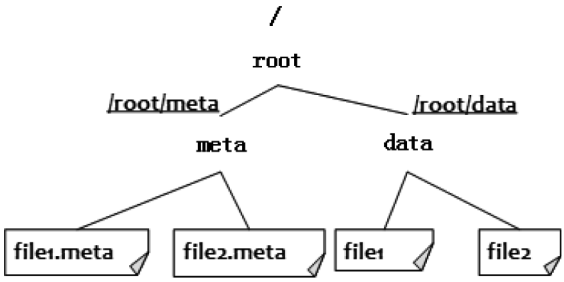

Fig. 3. Directory structure of the new system

As the Fig. 3 shows, we create a metadata directory on each subsystem. Here is an instance to specify how it works. Suppose that we create a new file named "file1", a complete representation of file1 is hdfs://ip:port/root/data/file1, and the path of its metadata has the same form like hdfs://ip:port/root/meta/file1.meta, they are quite the same except a extra "meta" directory of the meta file's path.

Write and Read

Write operations begin with a preparation process. To write a new file, the interface should find the boot file first. A boot file records all available location paths of the three subsystems, the interface knows how to find this boot file. Then, according to the parameter of the request, the interface chooses a valid path and saves all of the necessary information into the metafile. Finally, act just like a common write call. To write an existing file, sometimes we call this operation "append", we will search metadata file inside the metadata directory which we mentioned above. Once get the metafile, we know the path of the file we want to write, call native write interface of HDFS to do the remaining jobs. The read operation is similar to the write, the only difference is that the last step of read will call the native "read" interface.

Delete

Most of the data are intermediate data and it is short-lived. We recommend the users of our interface to delete files which are no longer useful by themselves. In these kinds of scenarios, our delete interfaces act like the native delete interface. Actually, such a delete operation is not a necessity. Interface users probably never care about if the useless files have been deleted. Sometimes the users are not sure whether this file will be useful for other users, and sometimes they just forget to delete that file. Thus, our write operation offers a parameter to make sure that a file will be expired after an exact time. The metafile of that data file will record this number. We set a default time of 15 days. Then we run an independent process accomplish the deletion work. This process checks each of the metafile inside the metadata directory, figure out how long it take since the file was created. Compare the result with the number we record in the metafile, if the result is greater than the recording time, then delete the metafile and its data file.

4 Evaluation

We performed several experiments to evaluate the performance of our new system. The environment of our experiments is shown in Table 2. The test cluster consists of 64 nodes, each node has 20 cores. Besides that, we equipped this cluster with a 10 Gb/s-ethernet. The storage system of our test cluster has a storage space of over 600 TB, including a Distributed File System whose space over 216 TB and a 400 TB NAS storage system.

Table 2. Test environment

CPU	64 nodes * 20 cores/node
Network	10 GB full wire-speed ehternet
Net access storage	4* Isilon X400 (over 400 TB SATA)
Distribute storage	Hadoop Distribute File Sytem (54 nodes/216 TB SATA)

We run seismic applications as typical scientific data processing applications on HSAStore and the NAS-based platform to finish the experiments. We use a custom test framework to generate real seismic application I/O requests. We concerned about three important performance characteristics in our experiment: the utilization of CPU, the latency of interactive applications and the efficiency of the critical steps.

Firstly, it is better to introduce the conception of "load level". As is shown in Table 3, the test framework submits tasks to the cluster. There are two typical tasks, one is compute intensive applications and another is I/O intensive applications. Compute intensive application won't burden the storage system. On the contrary, I/O intensive applications often launch lots of requests to the storage system, so it will cause an additional burden on the storage system. For example, the second column which we call "Low Load", there are 4 nodes run I/O intensive applications with 20 CPU cores each node, and 60 nodes run CPU-intensive applications. Contrast with 10 nodes run I/O intensive applications and 54 nodes run CPU-intensive applications, the former lead to fewer requests to the storage system. Thus, the former is "Low Load" and the latter is "Heavy Load".

Table 3. Load level of the experiments

Features	Low load	Medium load	Heavy load
Compute intensive	4 nodes * 20 cores	7 nodes * 20 cores	10 nodes * 20 cores
I/O intensive	60 nodes * 20 cores	57 nodes * 20 cores	54 nodes * 20 cores

Figure 4 shows the CPU efficiency of the HSAStore and the CSS under the different work load. Obviously, the utilization of CPU on HSAStore is relatively high. Especially in high load conditions, the result of HSAStore is over twice as much as the CSS. In other words, the "storage bottleneck" occurs in the high load situation has been broken. The performance improvement under light load is not that significant as the

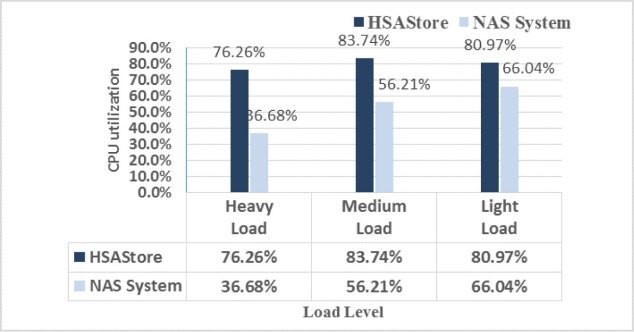

Fig. 4. CPU utilization at different loads under different platform

performance improvement under high load condition because that the bottleneck effect is not so obvious at low load condition. The result of this experiment shows that HSAStore can really help to improve the throughput of the system.

Figure 5 shows the latency of the interactive applications under different work load. We find that when the storage system is lightly loaded, there is about 60% performance improvement of our new system. The same as the previous experiments, high load leads to a marked decrease of the latency time. As the figure shows, there are almost 6 or 7 times of latency reduced under high load situation. The decrease is quite remarkable because that on the CSS the bottleneck occurs when I/O traffic gets busy. This is another proof that the HSAStore is more efficient than the CSS.

In a seismic data processing application, accomplish its critical steps as soon as possible is the ultimate goal. We insert code before and after those critical steps to record its execution time. Figure 6 shows that the critical steps of HSAStore always cost less time than the CSS regardless of the load level. Under heavy load situation, the execution time of HSAStore reduces to one fourth of the CSS. This is further evidence that our new system achieves our performance goal.

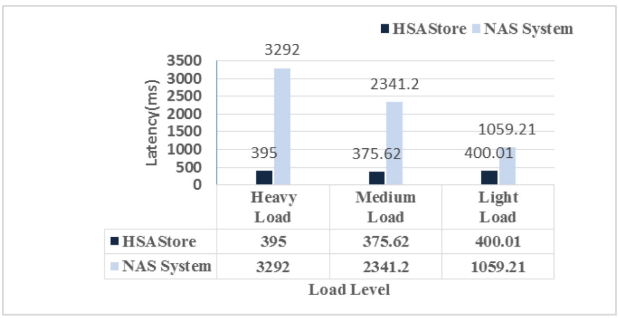

Fig. 5. Application latency at different loads under different platform

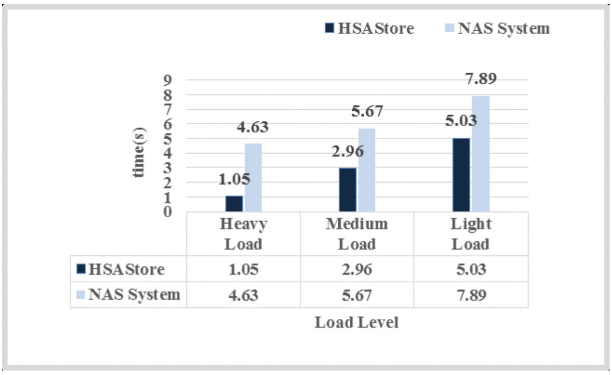

Fig. 6. Execution time of critical steps in seismic data processing application

5 Related Work

Our project is inspired by the research on hierarchical storage and metadata management. Hierarchical storage is a topic focus on how to use different storage subsystems for different working sets. Early works on this can be dated back to HP AutoRAID [11], a storage system which is divided into two levels. One with replications and another with RAID5. The system transfers data according to its status automatically. Facebook's storage system f4 [12], build such a hierarchical storage on a larger scale, and make choices in a more complicated strategy. Besides that, it takes a more simple model to distinguish different work load. Yang et al. [13] propose an I/O scheduler works on three layers: block, system call and page cache, such a multi-layer framework can make decisions with a more comprehensive information. And their split-I/O framework can be easily deployed to traditional operating system and databases. As a storage system for scientific applications, we construct our system with three parts, the system distributes requests of every special stage of the application to different storage subsystems.

Metadata management plays an important role in file systems. Beaver et al. [10] propose a design to minimize the size of metafile and thus the system can cache more metafiles into memory. More metafiles in memory can greatly reduce disk I/O. Zhang et al. [14] proposes several ways to group related metafiles, and they address an algorithm to accomplish this job intelligently. Patil et al. [15] use a sophisticated algorithm to manage millions of metafiles of a system. In our system, we just use a straight-forward way to manage metafiles, a more efficient method can be a further exploration of our future work.

6 Conclusion

In this paper, we proposed a new design of the storage architecture for computing systems containing large-scale intermediate data. The main idea of HSAStore is to avoid distributing all of the I/O requests to the NAS. In HSAStore, DFS will process

most of the requests of the intermediate data, and it takes full advantage of the cluster's communication network, so the bandwidth of HSAStore is the network's bandwidth plus the NAS's bandwidth. We have increased the bandwidth of the storage system with a very low cost. Experiments show that the HSAStore is effective and efficient for the computing systems containing large-scale intermediate data. In the future, we will apply HSAStore to the depth of learning.

Acknowledgments. This work was supported by the National Science Foundation of China (Grant Nos. 61370059, 61772053 and 61232009 (key project)), Beijing Natural Science Foundation (Grant No. 4152030), the fund of the State Key Laboratory of Computer Architecture (CARCH201507), the fund of the State Key Laboratory of Software Development Environment (SKLSDE-2017ZX-10), the State Administration of Science Technology and Industry for National Defense, the major projects of high resolution earth observation system under Grant No. Y20A-E03, and the National Key R&D Program of China under Grant NO. 2017YFB1010000.

References

1. Zhou, A.C., He, B., Ibrahim, S.: A taxonomy and survey of scientific computing in the cloud (2016)
2. Maheshwari, K., Wozniak, J.M., Yang, H., et al.: Evaluating storage systems for scientific data in the cloud. In: Proceedings of the 5th ACM Workshop on Scientific Cloud Computing, pp. 33–40. ACM (2014)
3. Ghemawat, S., Gobioff, H., Leung, S.T.: The Google file system. ACM SIGOPS Oper. Syst. Rev. ACM **37**(5), 29–43 (2003)
4. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
5. Gibson, G.A., Nagle, D.F., Amiri, K., et al.: A cost-effective, high-bandwidth storage architecture. ACM SIGOPS Oper. Syst. Rev. ACM **32**(5), 92–103 (1998)
6. Gibson, G.A., Van Meter, R.: Network attached storage architecture. Commun. ACM **43**(11), 37–45 (2000)
7. Bai, S., Wu, H.: The performance study on several distributed file systems. In: 2011 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), pp. 226–229. IEEE (2011)
8. Shvachko, K., Kuang, H., Radia, S., et al.: The hadoop distributed file system. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10. IEEE (2010)
9. Deng, K., Song, J., Ren, K., et al.: Exploring portfolio scheduling for long-term execution of scientific workloads in IaaS clouds. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 55. ACM (2013)
10. Beaver, D., Kumar, S., Li, H.C., et al.: Finding a needle in haystack: Facebook's photo storage. OSDI **10**, 1–8 (2010)
11. Wilkes, J., Golding, R., Staelin, C., et al.: The HP AutoRAID hierarchical storage system. ACM Trans. Comput. Syst. (TOCS) **14**(1), 108–136 (1996)
12. Muralidhar, S., Lloyd, W., Roy, S., et al.: f4: Facebook's warm blob storage system. In: Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation, pp. 383–398. USENIX Association (2014)

13. Yang, S., Harter, T., Agrawal, N., et al.: Split-level I/O scheduling. In: Proceedings of the 25th Symposium on Operating Systems Principles, pp. 474–489. ACM (2015)
14. Zhang, Z., Ghose, K.: hFS: A hybrid file system prototype for improving small file and metadata performance. ACM SIGOPS Oper. Syst. Rev. **41**(3), 175–187 (2007)
15. Patil, S.V., Gibson, G.A., Lang, S., et al.: Giga+: scalable directories for shared file systems. In: Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing'07, pp. 26–29. ACM (2007)