# XkitS：A computational storage framework for high energy physics based on EOS storage system

*Yaosong* Cheng[1,*], *Yujiang* Bi[1,3,**], *Yaodong* Cheng[1,2,3], *Haibo* Li[1,2,3], *Yu* Gao[1,2], *Minxing* Zhang[1,2]

[1]Institute of High Energy Physics, CAS, 100049 Beijing, China
[2]University of Chinese Academy of Sciences, 100049 Beijing, China
[3]Tianfu Cosmic Ray Research Center, Institute of High Energy Physics, Chinese Academy of Sciences, 610041 Chengdu, China

**Abstract.** Large-scale high-energy physics experiments generate scientific data at the scale of petabytes or even exabytes, requiring high-performance data IO for processing. However, in large computing centers, computing and storage devices are typically separated. Large-scale data transfer has become a bottleneck for some data-intensive computing tasks, such as data encoding and decoding, compression, sorting, etc. The time spent on data transfer can account for 50% of the entire computing task. The larger the amount of data accessed, the more significant this cost becomes. One attractive solution to address this problem is to offload a portion of data processing to the storage layer. However, modifying traditional storage systems to support computation offloading is often cumbersome and requires a broad understanding of their internal principles. Therefore, we have designed a flexible software framework called XkitS, which builds a computable storage system by extending the existing storage system EOS. This framework is deployed on the EOS FTS storage server and offloads computational tasks by invoking the computing capabilities (CPU, FPGA, etc.) on FTS. Currently, it has been tested and applied in the data processing of the Large High Altitude Air Shower Observatory (LHAASO), and the results show that the time spent on data decoding using the computable storage technology is half of that using the original method.

## 1 Introduction

We are currently in the era of data. With the rapid development of intelligent technologies in fields such as autonomous driving, the Internet of Things, and biomedicine, data production and analysis have become crucial factors in determining productivity. By 2030, the world will be generating and requiring the processing of nearly 1 YB (yottabyte) of data per year, posing new challenges for data storage, transmission, and processing. These challenges are

[*] e-mail: chengys@ihep.ac.cn
[**] e-mail: biyujiang@ihep.ac.cn

particularly prominent in the field of high-energy physics, as high-energy physics experiments often rely on large-scale scientific facilities, and the complexity and scale of these advanced experimental setups are increasing every year, resulting in hundreds of terabytes of data being generated with each experiment. Taking the Chinese Academy of Sciences High Energy Physics Science Data Center as an example, the amount of data that needs to be analyzed annually has reached 400 PB (petabytes), placing an enormous demand on data storage and computing resources.

The data processing workflow in high-energy physics experiments typically includes stages such as data acquisition, filtering, reconstruction, analysis, and simulation. Each stage requires the simultaneous processing of a large number of events, making it a typical data and I/O-intensive computation. To facilitate management, data centers and many international high-energy physics research organizations adopt a storage-compute separation model[1] to build the entire data analysis environment, as shown in Figure 1. This environment includes login clusters, storage clusters, computing clusters, etc.

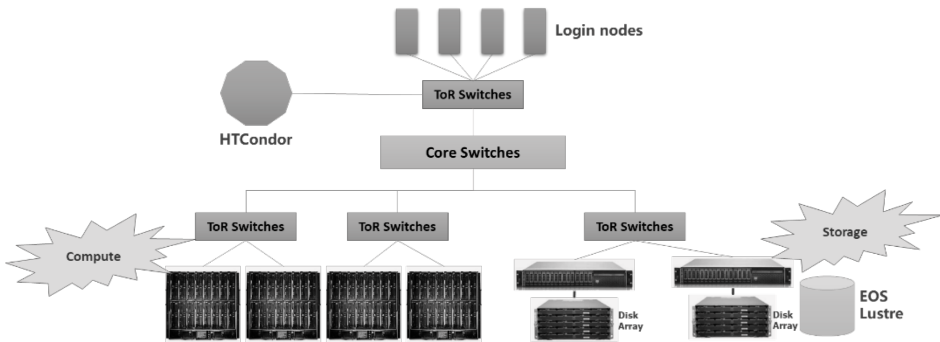


**Fig.1** Typical high energy physics data processing environment.

These subsystems are independent of each other and interconnected through a highly reliable high-speed core network. After users submit computing tasks on the login cluster, the computing cluster integrates CPU resources from a large number of computing nodes through a job scheduling system to uniformly schedule and arrange the tasks submitted by users. Subsequently, the data is read from the storage cluster to execute the computing tasks.

This system architecture provides high efficiency when dealing with small-scale data. However, with the construction of new-generation experimental facilities, the drawbacks of the storage-compute separation architecture are also fully exposed in I/O-intensive tasks. In a complete set of computing tasks, data is first transferred from storage nodes to computing nodes through the network, and after the computation is completed, the results are transmitted back to the storage nodes. Frequent data transfer consumes a large amount of network bandwidth, especially for tasks that only require minimal CPU usage. The time spent on data transfer may account for 50% or even more of the entire computing task. When such types of computing tasks are numerous in the computing cluster, it often leads to CPU idle time, network congestion, packet loss in switches, and many other issues. Figure 2 represents the data read/write bandwidth of the EOS storage cluster[2] in the data center at a specific moment.

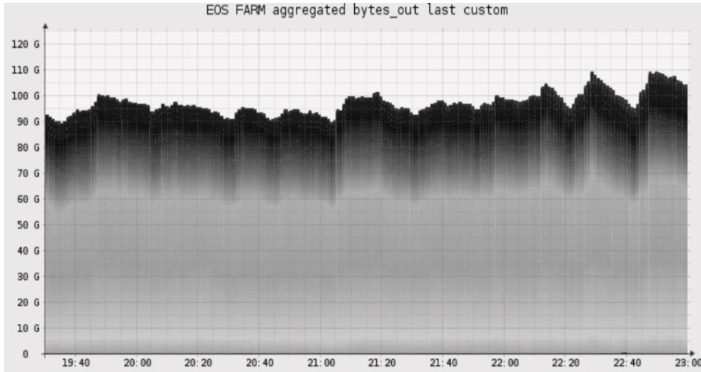

**Fig.2** Data read/write bandwidth of the EOS storage cluster.

One approach to problem-solving is to increase network bandwidth, but this does not eliminate CPU waste and can be very costly. Therefore, it is necessary to explore new computing architectures to address these challenges. Computational storage technology has been proposed in this context, with the goal of equipping storage units with computing capabilities. This involves offloading a portion of the computational tasks from the computing units to the storage units, reducing the amount of data transmitted over the network, achieving faster response times, and saving energy.

## 2 Computational storage technology

According to the definition provided by the Computational Storage Technical Workgroup of the Storage Networking Industry Association (SNIA), computational storage is an architecture that provides computing capabilities tightly coupled with storage to offload host computing tasks or reduce data movement[3]. The implementation approaches mainly include the following:

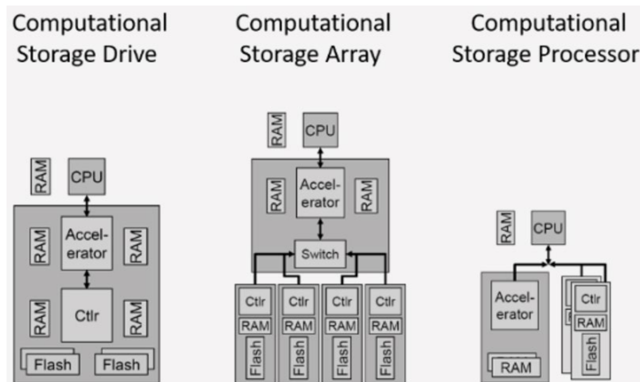

**Fig.3** Three types of computational storage architectures.

Computational Storage Drive (CSD): This involves adding computational acceleration components such as FPGA to the hard disk drive, enabling the disk itself to possess computing capabilities and provide computational storage services and persistent data storage.

Computational Storage Array (CSA): In this approach, computational acceleration components such as FPGA are added to the disk array controller, empowering the disk array with computing capabilities.

Computational Storage Processor (CSP): This involves adding dedicated computing chips (such as ASIC, FPGA, or GPU) directly to the server to process data connected to the server's hard disk. Since external storage is connected via the motherboard bus like PCIe, CSP can theoretically manage a large disk capacity.

Based on these architectures, there have been numerous computational storage implementation solutions internationally. For instance, In the context of the relational database PolarDB, literature [4] offloads the computationally expensive table scan operations from the CPU to the computational storage drive. Through collaborative innovations at the software and hardware levels, they effectively reduced query latency and data transfer volume in the database. In virtualized environments, the fundamental challenge for applying computational storage is achieving virtualization in an economically efficient manner. Literature [5] proposed the FCSV-Engine FPGA card, which utilizes hardware-assisted virtualization and resource orchestration to achieve high virtualization performance. By dynamically constructing multiple virtual computational storage devices at the hardware level, they perform near-storage processing, achieving cost-effectiveness. Literature [6] extended the data processing and management capabilities of the Ceph distributed storage system by custom extensions, data partitioning, and structured data storage. This allows storage servers to semantically interpret object data to execute certain SQL statements. The benefits include both I/O and computational elasticity, with the storage system automatically rebalancing objects across available servers.

To better apply computational storage in high-energy physics experiments, we have also developed our own computational storage solution called XkitS (eXtendable kit for computational Storage). It is built on the distributed storage system EOS developed by the European Organization for Nuclear Research (CERN)

## 3 XkitS: A computational storage framework based on EOS.

EOS is a distributed storage system based on the XrootD framework, consisting of main modules including the client, metadata server, and data storage server, as shown in Figure 4. In the metadata server, the authentication, authorization, data scheduling, metadata management, and storage management tasks are handled by the MGM module, while the message proxying is done by the MQ module. In the data storage server, the FST module is responsible for file storage and transfer.

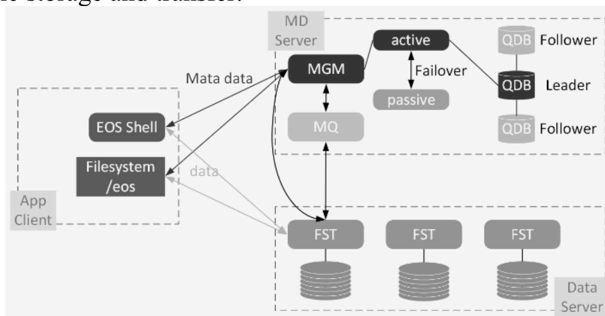


**Fig.4** EOS architecture.

When a user program accesses data in EOS, it typically calls built-in methods such as Open, Write, Read, etc. For example, a command to open a file would be parsed as Open("root://eos01/eos/data.txt"). When EOS receives this command, it transfers the file data.txt to the client for subsequent computational tasks. However, as mentioned earlier, when the data file is large and the computational task is simple, most of the job execution time is spent on data transfer. One approach to implementing a computational storage

processor (CSP) is as follows: when EOS receives a request to open a file, the file is opened locally on the FST where it resides, and the computational task is executed on the CPU or other computational resources of that server. The computation results are then directly written back to the current FST, saving time on data transfer over the network.

XkitS includes a plugin called EosFstCss that implements this functionality, as shown in Figure 5.

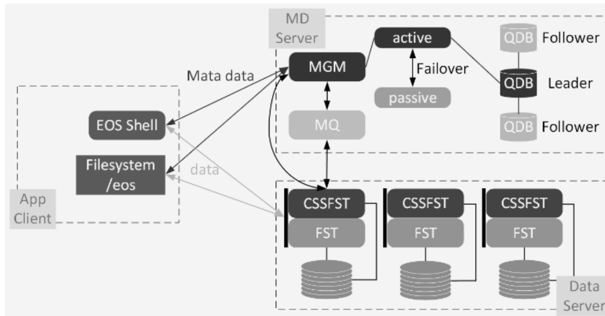


**Fig.5** The functioning mechanism of EosFstCss.

To invoke this functionality, a keyword "ccs" needs to be added at the end of the data request command to differentiate it from the native data access methods. For example, the Open command mentioned earlier can be written as: Open("root://eos01/eos/data.txt?css=sort"). In this command, "?" is the separator, "ccs" indicates accessing the file through CCSFST instead of FST, and the parameter "sort" indicates using the sort function specified in the configuration file on the FST server to perform sorting computation. Therefore, the result returned by this command would be the sorted data.txt.

In this example, the implementation principle of CCSFST is as follows:

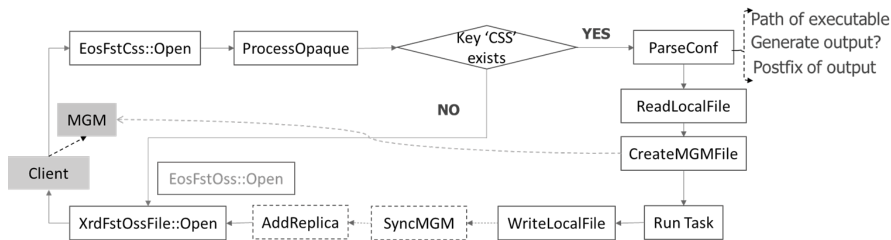


**Fig.6** The workflow of EosFstCss.

First, the target file address, "ccs" flag, and corresponding computational storage algorithm are obtained by parsing the OPEN command. Then, the file is opened on the server where it is stored and registered with the MGM service. Next, the specified computational storage algorithm is applied to the target file. The output of the algorithm is written to a local output file on the node, which is named according to EOS naming rules, such as fid/10000. If the algorithm itself does not generate an output file, stdout or stderr can be redirected to the output file. Subsequently, the results are synchronized to the MGM module of the metadata server and additional replica backups are created. Finally, an Open call is made to the output file and returned to the client.

When parsing user commands, the client recognizes the computational storage parameters and notifies the CSSFST in the storage server to perform specific operations on the file instead of the FST. Based on file I/O, the computational resources of the storage server are

used to execute the computational operations, which are then stored locally on the storage server or returned to the caller based on the specific type of computational task.

## 4 Deployment and usage of XkitS

We aim to make the deployment of XkitS plugins as simple as possible, preferably without modifying any EOS code. This facilitates faster adaptation when updating EOS versions. Therefore, we have created a cssfst RPM package that depends on EOS's FST service. After installing it, only two configuration files need to be modified to enable computational storage functionality in EOS. Specifically, the standard configuration file /etc/xrd.cf.fst needs to be modified by replacing "xrootd.fslib -2 libXrdEosFst.so" with "xrootd.fslib -2 libEosFstCss.so -2 libXrdEosFst.so". Then, the configuration file /etc/eoscss.conf needs to be edited to customize the computational storage functionality. The configuration file is in JSON format and consists of four main sections: "name", "path", "out", and "postfix". Here is an example that includes two computational storage functionalities:

```
{ "sort" :    {
   "name" : "sort",
   "path" : "/usr/local/libexec/cssfst/sort.sh",
    "out"  : false },
  "km2a_decode" : {
   "name" : "km2a-decode",
   "path" : "/usr/local/libexec/cssfst/km2a-decode.sh",
   "out"  : true,
   "postfix" : "root"} }
```

In the configuration file, "name" refers to the algorithm name, which serves as the basis for client calls to the CSS service and corresponds to the name of the executed algorithm. "path" is the path to the executable file of the algorithm, which is independent of EOS and is used to perform computational tasks on files. It can be freely set by the user. "out" and "postfix" are related. "out" indicates whether the algorithm has an output. If it is set to "true", "postfix" must be set, which represents the suffix of the output file. If it is set to "false", "stdout" or "stderr" can be redirected to the output file.

Customized computational storage functionalities are the user's own programs, which can be shell scripts or other executable files. The programs utilize the CPU, GPU, FPGA resources of the FST server, and can even be containers (docker, singularity). All executable programs should take an input file and produce an output file, returning "EXEC_SUCCESS" or "EXEC_FAILED". Here is an example of a sorting algorithm:

```
#cat sort.sh
/usr/bin/sort -n $1 > $2
if [ $? -eq 0 ];then
 echo "EXEC_SUCCESS"
exit 0
echo "EXEC_FAILED"
exit 1
```

Once the configuration is completed, the computational storage functionality can be invoked using either xrdcp or the dedicated cssclient tool. If using xrdcp, simply append the CSS function name to the file path. For example: Once the configuration is completed, the computational storage functionality can be invoked using either xrdcp or the dedicated

cssclient tool. If using xrdcp, simply append the CSS function name to the file path. For example:

*xrdcp root://eosbak02.ihep.ac.cn//eos/user/chyd/data.txt?css=sort -*

The cssclient is based on the XrdPosixXrootd wrapper. Before using it, you need to export the environment variable EOS_MGM_URL:

*export EOS_MGM_URL=root://eos01.ihep.ac.cn/*

*cssclient -f /eos/user/chyd/data.txt -c sort*

We conducted tests using the experimental program "decode" from the Large High Altitude Air Shower Observatory (LHAASO)[7]. LHAASO is a large-scale cosmic ray detector array located at an altitude of 4410 meters in southwest China, approximately 2000 km away from Beijing. It generates 12PB of data annually, which is then transferred to Beijing. The "decode" process converts the raw binary data acquired by the detectors into ROOT files. This process involves reading and writing a large amount of data but consumes minimal CPU power. In the traditional computing mode, a compute node reads the raw data (.dat) from an FST server and writes the output data (.root) to another EOS server. However, by using the computational storage functionality through XkitS, any XRootd client can launch the "decode" process on the FST server for local data reading and writing, either through the XRootd client or cssclient. Figure 7 compares the data processing time between the two modes. It can be observed that when processing the same file with the same program, CSSFST only takes half the time compared to the traditional mode.

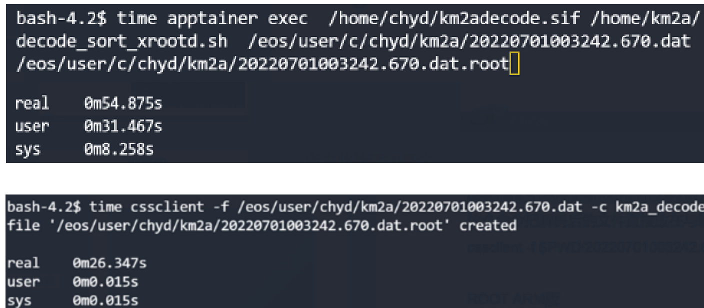


**Fig.7** Running time of the "Decode" in Traditional Mode and Computational Storage Mode.

## 5 Conclusion

XkitS, which we have developed, is a computational storage implementation method that utilizes the computing resources of EOS servers. It has the characteristics of scalability, configurability, ease of deployment, and ease of use. Through real-world cases, we have demonstrated the high performance advantages of computational storage in simple computing tasks with large data transfers. Additionally, we can further accelerate data processing speed by incorporating heterogeneous computing capabilities such as GPU, CPU/SoC, and FPGA into the storage servers. However, during our development process, we still encountered some issues, such as the difficulty of reducing data movement in EOS's RAIN (erasure code) mode and task scheduling between storage servers. We hope to collaborate with the community to enhance the computational storage functionality and make it one of the optional features of EOS.

## 6 Acknowledgments

This work is supported by the National Natural Science Foundation of China (No.12075268).

This work is partly supported by the Science and Technology Innovation Project of Institute of High Energy Physics, Chinese Academy of Sciences (No.E15451U2).

This work is supported by the Youth Innovation Promotion Association of CAS (No. 203013).

## References

1. Y.D. Cheng, J.Y. Shi, G. Chen, Overview of High Energy Physics Computing Environment. Sci. Info. Tech. & Appl. **5(3)**, 3-10 (2014)
2. P.AJ, S.EA, A.G. EOS as the present and future solution for data storage at CERN.pdf. J. Phys.: Conf. Ser., 042042 (2015)
3. SNIA-Computational Storage Architecture and Programming Model Version 1.0, Computational Storage – SNIA on Storage (sniablog.org)
4. W. Cao, Y. Liu, Z. Cheng, et al. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. FAST, (2020)
5. D. Kwon, W. Lee, D. Kim, et al. SmartFVM: A Fast, Flexible, and Scalable Hardware-based Virtualization for Commodity Storage Devices. ACM, **2**, 18 (2022)
6. J. Lefevre, C.M. Ahn, SkyhookDM: Data Processing in Ceph with Programmable Storage, **45**, 2, (2020)
7. Y.D. Cheng, H.B. Li, et al. Construction and application of LHAASO data processing platform. Radiat Detect Technol Methods, **6**, 418–426 (2022), https://doi.org/10.1007/s41605-022-00328-2