OOOPS: An Innovative Tool for IO Workload Management on Supercomputers

Lei Huang
Texas Advanced Computing Center
The University of Texas at Austin
10100 Burnet Rd. Austin, TX, 78758
Email: huang@tacc.utexas.edu

Si Liu
Texas Advanced Computing Center
The University of Texas at Austin
10100 Burnet Rd. Austin, TX, 78758
Email: siliu@tacc.utexas.edu

***Abstract*—Modern supercomputer applications are demanding powerful storage resources in addition to fast computing resources. However, these storage resources, especially parallel shared filesystems, have become the Achilles' heel of many powerful supercomputers. A single user's IO-intensive work can result in global filesystem performance degradation and even unresponsiveness. In this project, we developed an innovative IO workload managing tool that controls the IO workload from user applications' side. This tool is capable of automatically detecting and throttling intensive IO workload caused by supercomputer users to protect parallel shared filesystems.**

***Index Terms*—Supercomputers, User support tool, IO, Parallel filesystem, Metadata server**

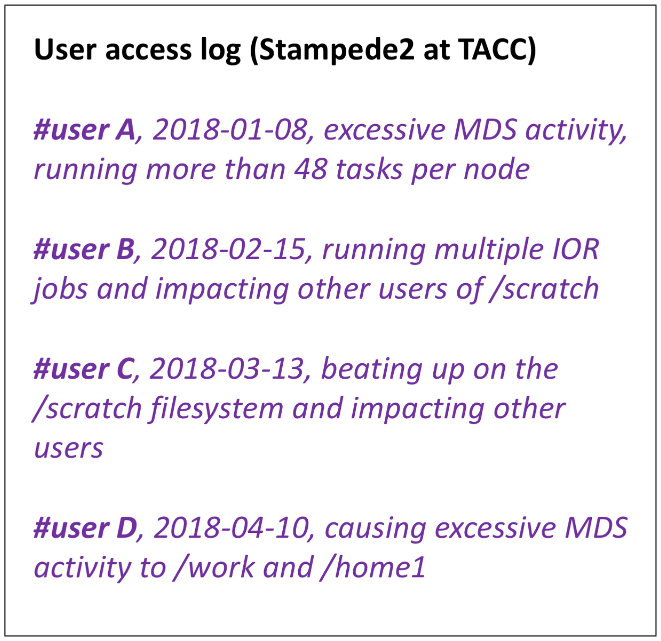
I. Introduction

Many widely-used supercomputer applications in computational fluid dynamics, quantum chemistry, machine learning, etc. involve IO-intensive work that can raise a significant number of IO requests in a very short period of time. Generally, there is no strictly enforced IO resource limitation in production (e.g. metadata throughput, IO bandwidth) on the user level, application level, or node level. Therefore, a single user's IO-intensive work raised from a small number of compute nodes can easily cause excessive pressure on the metadata servers (MDS) or service nodes, which results in global filesystem performance degradation and even unresponsiveness. Consequently, other users working on the same supercomputers and the rest of compute nodes cannot acquire IO resources effectively.

A. *Administrators facing IO issues*

Authors of this paper have been working in the user support team of Texas Advanced Computing Center (TACC) and The Extreme Science and Engineering Discovery Environment (XSEDE) for many years. TACC, as well as many other XSEDE institutions or supercomputer centers, supports hundreds and even thousands of active users over the lifetime of each supported supercomputer. However, almost every one or two weeks, TACC supercomputer administrators have to temporarily block some users to protect the shared filesystem from hazardous IO-intensive work.

Most TACC supercomputer systems use an in-house Simple Linux Utility for Resource Management (SLURM) plugin filter to control the users access list. Therefore, administrators can conveniently update and track the blocked users. Fig. 1 shows a snippet of Stampede2 [29] user access logs in early 2018. It is observed from these logs that many users were blocked from the system because of their hazardous IO work, particularly the IO-intensive work that generated a lot of MDS requests.



User access log (Stampede2 at TACC)

***#user A**, 2018-01-08, excessive MDS activity, running more than 48 tasks per node*

***#user B**, 2018-02-15, running multiple IOR jobs and impacting other users of /scratch*

***#user C**, 2018-03-13, beating up on the /scratch filesystem and impacting other users*

***#user D**, 2018-04-10, causing excessive MDS activity to /work and /home1*

Fig. 1. This figure shows a snippet of Stampede2 access logs in early 2018. Real user names are hidden for privacy.

B. *IO threatening applications*

This IO problem is not new to system administrators or supercomputer support teams. Such an issue is also well-known in the user community. Many popular applications like IOR [1], Tensorflow [3], or NWChem [14] can cause this kind of IO problem in their routine work. According to our recent experience, one of the most representative applications that often trouble users is OpenFOAM [18], [32]. OpenFOAM (The Open-source Field Operation And Manipulation) is very popular in the supercomputer and fluid dynamics community. It contains intensive IO work, especially for large scale simulations. Nevertheless, OpenFOAM's IO workflow is unsatisfactory [5]. It creates separate data files for each decomposed field and mesh, at each time step, and from each individual processor. This is catastrophic for long-term large-scale runs on modern supercomputers.

Authors of this paper were requested to work with several OpenFOAM IO issues before. For example, scientists and engineers from the Pittsburgh Supercomputing Center (PSC) and the University of Colorado at Boulder tried to complete a series of large-scale simulations on TACC and PSC systems in 2014-2015. They were not allowed to do so because of the huge number of IO requests raised by their planned OpenFOAM simulations. A few IO experts rewrote several IO plugins of OpenFOAM with parallel HDF5 and NetCDF format to reduce the number of files accessed within OpenFOAM in order to reduce the MDS requests [31], [35]. The specific problem was resolved, but it took several months of hard work. Such a project also requires much expertise on both the application side and the system side.

C. Our goal

According to these early experiences from both system administrators and domain experts, we can see that Supercomputer users are encountering more IO issues lately. The pressure on the MDS or service nodes has caused severe problems to the supercomputer community along with the growth of the computing power, system scale, and problem size. Users' IO work is frequently limited by filesystems' metadata capacity and metadata processing speed rather than the IO network bandwidth. In our work, we would like to provide a user-support tool that is able to release the pressure of MDS and service nodes. This tool must be easy-to-use for users and administrators and should not hurt the performance dramatically for the applications without extremely intensive IO work.

Meanwhile, modern large-scale supercomputers are normally shared by hundreds of users at any given time. Similar to the computing resource, memory resource, network bandwidth, etc., the MDS processing capacity should be treated as an allocatable resource on a supercomputer as well. By implementing an extra limitation on the MDS usage, this tool practically realizes real-time MDS provisioning, therefore no single user or team can monopolize the MDS resource on a system with only a small fraction of the whole cluster.

The rest of the paper is organized as follows. Section II presents background and related work in the field of IO optimization and management. Section III describes the design and implementation of our new tool. Section IV presents several test cases and results. Highlights of this new user support tool and future works are discussed in Section V and VI. The conclusion is presented in Section VII.

II. Background and related work

A. Parallel Filesystems and Metadata Server

A modern supercomputer systems usually consists of a set of computer nodes, a high-speed interconnection network, and a parallel filesystem. A parallel filesystem allows multiple users or clients to read and write with it simultaneously and allows data to be spread across multiple disks or data servers. Common parallel filesystems include the open-source Lustre [19], [21], [37] filesystem, IBM General Parallel File System(GPFS)/Spectrum Scale [10], [36], Parallel Virtual File System(PVFS)/OrangeFS [2], [38], Panasas PanFS [20], and so forth [23].

A common parallel filesystem normally employs one or more servers to manage the metadata including the file name, directory structure, access permissions, extended attributes, file layout, etc. In addition, a parallel filesystem also employs a set of data servers, usually separated from the metadata server(s), to manage the storage of the data content on a set of storage volume. For example, a typical Lustre filesystem usually has one MDS with the metadata content stored on one Meta Data Target (MDT) and a set of Object Storage Servers (OSS), each handling multiple Object Storage Targets(OST). A typical GPFS/Spectrum Scale filesystem is usually comprised of a set of storage servers dealing with the metadata and/or the data content, and a series of shared disks accessible from these servers [23].

The metadata operations usually happen at the beginning and end of each IO function call, i.e. when a target data file is opened and closed. The metadata server is not involved in the transaction while the data servers are working on the data content. It is intentionally designed to avoid transition between metadata work and disk work to reduce extra overheads.

The metadata requests kept in the memory of each individual metadata server are generally processed in a single-thread way. When there are too many IO operations generated in a short period of time, the metadata request queue is full or overflow and the memory resource is in contention. Unfortunately, the number of IO requests/operations is decided by the usage from the user side. Many IO-intensive applications can generate thousands to millions of IO requests instantly, especially in large-scale runs. The growth of computer processing power makes it even worse. This easily leads to a performance degradation and even unresponsiveness of the whole filesystem. In the worst case, the filesystem needs to be rebooted and those unfinished IO requests are lost. In such extreme situations, a fraction of running jobs could fail. Consequently, a large number of computing resources consumed are wasted, considering many applications do not support checkpoints or they are not able to save/restart efficiently. The whole supercomputer could enter an unusable state until the filesystem issue is resolved.

B. Potential solutions

There are several potential solutions to this IO issue. They can be mainly grouped into three different levels.

On the system level, users expect to work with a strong filesystem that can handle any kind of IO requests from all users without losing significant efficiency. Blue Gene from IBM employed a separate network for IO communication to make the filesystem relatively more robust and not interfered by routine application inter-node communications [15]. Alam et al. replaced SATA disk with PCI-based SSD to boost the MDS performance by about two times [4]. However, it is still impractical or too expensive to achieve much higher performance boost (one to two orders). Meanwhile, filesystem designers are also planning for some self-protection features in the future product that can protect the filesystem from IO-intensive workload.

On the application level, it is expected that all supercomputer applications are developed with a well-designed workflow with reasonable IO workload. This is probably the best way to complete the prospective IO work. Authors of this paper also recommended this way to many supercomputer collaborators and helped them re-design their workflow correspondingly. Nevertheless, this kind of development work requires a lot of expertise on both the application side and system side. For many programs that have existed for quite a long time, it is hard and time-consuming to rewrite all those legacy programs. For instance, C++ IO stream is used for IO work throughout the OpenFOAM with over one million lines of code [5]. It is difficult to quickly redesign and rewrite the whole IO workflow.

Fortunately, more and more domain experts and engineers have realized the importance of this IO issue and contributed more time to improving it. Some tool and middleware systems were introduced to reduce the metadata workload and improve IO performance [13], [24], [40]. Instead of generating many separate files, using a file container like hdf5 [30] that accommodates many datasets can effectively decrease the number of requests to access MDS. The user community of HDF and NetCDF developed convenient parallel versions of these data formats and leaned to use them more in daily usage [12], [30], [31], [35]. A collated file format for OpenFOAM Parallel I/O was also introduced to the development version to reduce the number of output files lately [33].

On the user level, some advanced users took advantage of extra storage devices in addition to the shared parallel filesystem, like the local disk drive on each compute node, to complete the prospective IO work [25]. In many other cases, users were forced to give up some prospective IO work by either reducing the IO work frequency or reducing the amount of data at each IO step. Users may also reduce the overall scale of the problem or the number of jobs running simultaneously. This is usually conducive to avoiding excessive IO requests, but it needs a lot of users' effort to proactively improve existing code or workflow, profiling their jobs, and carefully tune parameters. Unfortunately, only a small number of users are equipped with such expertise.

III. Our method and solution

Though previous solutions have been proven to be helpful to the user community, their shortcomings are significant. The solutions on the system level are usually expensive and the solutions on the user/application level are usually troublesome and time-consuming. Particularly, many effective solutions require a lot of workflow changes and code modification, which is almost impossible for those applications users who are not familiar with the details of application programs.

In this paper, we present an innovative tool to solve the described IO problem, hopefully once and for all. The tool is named "**O**ptimal **O**verloaded **IO** **P**rotection **S**ystem (or just **OOOPS**)". It is capable of automatically detecting and restricting intensive IO workload from supercomputer users on the user side to protect parallel shared filesystems on the system side without changing users' application code or existing workflow.

A. Overall design and implementation

The Portable Operating System Interface (POSIX) is defined for maintaining compatibility between operating systems [9]. POSIX IO is part of the standard for the IO interface for POSIX compliant applications. Low-level IO-related functions, like $open()$, $stat()$, $close()$, $read()$, $write()$, are defined and implemented in GNU C Library (glibc) [27]. Most of the high-level IO functions on supercomputers, like $fopen()$, $fwrite()$, $MPI\_File\_open()$, $MPI\_File\_write()$ are implemented based on these functions under the hood. OOOPS controls the IO workload from supercomputer users by the means of directly intercepting frequently used POSIX IO functions and proactively scheduling user applications IO requests.

Specifically, we firstly intercept these low-level IO related functions defined in glibc. Then, we keep a record of the real-time IO operation time (or the response time), as well as the real-time IO operation frequency (based on recent IO requests count and time stamp). Then we check the filesystem status, evaluating whether it is busy, modest used, or idle based on the instant response time per operation. Meanwhile, we evaluate users' real-time IO workload based on the recent IO requests count and frequency, which are node-based and user-based. If the system is too busy and/or the current frequency of IO request is too high, we insert decent delays to these IO related calls to protect the shared filesystem.

By setting the environment variable $LD\_PRELOAD$ to the path of the OOOPS shared library, we can force user applications to load the IO-related function in the OOOPS library over the default ones implemented in the glibc library [22]. Fig. 2 shows the function interception workflow with the OOOPS system. Assuming that there is a parallel application program implemented with MPI, there exist multiple IO-related functions, like $MPI\_File\_open()$, in the program. Under the layer, $MPI\_File\_open()$ calls the function $open()$ defined in the glibc library. Without OOOPS, the glibc version of $open()$ is used to complete the IO work. With OOOPS loaded, the workflow is intercepted. Once the application needs to call the $open()$ function, it jumps to the $open()$ function we define in the OOOPS library. Besides calling the glibc version of the $open()$ function, the $open()$ function in our OOOPS library evaluates the filesystem status and workload status. A proper delay is introduced when necessary. Other than the $LD\_PRELOAD$ trick, trampoline is also an excellent alternative method to intercept function [8].

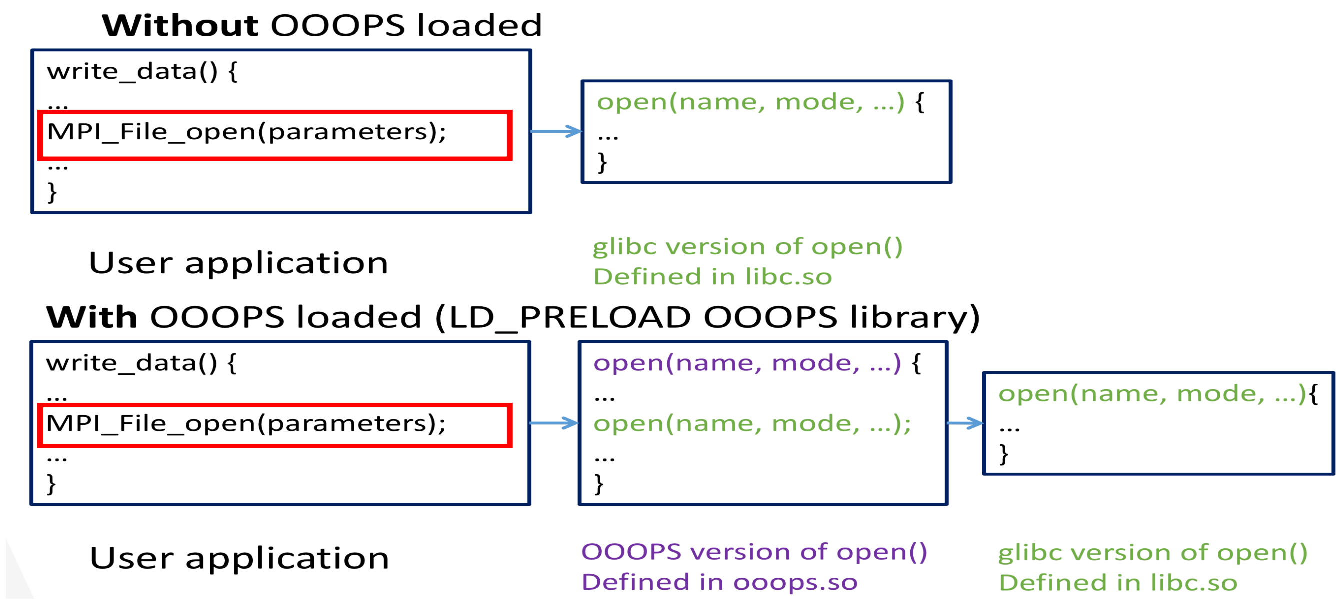


Fig. 2. This figure presents the interception pattern of OOOPS when an IO-related function is called within uses' application. The function implemented in OOOPS library with the same name will be called first.

A simplified version of the new $open()$ is presented in **Function 1** showing the basic structure of the functions defined in the OOOPS library. This OOOPS version of $open()$ calls the glibc version of $open()$ first to accomplish the real IO work. Then it obtains the instant response time and IO request frequency. The function finally collects and analyzes the filesystem server status. If the server is busy and/or the instant IO request frequency is too high, this function inserts proper delays and decreases the frequency of generating new IO requests.

Function 1 Simplified version of the open() function pesudo code defined in OOOPS

```
1: call the open() function in glibc;
2: get_IO_response_time();
3: get_IO_request_counts();
4: collect_and_analyze();
5: if server_busy or io_frequency_high then
6:     insert decent delays;
7: end if
```

B. Tuning parameters

One key issue of OOOPS is the standard to evaluate the filesystem status, i.e., whether the system is busy and whether an instant IO frequency is too high. In the current version of the OOOPS library, we mainly intercept two IO functions $open()$ and $stat()$ as well as their variants. Therefore, we need a set of thresholds of response time for these two functions and user requests count limit to evaluate the status of the filesystem and instant user requests.

To get a better view of the evaluation, we ran a few daylong IO benchmark programs on each target supercomputer system to collect some IO related data. As an example, we collected and analyzed the running time distribution of the $open()$ functions on the Stampede2 system from TACC. Stampede2 with two main sets of compute nodes, the Intel Knight Landing (KNL) and the Intel Skylake (SKX), supports multiple filesystems. We focus on the two productive Lustre filesystems, WORK and SCRATCH of it. Therefore, we have 4 combinations: SKXs on WORK, SKXs on SCRATCH, KNLs on WORK, and KNLs on SCRATCH. Fig. 3 shows one set of the test results of the IO operation count versus the IO operation time for each combination on the Stampede2 system. We can see that the time for the same IO operation differs among different combinations. Furthermore, it can be observed that the IO operation time for each combination is mostly distributed in a relatively small range. We treat this small range to be a normal status. If the IO operation time is beyond that range, i.e. the IO operation time is longer than a preset limit, we treat that IO operation as unusual.

If we divide the distribution by the total count, we can obtain the probability distribution function (PDF) of the operation time. The corresponding cumulative distribution function (CDF) is presented in Fig. 4. We can then pick a threshold of percentage of the overall distribution, 95% for instance, and obtain a threshold time for each compute node-filesystem combination. This threshold value means that 95% of the IO operations are normal and can complete within this time limit. The IO operations that take longer than this time limit are treated as unusually long ones.

Besides the IO operation time, we also set another threshold, the maximum acceptable input/output operations per second (IOPS), defined as a function $S(system, user)$ divided by the IO operation time threshold. The function $S(system, user)$

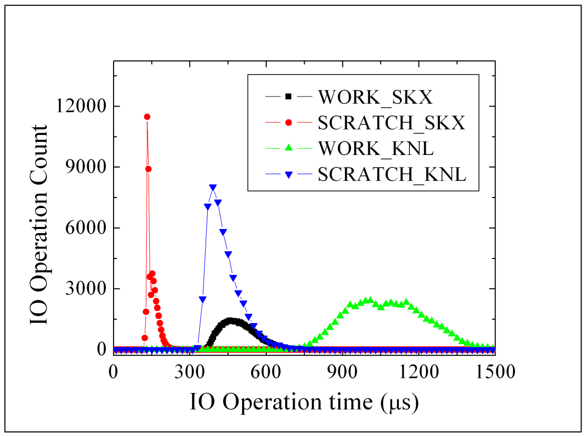


Fig. 3. This figure shows the IO operation count versus the IO operation time from IO applications running on different compute nodes (SKX and KNL) with different filesystems (WORK and SCRATCH) on Stampede2 supercomputer at TACC.

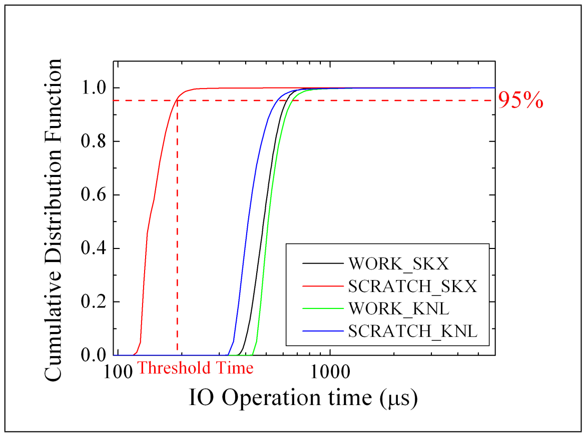


Fig. 4. This figure shows the Cumulative Distribution Function versus the IO operation time from IO applications running on different compute nodes (SKX and KNL) with different filesystems (WORK and SCRATCH) on Stampede2 supercomputer at TACC. This figure also demonstrates how the threshold time is decided.

depends on the filesystem throughput, the system size, the allocation proportion of IO resources, and many other factors. It reflects how the IO resources are allocated among different users or jobs.

In brief, we define a whole set of parameters used as the thresholds for the OOOPS system for each target supercomputer. The threshold values differ for different computational resources and filesystems even on the same supercomputer. The OOOPS system provides the default value on each tested system, but system administrators or advanced users can change or reset them when necessary, Please note that these parameters are system dependent. For each new computing system and new filesystem, a fresh set of parameters are required.

C. Runtime IO workload management

On many supercomputers, users submit long large-scale jobs to the system without knowing that these jobs are IO-intensive and can cause filesystem issues. These jobs may involve some heavy IO work in the middle or at the end of their work. Many system administrators are good at detecting these IO-intensive work when they appear. Unfortunately, the only thing the system administrators can practically do is terminating these hazardous jobs to protect the filesystem and all other users at that time.

With OOOPS enabled on the system, we now provide a better choice for administrators. System administrators can enable or change the IO limitation of a running job to release the pressure of the filesystem instantly. To realize this runtime throttling feature, we keep a record of the OOOPS threshold values under the RAMDisk (/dev/shm) on each allocated compute node. During the run of applications, the OOOPS library checks the threshold values regularly and always employs the latest updated ones.

For the convenience of advanced users, user support teams, or system administrators, we also provide an extra command-line interface to make OOOPS manage IO requests for running jobs. Particularly, the **set_io_params** command can be used to change the threshold values dynamically, even in the middle of the job running without terminating or restarting it. Users' submitted jobs can keep running with a different set of IO limit updated by OOOPS. These jobs that makes excessive IO requests do not need to be killed any more.

The **set_io_params** can be either implemented with specific threshold values or run with a preset IO workload level (low/medium/high(default)/unlimited) as shown below.

```
> set_io_param   [server_idx
                  time_open  max_open_freq
                  time_stat  max_stat_freq]
```

```
> set_io_param  low
> set_io_param  medium
> set_io_param  high
> set_io_param  umlimited
```

IV. Tests and Results

Two main supercomputers we work with in this project are Stampede2 [29] and Maverick2 [28] at TACC, with the WORK and SCRATCH Lustre filesystems. The main reason we choose these supercomputers and filesystems is that the authors are working closely with the administrators team and can obtain proper time windows to implement IO-intensive jobs without disturbing supercomputer users in production. Meanwhile, the metadata contention/bottleneck issue with Lustre filesystems is more significant due to Lustre filesystems' single MDS design [13].

Stampede2 is the flagship supercomputer at TACC [26]. It hosts $4,200$ compute nodes with 68-core Intel Xeon Phi Knights Landing processors and $1,732$ compute nodes with dual 24-core Intel Xeon Skylake processors. The interconnect of the system is a 100 Gb/sec Intel Omni-Path (OPA) network with a fat tree topology. Maverick2 is a new internal test system, having 31 compute nodes at this time, with 24 specialized nodes with 4 Nvidia GTX 1080ti GPUs and a few other nodes with Nvidia $P100$ and $V100$ GPUs.

The WORK filesystem (also named Stockyard) is TACC's global filesystem that is accessible to almost all TACC computing and visualization systems. It provides over 20 petabytes of storage capacity through its aggregate bandwidth of greater than 100 gigabytes per second. The SCRATCH filesystem is a local Lustre filesystem to the Stampede2 system with a capacity of over 30 petabytes.

In this paper, we collected and displayed filesystem input/output operations per second to demonstrate how OOOPS is working. The system utilization information was collected through the resource monitoring tool REMORA (REsource MOnitoring for Remote Applications) [6], which provided convenient interface to gather system utilization data on supercomputers. All presented test cases were ran with and without OOOPS on target systems to demonstrate the improved behavior. These test cases were designed to hold off the filesystem IO limit on purpose.

A. *MPI IO Benchmark*

First of all, we created an IO test program with Message Passing Interface (MPI). In this parallel program, each MPI task repeatably created a new file, wrote some random contents inside the file, and then closed the file. We ran this test program with 64 MPI tasks on 4 Stampede2 Skylake nodes. Each MPI task created and wrote $40,000$ small files on the SCRATCH filesytem and this single job produced millions of instant IO requests to the MDS. We ran the same executable file of this IO program without OOOPS and with OOOPS on three different IO limitation levels. Fig. 5 shows the aggregated IO operations of this program (over all involved compute nodes) collected by REMORA on the Stampede2 system. This figure demonstrates that OOOPS can effectively slow down user's IO requests and restrict the IO operations frequency to a specific level. With different threshold values, we can conveniently change the restriction level as needed. The restriction level can be job-based, user-based, or node-based.

Next, we tested the runtime IO limitation management of OOOPS with the same MPI IO program. At the very beginning, the parallel job ran as planned without throttling. Once the system administrators or user-support team noticed the corresponding IO workload was too heavy, they were able to use the **set_io_params** command to slow down the IO work immediately. Fig. 6 shows the IO operation count per second of one single job with OOOPS changing the IO limitation in the middle of the job (three times for different restriction levels). The IO work was restricted during runtime, but the job finished normally without interruption.

B. *OpenFOAM cavity run*

The next test is the two-dimensional lid-driven cavity case of OpenFOAM [16], which is the most common test case in all OpenFOAM tutorial and documentation. This test case was built with the latest OpenFOAM version 6.0, the Intel-18 compiler, and the Intel MPI stack. In this test case, the mesh

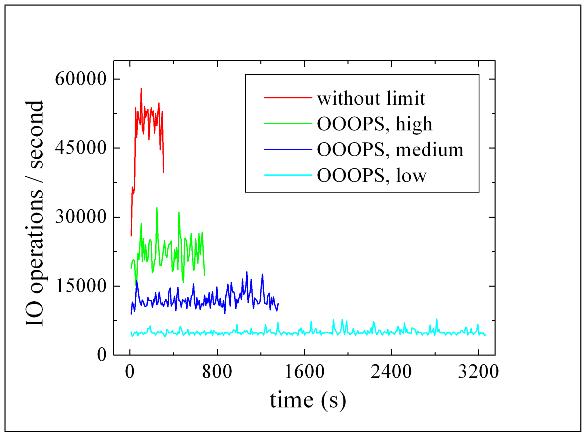


Fig. 5. This figure shows the aggregated IO operations per second on Stampede2 SCRATCH filesystem from the MPI IO benchmark running with 4 Skylake compute nodes. Identical programs ran without OOOPS and with OOOPS (three different levels).

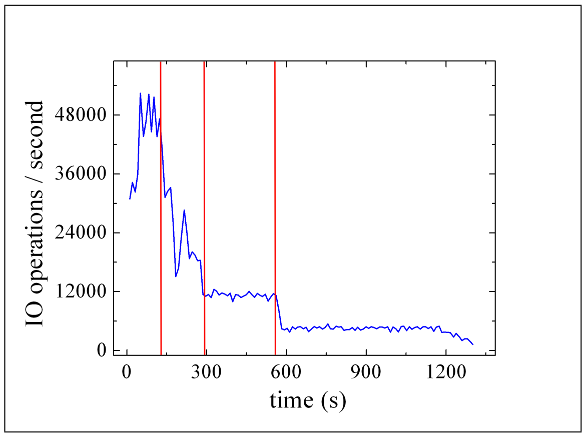


Fig. 6. This figure shows the IO operation per second for one single MPI job. OOOPS changes the IO operation limit of this running job three times without interruption.

was defined on $400 \times 400 \times 1 = 160,000$ cells. Then we used the domain decomposition method to divide the whole domain into $256 (16 \times 16)$ subdomains. The simulation was then ran with 256 cores on 16 Skylake compute nodes. The **startTime** is 0 and the **endTime** is 0.5 with iteration **timestep** equal to 0.0001 second. The pressure field, the velocity field, and the mesh information are recorded in this test case. These field data are written to individual files on the SCRATCH filesystem from each processor at each timestep.

Though it is a simple OpenFOAM program running less than 20 minutes (on 16 Stampede2 Skylake compute nodes), it generates over 5 million individual files. Fig. 7 shows the IO workload of this cavity case running on the Stampede2 system without and with OOOPS. In this test case, OOOPS successfully limited the IOPS to around 23 thousand, while the original IOPS was over 42 thousand.

Running IO-intensive applications like OpenFOAM with OOOPS makes the IO requests more smooth and manageable. For complicated problems (more fields to record) and large-

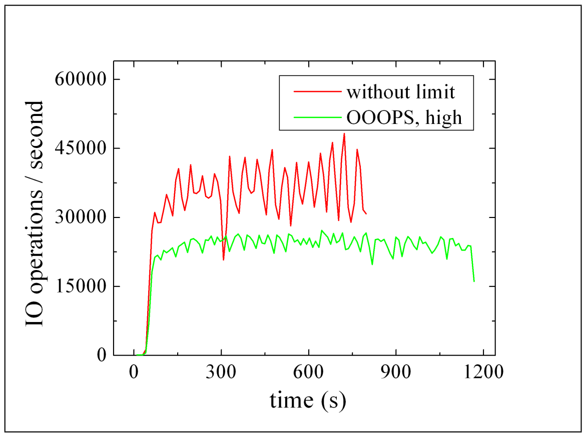


Fig. 7. This figure shows the IO operations per second of the OpenFOAM cavity case running with and without OOOPS. This figure demonstrates how OOOPS effectively manages the IO workload in a productive OpenFOAM run.

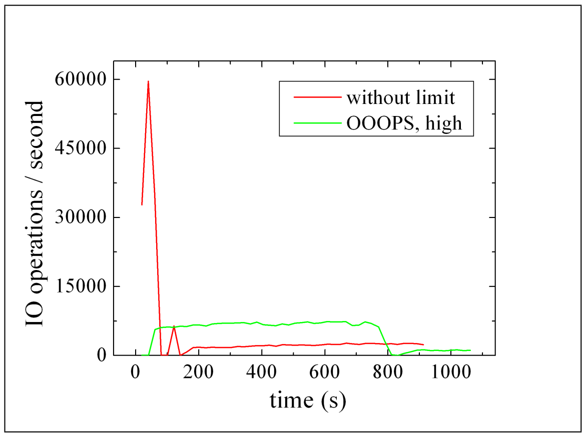


Fig. 8. This figure shows the IO operations per second of training ResNet-50 for Imagenet with and without OOOPS. This figure demonstrates how OOOPS effectively manage the IO workload during the early training stage.

scale runs (more individual tasks to write) in which the IO requests could be enormous, OOOPS could be more beneficial.

C. ResNet-50 for Imagenet

ImageNet [17] is a widely-used visual database designed for visual object recognition software research. There are a total of $1,281,167$ images for training and $50,000$ images for validation. Residual neural network models (ResNet-50, ResNet-101, and ResNet-152) [11] were used in ILSVRC and COCO 2015 competitions [34], which won the 1st place in ImageNet classification, ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation Deep Residual Learning. With balanced accuracy and complexity, ResNet-50 for ImageNet has become a popular example to demonstrate how HPC can greatly shorten the training time in deep learning research [39].

We have tested the training of Resnet-50 for ImageNet with Keras [7] (version 2.2.0) and TensorFlow [3] (version 1.8.0) with two nodes on the Maverick2 supercomputer. Eight MPI tasks were launched to match the number of GPUs available on two nodes. At the beginning, Keras enumerated all files under the training and validation directory and called $stat()$ for every file. High workload on MDS was observed in the first several minutes on every compute node corresponding to more than $1,331,167$ times calling function $stat()$ as shown in Fig.8. When the model training started, the MDS workload dropped significantly to a reasonable level. However, the training job can generate excessive IO workload if we scale out to run with several hundred nodes. From our profiling result, the time spending on file I/O could range from 1% to 30% (or more) of total running time depending on the I/O bandwidth provided. If the speed of reading files is largely throttled, the total wall time could be noticeably extended. OOOPS does effectively alleviate the filesystem workload with the cost of slowing down training jobs.

For such workload that constantly generates excessive IO requests, a better solution would be storing all data sets on local storage (including SSD, HDD, RAMDisk, etc.) when possible. Therefore, there will be no load on centralized filesystem and network from I/O process. If there is not enough space to accommodate all data locally, we can distribute data among multiple compute nodes and set up distributed filesystem with allocated compute nodes. In the case of no local storage available at all, the issue of millions of $stat()$ calls on each node can be fixed without too much trouble. Such $stat()$ for all files can be called only one time and stored into a separate file. Similar to what we already implemented in this work, we can design a library that can intercept all $stat()$ calls and directly return the results of $stat()$ by looking up a pre-compiled hash table instead of requesting file stat information from filesystems that are already in our list.

V. Highlights of OOOPS

As a user support tool, OOOPS is very convenient from users' perspective. Source code modification or workflow update for complicated IO work, which is usually necessary but troublesome, is not required at all. After being configured on target supercomputer systems, OOOPS can automatically introduce self-driven slowdown in the middle of heavy IO work when necessary. With the help of OOOPS, many users' supercomputer assignment, which are originally forbidden, can now accomplish without hurting the filesystem.

Meanwhile, OOOPS is valuable to supercomputers. It is capable of protecting parallel filesystems from overloaded IO requests. It is easy to deploy on almost any kind of clusters system-wide. There is little work for system administrators to maintain or run OOOPS. Benefited by the design of a user-side solution, OOOPS is scalable to deploy on large-scale supercomputers. Since OOOPS is tackling filesystem issues from the root cause, it can throttle excessive IO requests from problematic applications very effectively. With the dynamic control function, system administrators can also restrict running jobs' IO requests in real time without interruption. it is also flexible to add new features within OOOPS later, since all OOOPS work is distributed over all available nodes instead of a few service nodes.

VI. Future work

The current version of the product mainly focuses on protecting parallel shared filesystems from overloaded metadata requests. The chosen parameters are preset and relatively restrictive. In our future work, we will optimize our tools to carry out the systems with more relaxed and dynamic parameters that further cut down unnecessary IO delays and reach a better balance between protecting filesystems and fully employing available machine utilization.

With some simple extension, it would be possible to realize IO resource provisioning for specific users or applications with customized configurations instead of enforcing a universal configuration system-wide. We are also considering to intercept $read()$ and $write()$ function calls to eventually implement read/write bandwidth resource provisioning on the user, application, or node level in the near future.

Furthermore, we will also collect and analyze the usage data from the filesystem server side and combine with the usage data from the user side. By integrating the information together, we will be able to infer the relationship between users' IO work and filesystem's usage status, and finally optimize the performance of the whole filesystem.

VII. Conclusion

Modern supercomputers are required to support numerous data-intensive applications with diverse IO workload and IO patterns. Heavy IO work on parallel filesystems becomes more and more common in the modern supercomputer environment, where simultaneous access to large or numerous files from multiple compute nodes is required. In this paper, we demonstrate our recent work of an innovative user-support tool "Optimal Overloaded IO Protection System (OOOPS)" for managing heavy IO workload on parallel shared filesystems. This tool is employed to help supercomputer users carry out heavy IO work that was originally forbidden or unsuited on modern supercomputers. It also significantly assists administrators in protecting the supercomputer filesystems from overload.

With OOOPS, we practically enforce an IO resource provisioning policy on all active users that share the parallel filesystems. The metadata server throughput is allocated as a resource for system stability, which results in noteworthy improvement of capacity on supercomputers.

VIII. Acknowledgment

We would like to thank our colleagues from Texas Advanced Computing Center who provided expertise and insight that greatly assisted this work. Particularly, we would like to show our gratitude to Junseong Heo, who helped us design, test, and debug the early versions of the product. We are also immensely grateful to Zhao Zhang, Tommy Minyard, Robert McLay, Hang Liu, and William Barth for their suggestions and comments that greatly improved the work. In addition, we would like to express our sincere gratitude to all system administrators and user support team members from the National Center for Atmospheric Research, the Johns Hopkins University, the University of Arizona that support related work.

References

[1] IOR: Parallel filesystem I/O benchmark. https://github.com/LLNL/ior. [Online; accessed 13-Aug-2017].

[2] The OrangeFS Project. http://www.orangefs.org. [Online; accessed 13-Aug-2017].

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[4] Alam, Sadaf R. and El-Harake, Hussein N. and Howard, Kristopher and Stringfellow, Neil and Verzelloni, Fabio. Parallel I/O and the Metadata Wall. In *Proceedings of the Sixth Workshop on Parallel Data Storage*, PDSW 11, pages 13–18, 2011.

[5] Bjørn Lindi. I/O-profiling with Darshan. *PRACE report*.

[6] Carlos Rosales and Antonio Gómez-Iglesias and Andrew Predoehl. Remora: a resource monitoring tool for everyone. *HUST '15 Proceedings of the Second International Workshop on HPC User Support Tools, SC15*, 2015.

[7] François Chollet et al. Keras: The python deep learning library. https://keras.io, 2015. [Online; accessed 13-Aug-2017].

[8] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. In *Third USENIX Windows NT Symposium*. USENIX, 1999.

[9] The Open Group. POSIX Frequently Asked Questions. http://www.opengroup.org/austin/papers/posix_faq.html, 2011. [Online; accessed 13-Aug-2018].

[10] IBM. IBM Spectrum Scale. https://developer.ibm.com/storage/products/ibm-spectrum-scale. [Online; accessed 13-Aug-2017].

[11] Kaiming He and Xiangyu Zhang and Shaoqing Ren and Jian Sun. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[12] Jianwei Li, Wei keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, and Rob Latham. Parallel netcdf: A scientific high-performance I/O interface. *Proceedings of the ACM/IEEE SC2003 Conference*, 2003.

[13] Jay Lofstead, Fang Zheng, Scott Klasky, and Karsten Schwan. Adaptable, metadata rich IO methods for portable high performance IO. *2009 IEEE International Symposium on Parallel and Distributed Processing*, 2009.

[14] Marat Valiev and Eric J. Bylaska and Niranjan Govind and Karol Kowalski and Tjerk P. Straatsma and H. J. J. Van Dam and Dong Wang and Jarek Nieplocha and Edoardo Aprá and Theresa L. Windus and Wibe A. deJong. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181:1477–1489, 2010.

[15] José E. Moreira, Michael Brutman, José G. Castaños, Thomas Engelsiepen, Mark Giampapa, Thomas Gooding, Roger L. Haskin, Todd Inglett, Derek Lieber, Patrick McCarthy, Michael Mundy, Jeff Parker, and Brian P. Wallenfelt. Designing a Highly-Scalable Operating System: The Blue Gene/L Story. *ACM/IEEE SC 2006 Conference (SC'06)*, pages 53–53, 2006.

[16] NTNU HPC GROUP. OpenFOAM - Cavity Tutorial. https://www.hpc.ntnu.no/display/hpc/OpenFOAM+-+Cavity+Tutorial. [Online; accessed 13-Aug-2017].

[17] Olga Russakovsky and Jia Deng and Hao Su and Jonathan Krause and Sanjeev Satheesh and Sean Ma and Zhiheng Huang and Andrej Karpathy and Aditya Khosla and Michael Bernstein and Alexander C. Berg and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[18] OpenCFD Ltd (ESI Group). The Open Source CFD Toolbox . https://openfoam.com, 2014-2018. [Online; accessed 13-Aug-2017].

[19] OpensFS. Lustre File System. http://opensfs.org/lustre/. [Online; accessed 13-Aug-2017].

[20] PANASAS. PanFS Parallel File System. https://www.panasas.com/products/architecture/panfs-parallel-file-system, note = ”[Online; accessed 13-Aug-2017]”.

[21] Peter Braam. Lustre: A Scalable, High-Performance File System. *Lustre Whitepaper Version 1.0*, 2012.

[22] Peter Goldsborough. The LD_PRELOAD trick. http://www.goldsborough.me/c/low-level/kernel/2016/08/29/16-48-53-the_-ld_preload-_trick, 2016. [Online; accessed 19-Aug-2017].

[23] Philippe Wautelet. Introduction to parallel filesystems. http://www.idris.fr/media/docs/docu/idris/idris_patc_filesystems_proj.pdf. [Online; accessed 13-Aug-2017].

[24] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. Indexfs: scaling file system metadata performance with stateless caching and bulk insertion. *In Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC14)*, 2014.

[25] Si Liu and John Cazes and Greg Foss and Greg Abram and Donald Cook and Craig Stair. Extremely high-resolution weather model simulation, data processing, and visualization. *95th American Meteorological Society Annual Meeting*, 2015.

[26] Dan Stanzione, Bill Barth, Niall Gaffney, Kelly Gaither, Chris Hempel, Tommy Minyard, Susan. Mehringer, Eric Wernert, Henry Tufo, D.K. Panda, and Patricia Jane Teller. Stampede 2: The evolution of an xsede supercomputer. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, PEARC17, pages 15:1–15:8. ACM, 2017.

[27] The GNU Operating System. The GNU C library manual: Low-Level I/O. https://www.gnu.org/software/libc/manual/html_node/Low_002dLevel-I_002fO.html, 2017. [Online; accessed 13-Aug-2018].

[28] Texas Advanced Computing Center. Maverick2 User Guide. https://portal.tacc.utexas.edu/user-guides/maverick2, 2018. [Online; accessed 13-Aug-2018].

[29] Texas Advanced Computing Center. Stampede2 User Guide. https://portal.tacc.utexas.edu/user-guides/stampede2, 2018. [Online; accessed 13-Aug-2018].

[30] The HDF Group. HDF5. https://support.hdfgroup.org/HDF5/. [Online; accessed 13-Aug-2017].

[31] The HDF Group. Parallel HDF5. https://support.hdfgroup.org/HDF5/PHDF5/. [Online; accessed 13-Aug-2017].

[32] The OpenFOAM Foundation Ltd. OpenFOAM and The OpenFOAM Foundation. https://openfoam.org, 2011-2018. [Online; accessed 19-Aug-2017].

[33] The OpenFOAM Foundation Ltd. OpenFOAM Parallel I/O. https://openfoam.org/news/parallel-io/, 2017. [Online; accessed 13-Aug-2017].

[34] UNC Chapel Hill Vision Lab. Large Scale Visual Recognition Challenge 2015. http://image-net.org/challenges/LSVRC/2015/, 2015.

[35] Unidata. Network Common Data Form (NetCDF). https://www.unidata.ucar.edu/software/netcdf/. [Online; accessed 13-Aug-2017].

[36] Wikipedia contributors. IBM Spectrum Scale. https://en.wikipedia.org/wiki/IBM_Spectrum_Scale. [Online; accessed 13-Aug-2017].

[37] Wikipedia contributors. Lustre (file system). https://en.wikipedia.org/wiki/Lustre_(file_system). [Online; accessed 13-Aug-2017].

[38] Wikipedia contributors. Parallel Virtual File System. https://en.wikipedia.org/wiki/Parallel_Virtual_File_System. [Online; accessed 13-Aug-2017].

[39] You, Yang and Zhang, Zhao and Hsieh, Cho-Jui and Demmel, James and Keutzer, Kurt. ImageNet Training in Minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 1:1–1:10, New York, NY, USA, 2018. ACM.

[40] Fang Zheng, Hasan Abbasi, Jay F. Lofstead Ciprian Docan, Qing Liu, Scott Klasky, Manish Parashar, Norbert Podhorszki, Karsten Schwan, and Matthew Wolf. Predata – preparatory data analytics on peta- scale machines. *Proceedings of the IEEE International Symposium on Parallel Distributed Processing*, 2010.