`iez`: Resource Contention Aware Load Balancing for Large-Scale Parallel File Systems

Bharti Wadhwa[†], Arnab K. Paul[†], Sarah Neuwirth[⋆], Feiyi Wang[‡],
Sarp Oral[‡], Ali R. Butt[†], Jon Bernard[†], Kirk W. Cameron[†]
[†]*Virginia Tech*, [⋆]*University of Heidelberg, Germany*, [‡]*Oak Ridge National Laboratory*

Abstract—**Parallel I/O performance is crucial to sustaining scientific applications on large-scale High-Performance Computing (HPC) systems. However, I/O load imbalance in the underlying distributed and shared storage systems can significantly reduce overall application performance. There are two conflicting challenges to mitigate this load imbalance: (i) optimizing system-wide data placement to maximize the bandwidth advantages of distributed storage servers, i.e., allocating I/O resources efficiently across applications and job runs; and (ii) optimizing client-centric data movement to minimize I/O load request latency between clients and servers, i.e., allocating I/O resources efficiently in service to a single application and job run. Moreover, existing approaches that require application changes limit wide-spread adoption in commercial or proprietary deployments. We propose `iez`, an "end-to-end control plane" where clients transparently and adaptively write to a set of selected I/O servers to achieve balanced data placement. Our control plane leverages real-time load information for distributed storage server global data placement while our design model leverages trace-based optimization techniques to minimize I/O load request latency between clients and servers. We evaluate our proposed system on an experimental cluster for two common use cases: synthetic I/O benchmark IOR for large sequential writes and a scientific application I/O kernel, HACC-I/O. Results show read and write performance improvements of up to 34% and 32%, respectively, compared to the state of the art.**

I. Introduction

Load imbalance and poor resource allocation have been identified as major causes of performance degradation in many HPC storage systems, including Lustre [1], the widely-used parallel file system for scientific computing. A number of recent works [2]–[5] have targeted load imbalance in Lustre[1]. Some aim to simultaneously perform resource allocation for all concurrently running applications on the system (i.e., server-side approaches [2], [3]). These include the default approach adopted in Lustre's request ordering system, the Network Resource Scheduler (NRS) [6], which reorders incoming I/O requests on the server-side. Other techniques attempt to minimize resource contention on per-application basis (i.e., client-side approaches [4], [5]). Unfortunately, while client- and server-side approaches in isolation work well for some applications, the diversity of I/O workloads (e.g., large sequential writes from many clients versus scientific data processing on a large data set for a single application) leads to situations where both isolated approaches suffer reduced performance.

[1]Note that while we focus on Lustre given its wide-spread use, our approach is applicable to and can be extended for use in other related HPC parallel file systems.

We propose a novel end-to-end control plane, `iez`, that combines the application-centric strengths of client-side approaches with the system-centric strengths of server-side approaches. To this end, we faced a number of challenges. First, existing storage servers lack a mechanism for global coordination. Second, at any given time, the file system simultaneously serves multiple, asynchronous, diverse client applications with no mechanism for data placement coordination. Furthermore, in the case of Lustre, each client application leverages somewhat opaque high-level I/O libraries such as HDF5 [7] and MPI-I/O [8]. Third, in contrast to existing approaches that modify the application, to enable adoption in real deployments, the end-to-end control plane techniques must maintain application portability while providing efficient data placement.

`iez` supports three key functions to address the aforementioned challenges. First, it provides an application-agnostic global view of all resources to the MetaData Server (MDS). This includes the current statistics of the distributed Object Storage Targets (OSTs) that are managed by Object Storage Servers (OSSs) for storing the application data. Second, it automatically coordinates all the clients in a scalable fashion in order to optimize job placement strategies on per-client basis. Third, `iez` employs a user-level library to intercept file I/O calls (metadata operations) and transparently provide runtime optimization for the client application, thereby maintaining portability.

More precisely, the proposed system gathers real time information from clients and servers about the applications' storage requirements as well as the load on storage servers and maps the present and future job requests on OSTs in a balanced manner to provide efficient utilization across a set of servers. The system considers per-client job requests in a dynamic client-wide prediction model to synchronize holistic job placement and resource allocation. Our data placement strategy supports two widely-used classes of application file access, i.e., File-Per-Process (FPP), and Single-Shared-File (SSF) per job request. Moreover, our approach also supports commonly used I/O interfaces such as POSIX-IO [9], MPI-I/O [8] and HDF5 [7].

Specifically, we make the following contributions.

1) We introduce an end-to-end control plane, `iez`, to optimize I/O resource allocation in existing as well as next generation HPC systems that use Lustre for their storage platform.

2) We present the detailed design of `iez` for the Lustre file system, which incorporates both server side and client side functionality to optimize I/O data placement and job placement.
3) We implement and evaluate the effectiveness of `iez` on a cluster with two commonly used use cases: synthetic I/O benchmark IOR [10] and a scientific application I/O kernel HACC-I/O [11]. Results show that `iez` improves I/O performance by up to 34% compared to the extant round-robin based load balancing adopted in Lustre.

II. Background and Motivation

Recent studies [3], [4], [12]–[14] have shown that load imbalance in HPC systems creates resource contention and degrades overall performance. The complex path of an application I/O request—comprising myriad components such as I/O libraries, network resources, and backend storage—is a significant bottleneck. In today's HPC deployments, there is no global I/O coordinator to handle the overall resource contention problem. Thus, existing parallel file and storage systems can only partially optimize some portions of the I/O path, but not the entire end-to-end path. For example, *Network Request Scheduler* (NRS) [6] balances load in the distributed network of Lustre file system, i.e., on the server side; whereas *Balanced Placement I/O* (BPIO) [4] performs load balancing on per-application basis, i.e., on the client side.

A. Lustre Architecture

We have implemented `iez` atop Lustre, the parallel file system deployed most widely in the world's top supercomputing systems [15]. Lustre is a scalable storage platform that is based on distributed object-based storage. Figure 1 shows a high-level overview of the Lustre architecture and its key components. Lustre clients provide an interface between applications and the storage servers. The application data is managed by two types of servers, MDS and OSS. MDS manages all name space operations and stores the name space metadata on one or more storage servers called Metadata Targets (MDT). The bulk storage of contents of application data files is provided by OSSs. Each OSS manages a number of OSTs and stores the data on one or more OSTs. OSTs are a kind of direct-attached storage. Each data file can be striped across multiple OSTs, with the stripe count specified by the user. The distributed components are typically connected via a high-speed data network protocol, LNet [16], which supports a host of networks, e.g., Ethernet, Infiniband [17], etc.

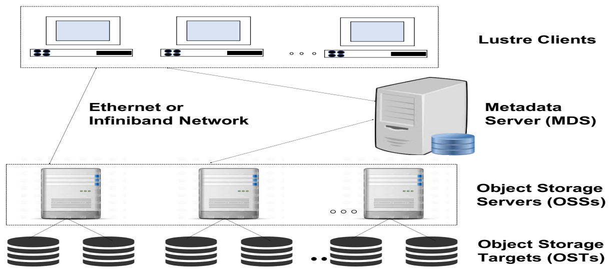

Lustre Clients
Ethernet or Infiniband Network
Metadata Server (MDS)
Object Storage Servers (OSSs)
Object Storage Targets (OSTs)

Fig. 1: Overview of Lustre architecture.

B. Approaches for Load Balancing in Lustre

Given typical non-uniform data accesses patterns, the striping of application data across OSTs give rise to load imbalance in most cases. Currently, the default OST load-balancing approach adopted in Lustre is round-robin (RR), also known as RAID 0. The main limitation is that, RR aims to balance the load of OSTs only, i.e., without any consideration to the load on other components, e.g., MDS and OSS. Moreover, our earlier work on quantitatively studying HPC I/O behavior [2] has shown that RR can take a long time to balance a system, and it is unable to capture the complex application behavior. Consequently, the default policy falls short of providing the desired I/O balanced system.

I/O load balance in scalable parallel file systems is being studied extensively [18]. One approach is to address the problem from the client side on per job basis [19]–[22]. The applications' I/O calls can be intercepted from client side during runtime and the OSTs assignment can be managed accordingly to mitigate resource contention [23]–[26]. An example of such an approach is TAPP-I/O [5]. TAPP-I/O intercepts file I/O calls (metadata operations) during runtime, supports both statically and dynamically linked applications, and provides an automatic placement strategy for both FPP and SSF I/O modes. However, the main limitation is that these approaches do not consider the requirements of other applications running simultaneously on the system due to lack of a global view of the storage servers.

Conversely, another approach is to have a global view of storage servers and consider the load balance across all applications instead of per-application basis. For this, the load balancing problem can be handled from server side [2], [3], [27], [28]. The main limitations of such approaches are that they require modification of application source code, and do not consider the different file I/O layouts (SSF or FPP).

The above approaches improve the I/O performance of Lustre by effectively reducing the resource contention and improving load balance, but fail to exploit the opportunity for end-to-end I/O path optimization. In contrast, `iez` aims to provide an end-to-end load balancing solution, which globally coordinates between clients and servers of parallel file and storage systems.

C. Use Cases and Benchmarks

We use a representative synthetic I/O benchmark IOR [10] as well as a real-world scientific I/O application kernel HACC-I/O [11] for presenting the design and implementation of `iez`.

1) HACC-I/O: The *Hardware Accelerated Cosmology Code* (HACC) [11] application uses N-body techniques to simulate the formation of structure in collision-less fluids under the influence of gravity in an expanding universe. HACC-I/O captures the I/O patterns and evaluates the performance for the HACC simulation code. It utilizes the MPI-I/O interface and differentiates between FPP and SSF parallel I/O modes.

2) IOR: The *InterleavedOrRandom* (IOR) [10] benchmark provides a flexible way to measure parallel file system's I/O performance. It measures the I/O performance with different parameter configurations including I/O interfaces ranging from traditional POSIX to advanced parallel I/O interfaces like MPI-I/O. It performs reads and writes to and from files on parallel file systems like Lustre, and provides the throughput rates.

D. Load Imbalance in a Default Lustre Deployment

In order to highlight the load imbalance in a default Lustre deployment, we conducted a quantitative study that uses RR to distribute I/O load on OSTs. We deployed a testbed of 6-node Lustre cluster, with 1 MDS, 3 OSSs and 2 Clients. Each OSS in our cluster manages 5 OSTs with a capacity of 10 GB each. Hence, the cluster has 15 OSTs in total with a total capacity of 150 GB. We ran IOR benchmark with 8, 16, and 64 processes that produce 16, 32, and 64 GB of data, respectively, to be stored on the OSTs in the FPP access mode. We also ran HACC-I/O application for 8, 16, and 32 processes for 10 M particles.

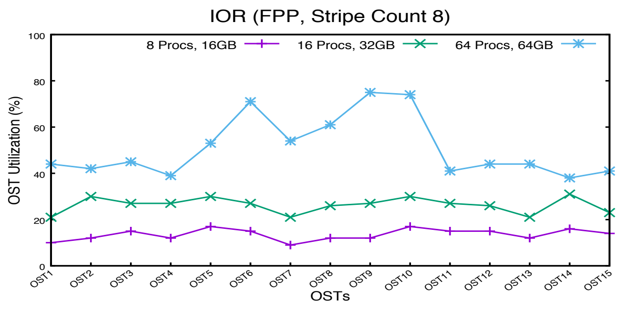


Fig. 2: OST Utilization under RR for IOR.

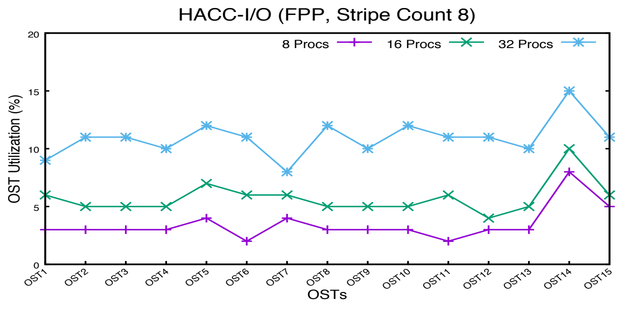


Fig. 3: OST Utilization under RR for HACC-I/O.

Figure 2 shows the storage utilization in each OST, for different number of processes and a stripe count of 8 for IOR in FPP mode. In a balanced load setting, these graphs would be straight lines (representing ideal load balance), but in the studied scenario, the load is observed to be imbalanced with some OSTs getting a lot more load than others. A similar pattern can be seen in Figure 3 for HACC-I/O for FPP mode with stripe count of 8.

These results show that with the default Lustre deployment that uses RR scheduling to allocate OSTs for each job, there can be a significant load imbalance at the server level. The load imbalance persists at different scales and different stripe-count and thus can lead to imbalanced resource usage and resource contention.

III. System Design

We have implemented `iez` for the widely-used Lustre file system. However, our design can be extended for use in other HPC distributed storage and I/O systems that employ a similar hierarchical structure.

Figure 4 shows an overview of the `iez` architecture. It presents an "end-to-end" control plane for managing I/O with components both on the client side and the server side. When running applications for the first time, the client side makes use of a customized tracing tool, `miniRecorder`. This tool collects information about the I/O accesses, such as the number of bytes written, file name, number of stripes, and MPI rank and communicator for each file. `MiniRecorder` needs to collect traces only for the first run of an application. `iez` identifies an application's I/O behavior, which does not change across multiple runs of an application. The collected traces are fed into the Client (C)-Parser that then uses the information to drive the prediction algorithm. Our predictions are based on ARIMA time series modeling [29]. The output of the time series prediction provides estimates of future application requests, which are stored in an "interaction database" for later use by `iez`. We refer to the database as "interaction database" because it offers a point of interaction between our server-side and client-side libraries.

On the server side, OSSs collect the CPU and memory usage information, associated OSTs capacity (*kbytestotal*) and the number of bytes available on the OSTs (*kbytesavail*). These statistics are sent to the MDS. The collected information on the MDS is parsed using Server (S)-Parser and fed to the OST allocation algorithm. The input to the OST allocation algorithm is the predicted set of requests received from the clients via HTTP-based interactions, and the output is the OSTs to be allocated for every request, which will yield a load-balanced distribution over the involved OSSs and OSTs. The allocated OSTs are stored in the interaction database along with the predicted requests. Next, the placement library, `iez-PL`, intercepts the I/O requests from the applications, consults the interaction database, and routes the application requests to appropriate resources.

A. Client-Side `iez` Components

In the following, we describe the `iez` components that run on the clients.

1) Tracing Tool: We implement a simple I/O tracing library, `miniRecorder`, based on Recorder [30]. Recorder is a multi-level I/O tracing framework, which can capture I/O function calls at multiple levels of the I/O stack, including HDF5, MPI-IO, and POSIX I/O. For our end-to-end system, we limit the range of intercepted function calls to file creation and write calls, and record the bytes written, file name, stripe count, and MPI rank and communicator for each file. We focus on writes more than reads because caching mechanisms and burst buffers that are typical in modern HPC deployments absorb most of the read requests once the file has been written. Therefore, the load imbalance is mainly due to write requests [2], [30]. The traced data is processed by the *C-Parser*

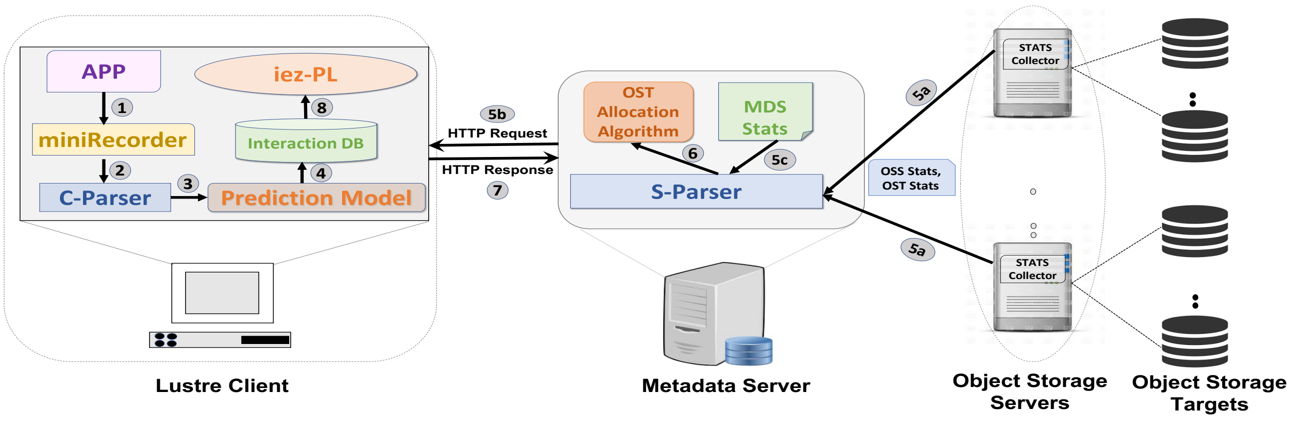


Fig. 4: Overview of the proposed `iez` architecture.

and converted into a readable (comma separated) *.csv* format file. This file is then sent to the prediction library (AIPA-ARIMA Inspired Prediction Algorithm discussed in the next section). The tracing tool is lightweight, and our tests show that it adds a negligible memory and CPU overhead of $\sim 0.3\%$ and $<1\%$, respectively, during application execution.

*2) ARIMA-Inspired Prediction Algorithm (*AIPA*):* HPC applications have been known to show distinct I/O patterns [2]. Based on our interactions with HPC practitioners, this predictability is expected for emerging applications as well. We leverage this observation to model three key properties of HPC I/O, namely write bytes, stripe count, and MPI rank. We collect these parameters using the tracing tool on the first run of an application to train our model. We use AutoRegressive Integrated Moving Average (ARIMA) model to fit our multivariate time series data and predict future values. Our choice of ARIMA is dictated by its performance and the time-series nature of the write bytes and stripe count of the requests. We experimented with several alternatives, such as the Markov Chain Model [31], to model the data. However, ARIMA yielded better accuracy with lower memory and computing overhead. For example, we observed a 99.1% accuracy in IOR data using ARIMA, while Markov chain model yielded an accuracy of 95.5%. The CPU overhead for ARIMA was less than 1.2% compared to 4.5% in Markov Chain, while the memory usage for ARIMA is 10 MB in comparison to 90 MB usage in Markov chain.

We implement our prediction model on the client side, where the calls would be intercepted, rather than on MDS. This has two advantages: i) since each application has its own client, the model can be applied at scale without overwhelming the centralized MDS; and (ii) the approach makes the MDS application-agnostic, where the server can focus on write requests and types in a global fashion and not be concerned with individual applications. The approach also provides for a much more efficient solution when multiple applications run on (multiple) clients simultaneously.

A time series is defined as a sequential set of data points, measured typically over successive times. It is represented as a set of vectors $x(t)$, $t = 0, 1, 2...$, where t is the elapsed time [32]. The term `ARIMA` involves three parts, `AR` denotes that the variable is regressed on its prior values, `I` stands for 'integrated', which means that the data values are replaced with the difference between the present and previous values, and `MA` represents the fact that the regression error is a linear combination of error terms occurring in the past. There are three parameters used for every ARIMA model. The parameter '`p`' is the number of lag observations (lag order), '`d`' denotes the number of times raw observations are differenced (degree of differencing), and '`q`' represents the size of moving average window (order of moving average).

The first step is to select the values of parameters `(p, d, q)`. To this end, we run the model on all combinations (skipping the ones that fail to converge) of the parameters over our dataset, which we get from the tracing tool, and select the combination with the least Root Mean Square Error (RMSE). We vary the values of *p*, *d*, and *q* from 0 to 5. We go from 0 to 5 for all the values because going beyond 5 would be computationally expensive. For *HACC-IO*, the least RMSE was found for `(5, 1, 2)` and *IOR* gave the minimum RMSE for `(2, 1, 1)`. The next step is to fit the ARIMA`(p, q, d)` model by exact maximum likelihood via Kalman filters [33]. This fitted model is then used to predict the write bytes, stripe count, and MPI rank values for future application I/Os. We use `statsmodels.tsa.arima_model` package in Python for our ARIMA implementation. Our results show a 98.3% accuracy in HACC-I/O data and 99.1% accuracy in IOR data.

3) Interaction Database: The interaction database is an SQL database located on the Lustre clients. It serves as the medium through which the MDS and clients interact with one another. First, the values predicted by AIPA are stored in the database. Table I shows an example snapshot of the interaction database for HACC-I/O in FPP mode. We store the file names, the number of bytes written, number of stripes associated with every file, and the MPI Rank. The MDS uses the HTTP protocol to access the interaction database. The process starts with the MDS sending a HTTP Request to the client to retrieve the required database contents. The specific steps are discussed in Section III-B3. For our implementation, we use MySQL 8.0.12 Community Server Edition. Our results show that writing and retrieving data from the interaction

TABLE I: Interaction Database Snapshot showing write requests for HACC-I/O in FPP mode.

File Name	Write Bytes	Stripe Count	MPI Rank
/lustre/scratch/hacc-io/FPP1-Part00000000-of-00000008.data	803405824	8	0
/lustre/scratch/hacc-io/FPP1-Part00000001-of-00000008.data	803405824	8	1
/lustre/scratch/hacc-io/FPP1-Part00000002-of-00000008.data	803405824	8	2
/lustre/scratch/hacc-io/FPP1-Part00000003-of-00000008.data	803405824	8	3
/lustre/scratch/hacc-io/FPP1-Part00000004-of-00000008.data	803405824	8	4
/lustre/scratch/hacc-io/FPP1-Part00000005-of-00000008.data	803405824	8	5
/lustre/scratch/hacc-io/FPP1-Part00000006-of-00000008.data	803405824	8	6
/lustre/scratch/hacc-io/FPP1-Part00000007-of-00000008.data	803405824	8	7

```
Input: File Name $file$, Access mode $flags$
Output: Call real metadata operation, e.g., $open()$
begin
    if fileExits(file) == TRUE then
        return realMetadataOperation($file$, $flags$)
    end
    $r$ = $queryInteractionDatabase(file)$
    $layout \rightarrow scount$ = $row \rightarrow stripe\_count$
    $layout \rightarrow stripe\_size$ = $row \rightarrow stripe\_size$
    $layout \rightarrow ost\_list$ = $row \rightarrow ost\_indices$
    createInode($layout$, $flags$, $mode$)
    return realMetadataOperation($file$, $flags$)
end
Function createInode
    Input: File Layout $layout$, Access mode $flags$
    $mode$ = 0644
    $flags$ = $flags$ | O_LOV_DELAY_CREATE
    $request$ = LL_IOC_LOV_SETSTRIPE
    $fd$ = __real_open($file$, $flags$, $mode$)
    ioctl($fd$, $request$, $placement$)
    close($fd$)
```

Algorithm 1: File layout and Lustre inode creation.

database is very efficient, using $<0.3\%$ and $<0.4\%$ of CPU and memory, respectively.

4) Placement Library: The placement library, `iez-PL`, complements the prediction model by providing a lightweight and user-friendly mechanism to place an application's I/O workload in accordance with the predicted set of OSTs. `iez-PL` utilizes function interposition to prioritize itself over standard function calls, and the profiling interface to MPI (PMPI) for MPI and MPI-IO routines. `iez-PL` is implemented as a shared, dynamic user library, and can be used by specifying it as the preloading library via the environment variable `LD_PRELOAD`. The POSIX I/O, MPI-IO, and HDF5 metadata operations (e.g., `open()`), which are issued by the application, are intercepted and re-routed to `iez-PL` for processing purposes. For every I/O cycle, the library queries the interaction database with the file name passed by the function call and fetches the corresponding row with the predicted placement information. This information includes the stripe count, stripe size, and a list of OST indices. Next, `iez-PL` creates a Lustre inode on the MDS and places the stripes on the OSTs as returned by the prediction model. For this purpose, Lustre provides the user library *llapi*, which allows the user to describe a specific file layout, i.e., the striping pattern. However, `iez-PL` cannot directly take advantage of the user library, since *llapi* calls `open()` internally, which means that the preloading mechanism would cause an endless loop. Therefore, `iez-PL` mimics the behavior of *llapi* and communicates directly with the Lustre Logical Object Volume (LOV) client software layer to create the Lustre inode. Algorithm 1 presents a simplified overview of the preloading library, which is run on every client. If the file does not exist yet, it queries the prediction database to receive the predicted layout information. The next step is to create the Lustre inode by issuing the `ioctl()` system call with the Lustre-specific request code `LL_IOC_LOV_SETSTRIPE`, which allocates a Lustre file descriptor and applies the striping pattern to the file-to-be-created. In the last step, `iez-PL` forwards the call to the original metadata operation (e.g., `open()` or `MPI_File_open()`). Currently, `iez-PL` supports the following I/O function calls: `open[64]()`, `creat[64]()`, `MPI_File_open()`, and `H5Fcreate()`. The key advantage of this transparent approach is that user applications can directly benefit from the prediction model without modifying the source code.

B. Server-Side `iez` Components

In the following, we describe the `iez` components that run on the servers (OSSs and MDS) and how they interact with each other.

1) Statistics Collection: Statistics collection is done for every OSS. The list of all the OSTs for a particular OSS is saved in a configuration file that is provided as input to the collector for that specific OSS along with the OSS id. For every OST, we collect the total and available capacity, found in the files $/proc/fs/lustre/obdfilter/ost\_name/kbytestotal$ and $/proc/fs/lustre/obdfilter/ost\_name/kbytesavail$, respectively. We also collect the CPU and memory utilization of the OSS by reading data from the files $/proc/meminfo$ and $/proc/loadavg$. We save all the statistics in a space separated string *statistics*. The statistics collection algorithm runs every 60 seconds on all the OSSs. We choose 60 seconds as our interval for statistics collection so that we get updated statistics on the MDS without over-loading the OSSs. These statistics are sent to the MDS.

The load monitoring (statistics collection) solution needs to be scalable. Therefore, we use a publisher-subscriber model [34] for the statistics collection framework. This is shown in Figure 5. OSSs act as publishers and MDS as the subscriber. Statistics collected in the OSSs are sent to the MDS via a message queue. We use ZeroMQ (ϕMQ) [35] as our message queue because it is lightweight and has been shown to be very efficient at large scale. We also collect the CPU and memory utilization of the MDS every 60 seconds, in the same way as it is collected in the OSSs. Our tests with the implementation show that the statistics collection framework

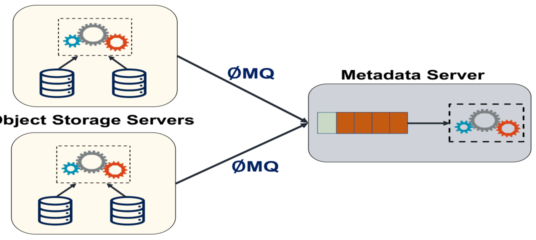


Fig. 5: Publisher-Subscriber model for statistics collection.

on average has negligible CPU and 0.1% memory utilization on the OSSs, and 0.6% CPU and 0.1% memory on the MDS.

2) Statistics Parser: The *S-Parser* in the MDS is responsible for handling the following:

- Statistics collected from the MDS.
- Statistics collected in the OSSs and published to the MDS via ZeroMQ.
- Current and predicted write requests retrieved from the clients via HTTP.

The CPU and memory utilization collected in the MDS is important to determine when the OST allocation algorithm will run. We allow the allocation algorithm to run only if the CPU utilization is lower than 70% and memory utilization is lower than 50%. This is done so that the load balancing algorithm does not disrupt the normal functionalities of Lustre's MDS. The parsed statistics of the OSS and the write requests from the clients are sent as input to the OST allocation algorithm (Section III-B3). Parsing is done whenever unparsed data arrives at the MDS, therefore should be done very efficiently. Our results show that the parser has a CPU utilization of 0.1% and negligible memory utilization.

3) OST Allocation Algorithm: The OST allocation algorithm runs only when the CPU utilization goes below 70% to ensure no interruption to Lustre's default activities. The input to the algorithm is the parsed OSS and OST statistics, and the write requests (file name, write bytes and stripe count) sent by the clients. Before we present the details of our algorithm, we discuss a key practical constraint imposed by Lustre.

a) Lustre's 64k alignment constraint for stripe size: Stripe size is an important parameter for load balancing in a distributed file system. Every stripe needs to be assigned to an OST. Intuitively, in order to calculate stripe size, we can divide the total file size by the stripe count. Both of these parameters are given as input to the OST allocation algorithm. But calculating stripe size is not that simple because Lustre imposes the constraint that in order to place stripes into the allocated OSTs, stripe size should be even multiples of 64k. We term 64k or 65536 bytes as Alignment Parameter (AP). This constraint becomes a problem for files which are not AP aligned, for example in Table I, $803,405,824$ is not an even multiple of AP. We consider two ways to overcome this constraint.

Method-I: This method assumes that we can allocate equal number of even multiples of AP into the first *(stripeCount - 1)* number of OSTs, and the remaining even multiple of AP goes into the last OST. Equation 1 gives the total allocation such that the stripes are AP aligned. The first part of the equation is the placement on first *(stripeCount - 1)* number of OSTs and the second part is the number of bytes written on the last OST.

$$writeBytes = (AP * 2 * N * (stripeCount - 1)) + \\ (AP * 2 * X) \tag{1}$$

where,

$$N = \left\lfloor \frac{writeBytes}{AP * 2 * (stripeCount - 1)} \right\rfloor \tag{2}$$

The remaining number of bytes to be written on the last OST is then given by:

$$remainingBytes = writeBytes - \\ (AP * 2 * (stripeCount - 1)) \tag{3}$$

Therefore,

$$X = \left\lceil \frac{remainingBytes}{AP * 2} \right\rceil \tag{4}$$

Note that we round down the allocation in the first *(stripeCount - 1)* number of OSTs and round up the allocation in the last OST. Multiplication with 2 ensures that the stripe size is an even multiple of AP. Stripe size for the last OST is $(AP * 2 * X)$, and for each of the remaining OSTs is $(AP * 2 * N)$. However, a major drawback here is that this method does not place the load evenly among all the OSTs. The number of bytes written on the last OST will always be smaller compared to the bytes written on the other OSTs for a particular file.

Method-II: This method overcomes the drawback of the previous method by allocating even multiple of AP in all the OSTs.

$$writeBytes = AP * 2 * N * stripeCount \tag{5}$$

where,

$$N = \left\lceil \frac{writeBytes}{AP * 2 * stripeCount} \right\rceil \tag{6}$$

Stripe size for all the OSTs is given by $(AP * 2 * N)$. Therefore, both methods ensure an even multiple of 64k alignment of stripe size for all the stripes allocated in the *stripeCount* number of OSTs, by allocating a little bigger file than was requested by the client. The second method places all the stripes equally on all the OSTs but needs a bigger file to be allocated in comparison to the first method. Thus, there is a trade-off between how big the file allocation we can allow versus balancing all the stripes among the OSTs. Our results show that for a 766.175 MB file size, we allocated 833 KB (0.1%) bigger file using the second method and 63 KB (0.008%) more in the first method. We proceed with the second method because in spite of allocating a little more than was requested by the client, this approach ensures allocating equal stripes on all the OSTs. This would lead to similar load accesses from all OSTs and OSSs, therefore approaching towards a load-balanced setup.

Algorithm 2 shows the the OST allocation algorithm, which employs a minimum-cost maximum-flow approach [36]. The

```
Input: OSS statistics $cpu$ & $mem$, OST statistics $totalKbytes$ &
       $kbytesAvail$, Write Requests $writeBytes$ & $stripeCount$
Output: $OSTAllocationList$
begin
    for request r in WriteRequests do
        $stripeSize$ = calculateStripeSize($writeBytes$, $stripeCount$)
    end
    for OSS oss in OSSList do
        $ossLoad = (cpuweight * cpu) + (memweight * mem)$
        for OST ost in OSTList do
            $ostCostToReach = OSSLoad$
            $ostCost = (totalKbytes - kbytesAvail)/totalKbytes$
            $ostCapacity = kbytesAvail/stripeSize$
        end
    end
    flowGraph = buildGraph(Requests, OSS, OST)
    OSTAllocationList = minCostMaxFlow(flowGraph)
    return ($OSTAllocationList$)
end
Function calculateStripeSize
    Input: writeBytes $w$, StripeCount $sc$
    Output: $stripeSize$
    $stripeSize = ceil(w/(AP * 2 * sc))$
    return ($stripeSize$)
Function buildGraph
    Input: Requests $req$, StripeCount $sc$, OSTCostToReach $ossLoad$, ostCost
           $ostLoad$, OSTCapacity $ostCap$
    Output: FlowGraph $G$
    totalDemand = sum of stripeCount for all Requests
    G.addNode('source', totalDemand)
    G.addNode('sink', -totalDemand)
    for request r in reg do
        G.addEdge('source', $r$, cost = 0, capacity = sc)
        for OST ost in ostList do
            G.addEdge($r$, $ost$, cost = ossLoad, capacity = 1)
        end
    end
    for OST ost in ostList do
        G.addEdge($ost$, 'sink', cost = ostLoad, capacity = ostCap)
    end
    return ($G$)
```

Algorithm 2: Obtaining list of OSTs for each request.

flow graph that is used to solve the problem is shown in Figure 6. We calculate the *stripeSize* based on Method-II described above, cost to reach an OST (which is the load of the OSS), cost of an OST (ratio of bytes already used in the OST to the total size of the OST), and capacity of an OST (the number of stripes that can be handled by the OST, given by the ratio of available space in the OST to the stripe size). In order to derive the flow graph, we need to identify the *source* and *sink* nodes. The total demand for the *source* node is the total number of stripes requested, and the total demand for the *sink* node is the negative amount of the total number of stripes requested. We solve the minimum-cost maximum-flow using the Ford-Fulkerson algorithm [37]. This outputs a list of OSTs (*OSTAllocationList*) using which will yield a balanced load over both OSS and OSTs. For our implementation, we use the `networkx` library in Python. Our results show that the algorithm on average uses 1.58% CPU and 0.1% memory on the MDS.

The list of OSTs obtained from the OST allocation algorithm, along with the *stripeSize*, are then sent to the respective clients where they are stored in the interaction database. An example entry for the database with the complete allocation for a HACC-I/O application in FPP mode is shown in Table II. We replace the *Write Bytes* in the database with the *stripeSize* and add a new column *OST List*. The *OST List* is a space separated load-balanced list of OSTs for every write request. This example is for a setup with 3 OSSs and 15 OSTs (5 OSTs

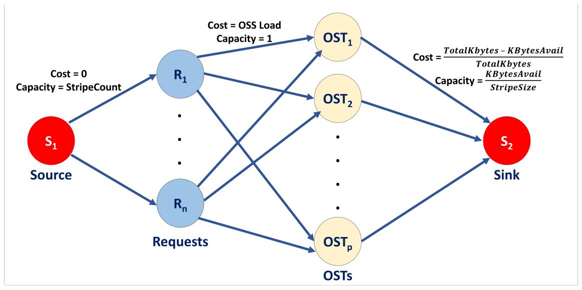


Fig. 6: Graph used in OST Allocation Algorithm.

associated with every OSS) – therefore, OST ids range from 1 to 15. As described earlier, the placement library (`iez-PL`) then uses this information to place the requests, thus completing the load-balanced allocation of resources. If for any run of the application, `iez-PL` is unable to find more than 50% of files in the interaction database, `miniRecorder`, AIPA and OST Allocation Algorithm will be executed again to update the interaction database.

Summary: `iez` components run both on the client and server side of a Lustre deployment. `MiniRecorder` along with AIPA runs on the clients during the first execution of an application (or if the system cannot find prediction information for most of the accessed files) to capture its I/O characteristics. The client and server libraries of `iez` interact using the Interaction Database. Statistics collection on the OSS and MDS happens periodically. Whenever, the interaction database is updated, it sends the new values to the MDS, which (if not overloaded) proceeds to parsing the most recent statistics via the Statistics Parser. The OST Allocation Algorithm is then executed and the results are sent back to the interaction database. For subsequent application runs, the placement library intercepts the application write calls, reads from the interaction database and writes the request to a load-balanced set of OSTs.

IV. Evaluation

We evaluate `iez` using a real Lustre deployment testbed. We use a Lustre cluster of 6 nodes with 1 MDS, 3 OSSs and 2 clients. All of the nodes run CentOS 7 atop a machine with 8 cores, 3.2 GHz processor, and 16 GB memory. Furthermore, each OSS has 5 OSTs, each supporting 10 GB of attached storage. Our tests use HACC-I/O kernel and IOR benchmark in FPP and SSF access modes.

To the best of our knowledge, `iez` is the first work to consider a global view of all the system resources in deciding application request stripe placement. Existing approaches (BPIO [4], TAPP-I/O [5] etc.) balance load among I/O servers on per-application basis, i.e., on the client side, and do not consider the global view, i.e., the requirements of the other applications as well as the OSS and OST states. Moreover, unlike `iez`, such client-side techniques cannot handle multiple simultaneous applications. Thus, these are not directly comparable to `iez`. For these reasons, we choose to use the default RR approach of Lustre as the basis for our comparison.

To capture the degree of load balancing across participating OSTs for a particular test run, we define a metric, *OSTCost*, as

TABLE II: Interaction Database Snapshot showing OST Allocation for HACC-I/O in FPP mode.

File Name	Write Bytes	Stripe Count	MPI Rank	OST List
/lustre/scratch/hacc-io/FPP1-Part00000000-of-00000008.data	100532224	8	0	8 13 12 9 7 4 3 5
/lustre/scratch/hacc-io/FPP1-Part00000001-of-00000008.data	100532224	8	1	11 2 15 10 1 6 8 13
/lustre/scratch/hacc-io/FPP1-Part00000002-of-00000008.data	100532224	8	2	12 9 7 4 3 11 2 5
/lustre/scratch/hacc-io/FPP1-Part00000003-of-00000008.data	100532224	8	3	1 6 15 10 8 13 12 9
/lustre/scratch/hacc-io/FPP1-Part00000004-of-00000008.data	100532224	8	4	7 4 3 11 2 1 6 5
/lustre/scratch/hacc-io/FPP1-Part00000005-of-00000008.data	100532224	8	5	15 10 8 13 12 9 7 4
/lustre/scratch/hacc-io/FPP1-Part00000006-of-00000008.data	100532224	8	6	3 11 2 1 6 5 15 10
/lustre/scratch/hacc-io/FPP1-Part00000007-of-00000008.data	100532224	8	7	14 8 13 12 9 7 4 3

the ratio of the maximum utilization of any OST to the mean utilization of all the OSTs. Therefore,

$$OSTCost = \frac{MaxOSTUtil}{MeanOSTUtil} \tag{7}$$

An ideal load balanced system has the *OSTCost* of 1. We also define *OST Utilization* of an OST as the storage used by the client application on the OST as a fraction of the total storage available on the OST.

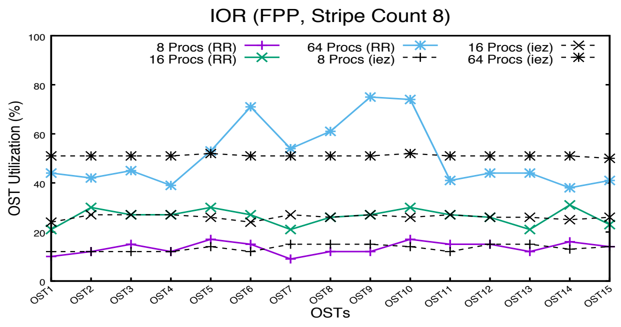


Fig. 7: OST Storage Utilization for IOR in FPP mode and Stripe Count 8.

A. Load Balance for FPP Access IOR

Figure 7 shows the comparison of load under `iez` and the default RR data allocation on 15 OSTs in the Lustre cluster represented as *OST Utilization* for the IOR benchmark with varying stripe count, processes, and data sizes. We see that `iez` balances the load on all OSTs in a near-optimal manner. For example, for 64 processes, the maximum load observed with RR approach is on OST-9: the OST utilization is 75%, while the mean utilization is 51.06%, resulting in the *OSTCost* of 1.47. In contrast, the maximum load observed under `iez` is 52% on OST-5 and OST-10 with the corresponding *OSTCost* of 1.01, i.e., near optimal. The almost horizontal line for *OST Utilization* for `iez` underscores its effectiveness. Overall, `iez` was able to reduce the *OSTCost* by 31.3% compared to the default RR approach.

We observed similar results while running IOR with stripe count of 4. As shown in Figure 8, `iez` is able to distribute data on all 15 OSTs in a balanced way for other studied cases as well. In this case, we observe an *OSTCost* of 1.22 and 1.01 under RR and `iez` respectively. Hence, `iez` provides 17% better *OSTCost* than the default RR approach.

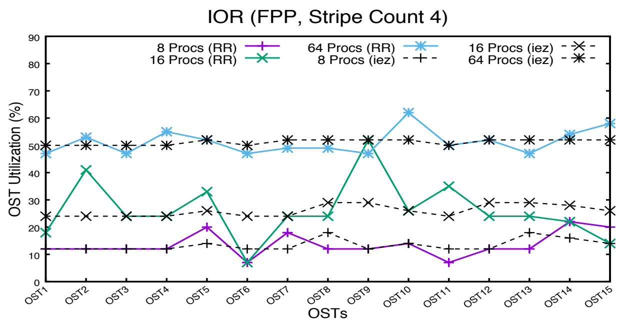


Fig. 8: OST Storage Utilization for IOR in FPP mode and Stripe Count 4.

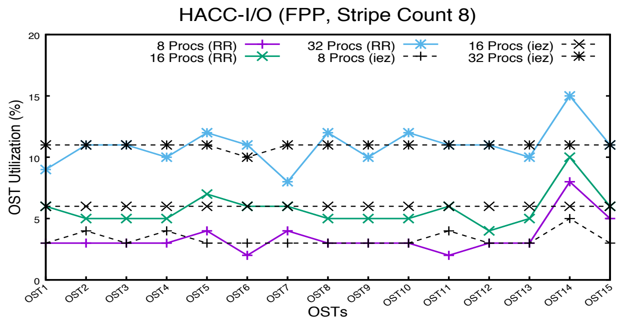


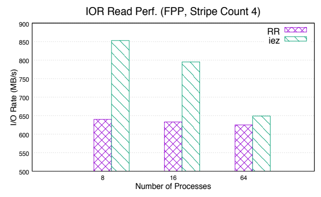
Fig. 9: OST Storage Utilization for HACC-I/O in FPP mode.

B. Load Balance for FPP Access HACC-I/O

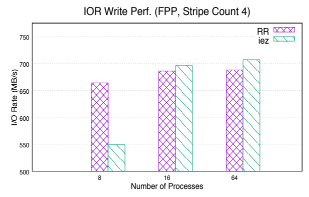
For HACC-I/O, we evaluated `iez` for particle data of 10 M for 8, 16 and 32 processes. Each process creates one data file that is stored in the Lustre OSTs with a stripe count of 8. Total data stored in the files is approximately 4, 7.7, and 13.5 *GB* for 8, 16, and 32 processes, respectively. As for IOR, we observe a significant improvement in load balancing for HACC-I/O as well (shown in Figure 9) compared to the default approach. *OSTCost* for 32 processes with default load balancing approach and `iez` is observed to be 1.37 and 1.00, respectively, with `iez` reducing the OSTCost by 27 %. Similarly, for 16 processes, the *OSTCost* is observed to be almost 1.00 under `iez`.

C. I/O Performance

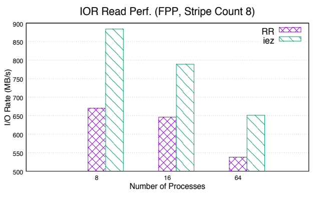
Next, we compared the read and write performance for the studied cases. We measured the 'I/O rate' for storing the data to and reading it from OSTs. Figures 10a and 10b show the read and write performance, respectively, for the IOR benchmark with stripe count 4 and FPP access mode. We observe that even though for smaller scale, there is a small decline in write performance, compared to the default RR approach, `iez`'s I/O rate is 34% higher for reading the



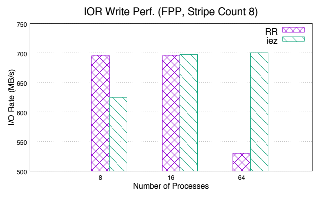
a)



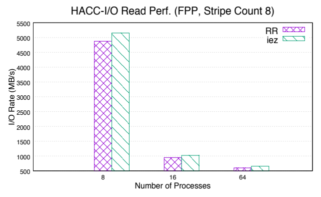
b)



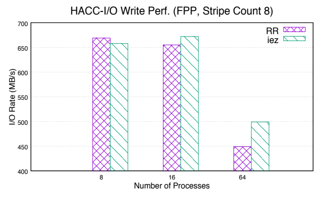
c)



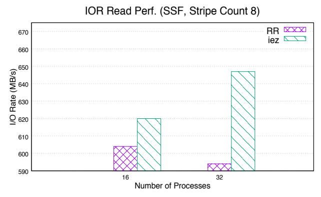
d)



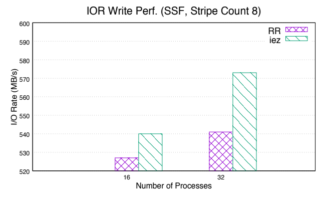
e)



f)



g)



h)

Fig. 10: Read and write performance of IOR and HACC-I/O for FPP and SSF accesses for different Stripe Counts.

data from OSTs. This improvement is achieved due to a balanced load over OSTs and OSSs, which help in mitigating the resource contention and hence improve the parallelism in the data access. Similar results were observed for the IOR benchmark with stripe count 8, as well as for HACC-I/O with stripe count 8. We observe an improvement of up to 32% in the I/O rate while reading the data for IOR with stripe count 8 as shown in Figure 10c. Moreover, in this case, there is improvement of up to 32% in the write performance (Figure 10d).

For HACC-I/O, we observe an improvement of up to 8% for reading (Figure 10e) and 11% for writing (Figure 10f). Here we also observe that the improvement gains vary for different observation points. This is because, OST load depends on a number of factors namely, write-bytes of a request, number of stripes, and MPI-Rank for MPI jobs. Different configurations of a job will yield different jobs requests, and hence different performance improvement, mainly due to varying load and stripe count used in the evaluation.

D. *Utilization of OSSs*

In our next experiment, we measured the storage utilization of each OSS under the default approach and `iez`. This is because we want to balance the load on both OSSs and OSTs for an overall load-balanced setup. To this end, we aggregated the load (storage utilization) on each OST and calculated the ratio of storage being used with respect to the total storage in each OSS. In a balanced scenario, each OSS should be utilized equally by hosting an equal share of application data. We observe that with the default approach the OSSs are imbalanced, while `iez` distributes the application data in a balanced manner across the OSSs. Figure 11 shows the comparison of *OST Utilization* of all 3 OSSs of our testbed under `iez` as compared to the default approach. Here we use the IOR benchmark with 64 processes, FPP access, and stripe count 8 storing a total application data of 64 GB. With the default approach, OSS-2 becomes highly loaded—31% more than mean utilization—as compared to OSS-1 and OSS-3. In contrast, with `iez`, all the three OSSs are utilized in a balanced manner.

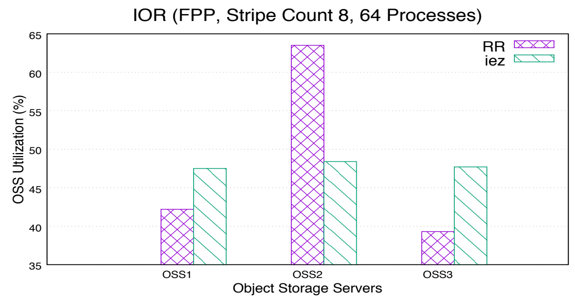


Fig. 11: OSS Storage Utilization for IOR.

E. *Load Balance for Single Shared File Access*

For SSF mode, all the processes write into and read from one shared file. We observe that since there is only one file created and the whole data is striped into almost equal stripes, under the default approach, there is only a little imbalance, as compared to the FPP access. But `iez` is able to eliminate even that imbalance. We ran both IOR and HACC-I/O with different stripe counts to observe the load distribution for SSF access. We found almost same pattern in all cases. Due to space limitation, we present two representative scenarios.

Figure 12 shows the *OST Utilization* of all OSTs for IOR benchmark with 16 processes, creating a file of 32 GB with a stripe count 8. Figure 13 shows the *OST Utilization* of all OSTs for IOR benchmark with 32 processes creating a file of 64 GB with a stripe count 8. For a single shared file access, for both HACC-I/O as well as IOR, all the processes create one single file that is striped across the eight OSTs, since the Lustre stripe count for these two data files is 8. Hence only eight OSTs are allocated to store each of the files, and the unallocated OSTs are not used. For example, as shown in Figure 12, with the default RR allocation scheme, for IOR, the created data file is striped across, OSTs 1, 2, 5, 7, 8, 13, 14, and 15. Out of the 8 allocated OSTs, the OST utilization for OSTs 5, 14 and 15 is greater than 50%, but for the rest of the allocated OSTs is less than 50%. In contrast, while using `iez`, the data file is striped across OSTs 3, 4, 7, 8, 9, 11, 12, and 13. For all of these allocated OSTs, the utilization is the same (47%). Hence, mitigating the imbalance with RR. Figure 13 shows a similar behavior. We also observed that in SSF access mode, `iez` performs better for reading and writing the data with SSF as compared to default approach. Figures 10g and 10h show the read and write performance of `iez` compared to the default approach for 16 and 32 processes, respectively. `iez` improves read and write performance by up to 8% and 6%, respectively, compared to the default approach.

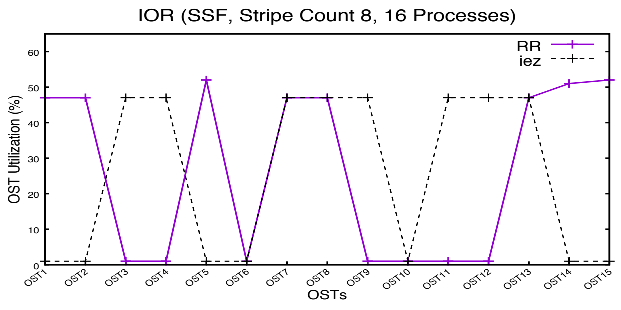


Fig. 12: OST Storage Utilization for IOR in SSF mode for 16 Processes.

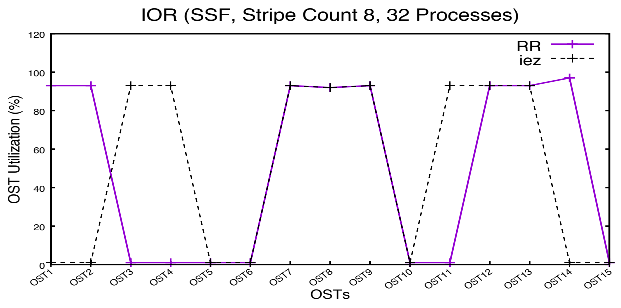


Fig. 13: OST Storage Utilization for IOR in SSF mode for 32 Processes.

F. Load Balance for Concurrent Applications

In our next test, we evaluated `iez` simultaneously running HACC-I/O and IOR with different job configurations from two different Lustre Clients with 16 processes on each. Each process creates one file that is stored on Lustre OSTs with a stripe count of 8 for HACC-I/O and 4 for IOR. The total amount of data stored for each file for HACC-I/O and IOR is 7.7 GB and 32 GB, respectively. As with the single application tests, we observe a significant improvement in load balancing for concurrent applications compared to the default RR approach. Figure 14 shows the comparison of load under both approaches on 15 OSTs in the Lustre cluster. We see that `iez` balances the load on all the OSTs. The maximum load observed with RR is on OST1 with a utilization of 35%, while the mean utilization is 30%, resulting in the *OSTCost* of 1.17. In contrast, under `iez`, the OST load observed on all of the 15 OSTs is 30%, and hence has the *OSTCost* of 1.0. Moreover, the CPU utilization and memory usage on MDS while using `iez` for load balancing in concurrent application runs is observed to be about 1.55% and 0.12%, respectively.

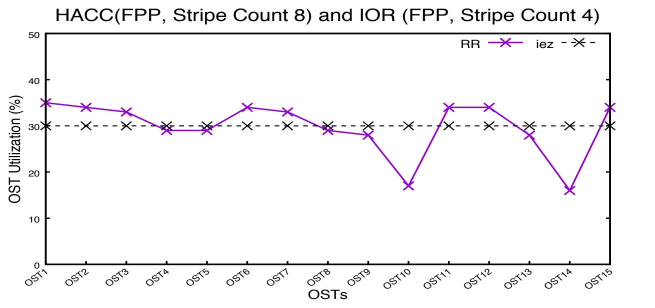


Fig. 14: OST Storage Utilization for a simultaneous IOR and HACC execution in FPP mode with Stripe Count of 4 and 8, respectively.

V. Discussion and Future Work

a) Limitations and Scope of `iez`*:* The key feature of `iez` is the construction of a global view of all system resources in the I/O path, which is then used to drive the data placement decisions and achieve load balancing under global knowledge. Future systems will continue to scale up as they must, and we expect the backend I/O servers and metadata servers will scale up accordingly - Lustre's DNE Phase 1 and 2 (Distributed Namespace Extension) are moving toward that direction. This will inevitably drive up the cost (communication and computation) of obtaining the global view needed by our approach. `iez` will have to take this into consideration and carefully balance the trade-offs, e.g., by adopting a hierarchical or partitioned design where only the view across a set of interacting applications or shared resources is needed.

b) Use of Different Datasets Across Different Runs: We plan to automate `iez` to support multiple runs of applications that use varying datasets as follows. If the number of unseen/new requests in subsequent application runs exceeds a threshold (currently 60%), it re-triggers the learning phase. For now, this re-triggering is manual, but we plan to automate it in our future work.

c) Implementation of `iez` *in other hierarchical HPC file systems:* While the current implementation of `iez` uses the widely-used Lustres monitoring facilities, similar monitoring exists/can be built into other multi-tiered file systems, e.g., BeeGFS [38], IBM GPFS (Spectrum Scale) [39] and Ceph [40]. Even though our particular prototype and evaluation are performed on Lustre platform, but the metrics we have chosen, and the assumptions on the architecture components are mostly generic across the spectrum of large scale parallel file systems, such as decoupling of MDS and data servers (OSDs), data striping, and striping width and length. The monitoring metrics we have chosen can find their equivalents in other systems as well. Case in point: Cephs built-in "ceph mon dump" and GPFSs "mmpmon" and its array of sensors can all provide even greater data collection capabilities, and are amenable with our proposed algorithms.

Our future work also involves evaluating `iez` on large-scale HPC systems such as the ones located at Oak Ridge National Laboratory. In addition, we would expand `iez` to enable Lustre Progressive File Layout (PFL) feature.

VI. Conclusion

We presented the design of an "end-to-end control plane" to optimize parallel and distributed HPC I/O systems, such as Lustre, by providing efficient load balancing across storage servers. Our proposed system, `iez`, provides global view of the system, enables coordination between the clients and servers, and handles the performance degradation due to resource contention by considering operations on both clients as well as servers. Our implementation of `iez` provides a balanced distribution of load over OSTs and OSSs in the Lustre file system. We evaluated `iez` on a real Lustre testbed using two representative benchmarks—IOR and HACC-I/O—with multiple stripe counts of files as well as SSF and FPP accesses. Compared to the default Lustre RR policy, `iez` provides up to 31.3% improvement in balancing the load. Moreover, we also observed an I/O performance improvement of up to 34% for reads and 32% for writes. Finally, the transparent design of `iez` makes it attractive for adoption in real-world deployments where access to application source code, needed for existing approaches, may not always be possible.

Acknowledgment

This work is sponsored in part by the NSF under the grants: CNS-1405697, CNS-1615411, and CNS-1565314/1838271, This research used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at the Oak Ridge National Laboratory, which is supported by the Office of Science of the DOE under Contract DE-AC05-00OR22725.

References

[1] P. J.Braam. The lustre storage architecture (tech. rep.). Technical report, Available: http://wiki.lustre.org/., 2004.

[2] Arnab K Paul, Arpit Goyal, Feiyi Wang, Sarp Oral, Ali R Butt, Michael J Brim, and Sangeetha B Srinivasa. I/o load balancing for big data hpc applications. In *Proc. Big Data 2017*. IEEE, 2017.

[3] Bin Dong, Xiuqiao Li, Qimeng Wu, Limin Xiao, and Li Ruan. A dynamic and adaptive load balancing strategy for parallel file system with large-scale i/o servers. *JPDC*, 72(10):1254–1268, 2012.

[4] Feiyi Wang, Sarp Oral, Saurabh Gupta, Devesh Tiwari, and Sudharshan S Vazhkudai. Improving large-scale storage system performance via topology-aware and balanced data placement. In *Proc. ICPADS 2014*, pages 656–663. IEEE, 2014.

[5] Sarah Neuwirth, Feiyi Wang, Sarp Oral, and Ulrich Bruening. Automatic and transparent resource contention mitigation for improving large-scale parallel file system performance. In *Proc. ICPADS*. IEEE, 2017.

[6] Yingjin Qian, Eric Barton, Tom Wang, Nirant Puntambekar, and Andreas Dilger. A novel network request scheduler for a large scale storage system. *Computer Science-Research and Development*, 23(3-4):143–148, 2009.

[7] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the hdf5 technology suite and its applications. In *Proc. EDBT/ICDT 2011*, AD '11, New York, NY, USA. ACM.

[8] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing mpi-io portably and with high performance. In *Proc. IOPADS*, NY, USA, 1999. ACM.

[9] Stephen R. Walli. The POSIX Family of Standards. *StandardView*, 3(1):11–17, March 1995.

[10] LLNL. Ior benchmark. Accessed: March 12 2018.

[11] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann. Hacc: Extreme scaling and performance across diverse architectures. In *Proc. SC*, Nov 2013.

[12] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R Butt. Mos: Workload-aware elasticity for cloud object stores. In *Proc. HPDC*. ACM, 2017.

[13] Arnab Kumar Paul, Wenjie Zhuang, Luna Xu, Min Li, M Mustafa Rafique, and Ali R Butt. Chopper: Optimizing data partitioning for in-memory data analytics frameworks. In *Proc. CLUSTER*. IEEE, 2016.

[14] Arnab Kumar Paul and Bibhudatta Sahoo. Dynamic virtual machine placement in cloud computing. In *Resource Management and Efficiency in Cloud Computing Environments*. IGI Global, 2017.

[15] Top 500. Top 500 supercomputers. Accessed: July 12 2018.

[16] Lustre Networking. High-performance features and flexible support for a wide array of networks, 2008.

[17] Gregory F Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001.

[18] Ali Anwar. *Towards Efficient and Flexible Object Storage Using Resource and Functional Partitioning*. PhD thesis, Virginia Tech, 2018.

[19] Ali Anwar, Yue Cheng, Hai Huang, Jingoo Han, Hyogi Sim, Dongyoon Lee, Fred Douglis, and Ali R Butt. bespo kv: application tailored scale-out key-value stores. In *Proc. SC*. IEEE, 2018.

[20] Ali Anwar, Yue Cheng, Hai Huang, and Ali Raza Butt. Clusteron: Building highly configurable and reusable clustered data services using simple data nodes. In *Proc. HotStorage*. USENIX, 2016.

[21] Bharti Wadhwa, Suren Byna, and Ali R Butt. Toward transparent data management in multi-layer storage hierarchy of hpc systems. In *Proc. IC2E*. IEEE, 2018.

[22] Nannan Zhao, Ali Anware, Yue Cheng, Mohammed Salman, Daping Li, Jiguang Wan, Changsheng Xie, Xubin He, Feiyi Wang, and Ali Butt. Chameleon: An adaptive wear balancer for flash clusters. In *Proc. IPDPS*. IEEE, 2018.

[23] Mingfa Zhu, Guoying Li, Li Ruan, Ke Xie, and Limin Xiao. Hysf: A striped file assignment strategy for parallel file system with hybrid storage. In *Proc. HPCC_EUC*. IEEE, 2013.

[24] Xiuqiao Li, Limin Xiao, Meikang Qiu, Bin Dong, and Li Ruan. Enabling dynamic file i/o path selection at runtime for parallel file system. *The Journal of Supercomputing*, 68(2):996–1021, 2014.

[25] Shuibing He, Xian-He Sun, and Adnan Haider. Has: Heterogeneity-aware selective data layout scheme for parallel file systems on hybrid servers. In *Proc. IPDPS*. IEEE, 2015.

[26] Yuichi Tsujita, Tatsuhiko Yoshizaki, Keiji Yamamoto, Fumichika Sueyasu, Ryoji Miyazaki, and Atsuya Uno. Alleviating i/o interference through workload-aware striping and load-balancing on parallel file systems. In *ISC*. Springer, 2017.

[27] Zeng Zeng and Bharadwaj Veeravalli. On the design of distributed object placement and load balancing strategies in large-scale networked multimedia storage systems. *IEEE TKDE*, 20(3):369–382, 2008.

[28] Huaiming Song, Yanlong Yin, Xian-He Sun, Rajeev Thakur, and Samuel Lang. A segment-level adaptive data layout scheme for improved load balance in parallel file systems. In *Proc. CCGRID 2011*, pages 414–423. IEEE, 2011.

[29] Peter J Brockwell, Richard A Davis, and Matthew V Calder. *Introduction to time series and forecasting*, volume 2. Springer, 2002.

[30] Huong Luu, Babak Behzad, Ruth Aydt, and Marianne Winslett. A Multi-Level Approach for Understanding I/O Activity in HPC Applications. In *Proc. CLUSTER*. IEEE, September 2013.

[31] Ramesh R Sarukkai. Link prediction and path analysis using markov chains1. *Computer Networks*, 33(1-6):377–386, 2000.

[32] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[33] Andrew C Harvey. *Forecasting, structural time series models and the Kalman filter*. Cambridge university press, 1990.

[34] Arnab K Paul, Steven Tuecke, Ryan Chard, Ali R Butt, Kyle Chard, and Ian Foster. Toward scalable monitoring on large-scale storage for software defined cyberinfrastructure. In *Proc. PDSW-DISCS*. ACM, 2017.

[35] Pieter Hintjens. *ZeroMQ: messaging for many applications*. O'Reilly, 2013.

[36] Ravindra K Ahuja. *Network flows: theory, algorithms, and applications*. Pearson Education, 2017.

[37] Alan Tucker. A note on convergence of the ford-fulkerson flow algorithm. *Mathematics of Operations Research*, 2(2):143–144, 1977.

[38] Jan Heichler. An introduction to beegfs, 2014.

[39] Frank B Schmuck and Roger L Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST*, volume 2, 2002.

[40] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proc. USENIX OSDI 2006*, pages 307–320. USENIX Association, 2006.