2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC) | 978-1-6654-9423-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/HiPC56025.2022.00041

# LuxIO: Intelligent Resource Provisioning and Auto-Configuration for Storage Services

Keith Bateman, Neeraj Rajesh, Jaime Cernuda Garcia, Luke Logan, Jie Ye,
Stephen Herbein *, Anthony Kougkas, Xian-He Sun
Illinois Institute of Technology, * Lawrence Livermore National Laboratory
{kbateman, nrajesh, jcernudagarcia, llogan, jye20}@hawk.iit.edu, stephen@herbein.net,{akougkas, sun}@iit.edu

***Abstract*—Storage in HPC is typically a single Remote and Static Storage (RSS) resource. However, applications demonstrate diverse I/O requirements that can be better served by a multi-storage approach. Current practice employs ephemeral storage systems running on either node-local or shared storage resources. Yet, the burden of provisioning and configuring intermediate storage falls solely on the users, while global job schedulers offer little to no support for custom deployments. This lack of support often leads to over- or under-provisioning of resources and poorly configured storage systems. To mitigate this, we present LuxIO, an intelligent storage resource provisioning and auto-configuration service. LuxIO constructs storage deployments configured to best match I/O requirements. LuxIO-tuned storage services show performance improvements up to 2× across common applications and benchmarks, while introducing minimal overhead of 93.40 ms on top of existing job scheduling pipelines. LuxIO improves resource utilization by up to 25% in select workflows.**

***Keywords*—resource provisioning, I/O behavior, storage auto-tuning**

## I. Introduction

Traditional storage in High Performance Computing (HPC) systems is a consistently available Remote and Static Storage (RSS) resource [1], [2], [3]. Typically, it takes the form of a Parallel File System (PFS). The remote property of RSS indicates that data resides in a permanent, centralized location separated from compute nodes by the network. The static property of RSS indicates that the storage deployment cannot change, implying the I/O requirements of all applications using the RSS do not change either.

However, the ever-increasing diversity in storage device type (e.g., NVMe, PMEM, etc.) and scientific application complexity [4] have put more and more pressure on RSS performance tuning, due to their diverse and often conflicting requirements. Storage hardware innovation has resulted in the compute-node-local deployment of fast storage devices that could significantly improve application I/O performance and promote the deployment of ephemeral I/O services (e.g., scratch filesystems) that last the duration of the job. Local storage has the potential to provide superior performance to remote storage, but only if it is managed appropriately. Modern complex scientific applications present several challenges for storage, since I/O requirements can be either dynamic or self-conflicting between I/O phases of an application; the static aspect of RSS is consequently contested, and custom storage solutions are a better way to optimize performance for such diverse applications [5], [2]. Due to the increased diversity of both the application I/O requirements and hardware composition available in production, there is a need for provisioning, configuring, and deploying custom storage services (which may be ephemeral) tailor-made for a particular set of performance targets. However, configuration and provisioning of optimized custom storage systems is complex due to the aforementioned variety of options. Domain scientists are typically made to do this task, but they are not experts in storage, which leads to a need for a service to automate it.

There are various challenges to address when creating this service. *Challenge 1:* this service requires performant and detailed characterization of I/O behavior in order to relate applications to corresponding storage services. *Challenge 2:* this service also requires a comprehensive classification of storage systems available on the target cluster, incorporating an understanding of the effect of different characteristics on I/O performance for varied I/O workloads. *Challenge 3:* this service needs to bridge the gap between provisioning decisions and the user in a way which improves resource utilization.

To address the aforementioned challenges, this paper presents LuxIO, an intelligent storage resource provisioning and auto-configuration service. LuxIO enables storage system configuration to be performed efficiently and automatically. It does this by suggesting storage deployments which satisfy the I/O requirements of a known application with similar I/O behavior. This is achieved through comprehensive modeling of application I/O behavior and storage performance characteristics. LuxIO handles the deployment and management of storage services via auto-generated deployment scripts. Additionally, LuxIO maintains a database which keeps track of active storage deployments for storage resource sharing. To improve resource utilization, LuxIO leverages various resource models, such as a storage device pricing model and an interference model.

The contributions of LuxIO are:

a) *Application I/O behavior model*: A model capturing the I/O behavior in an application and classifying applications according to their behavioral relationships (section III-B).
b) *Storage quality and semantic model*: A model describing the expected performance characteristics and semantics of storage systems and their relationships to each other (section III-C).
c) *Storage resource provisioning model*: A collection of models that perform intelligent storage resource provisioning. Examples include cost-aware, interference-aware, and performance-aware models. (section III-D).
d) *I/O requirement specification*: A standard for communicating I/O requirements between users, schedulers, and storage systems. This includes I/O Identities for representing application behavior, Storage Service Level Objectives (SSLOs) for representing storage system characteristics, and a storage deployment schema for representing the scheduling needs and requirements of storage. (section III-E).
e) *LuxIO service*: An automatic, transparent-to-user, end-to-end service for the provisioning of storage devices, and for configuration and deployment of storage services based on application I/O behavior (section III-A).

## II. Background and Motivation

### A. Application I/O Characterization

There are various techniques which can be used to profile and characterize the I/O requirements [5] of applications. These include I/O tracing [6], source code and binary analysis [7], [8], and analysis of execution traces and system monitoring logs [9]. Analysis and characterization of data collected from such sources is typically performed manually and offline using various statistical methods. For instance, Liu et al [5] has investigated Darshan traces from Argonne using cumulative density functions and t-SNE visualizations. Darshan is an I/O tracing library and runtime which captures the I/O calls of the application and reports information on various aspects of those calls such as their counts, sizes, and types [6]. It has become fairly ubiquitous, with many supercomputing sites such as Argonne and NERSC employing it as the standard I/O tracing software.

### B. Complexity of Storage Systems

There are a large variety of storage systems available in HPC, such as Ceph [10], Lustre [1], and OrangeFS [11]. Most storage systems have a multitude of configuration parameters (e.g., buffer-size-related as in the OrangeFS TCPBufferReceive parameter or scale-related as in the number of Lustre OSTs) that can be tuned to a particular architecture and workload. For example, Lustre has approximately 36 configurable parameters [12], while Ceph has roughly 320 configurable parameters [13]. In addition, there are a variety of storage device types available in HPC, including HDD, SSD, NVMe, PMEM, and more. The performance of an application workload will vary depending on which storage system, configuration, and resource set it is run on [14]. Therefore, storage systems will need to be carefully selected, configured, and deployed to reach optimal performance for a given application workload. This is a complex task, as each option exponentially increases the complexity of determining the optimal choice.

### C. Provisioning Storage Resources

Provisioning resources for storage systems and scheduling their deployment is typically performed manually by the users. Job schedulers have limited or no support for I/O subsystems. Job schedulers in HPC typically provision with a focus on compute and network resources, and there is comparatively little work focusing on storage provisioning [15]. Manual provisioning of storage can involve erroneous plans, such as over-provisioning resources to a job or greedy allocation.

### D. Motivation

Supercomputing sites such as Argonne tend to display a wide variety of workloads [5], some of which will see suboptimal performance on a single RSS, however optimized. This leads to a need for custom storage systems. Unlike RSS, which is typically configured and provisioned by the system admin, custom storage has to be selected, configured, and deployed by the domain scientist. Domain scientists lack the expertise required to configure and provision storage, and there is extreme complexity involved in doing so for each application. The result is that cluster resources get allocated inefficiently or greedily and storage systems underperform on those resources [16]. There is a need for a service to automate configuration and provisioning of optimal storage systems, which domain scientists can utilize as an alternative to manual methods.

In the current paradigm, ephemeral storage remains largely underutilized or poorly understood [17]. There are emergent ideas pertaining to achieving ephemeral storage systems on major supercomputers, such as the Rabbit architecture on the upcoming El Capitan supercomputer at Lawrence Livermore National Laboratory, which allows preconfigured ephemeral storage services to be deployed on Solid State Drives (SSDs) at the tops of the racks [18]. This is a paradigm shift from RSS. However, the problem of the user not being familiar with storage and how it maps to their particular application is not solved by this architecture. This represents a key usecase, as there is a need for a service to assist in the choice of ephemeral storage deployment.

## III. LuxIO

### A. LuxIO High-Level Architecture

LuxIO is an intelligent storage resource provisioning and auto-configuration service designed to efficiently improve I/O performance by matching application I/O behavior to an appropriately tuned custom-deployed storage. The source code for LuxIO can be found on GitHub [1]. A typical LuxIO interaction would involve the user accessing the global LuxIO service via a transparent wrapper over the job submission interface, sending along information about known or assumed I/O behavior via LuxIO-specific flags, with LuxIO ultimately ornamenting the user job submission with a storage deployment schema and passing it along to the job scheduler. LuxIO aims to achieve: (1) *detailed yet performant application I/O behavior modeling*, (2) *I/O-aware abstraction of storage system complexity*, (3) *constraint-based resource provisioning*, and (4) *a standardized set of semantic-rich user interactions*.

As a storage resource provisioning and auto-configuration service, LuxIO provides a comprehensive approach to managing custom ephemeral storage services. This approach is shown in Figure 1. LuxIO takes in input from various sources such as a *jobspec*, an I/O trace file, execution traces, source code, and/or the binary of the application. An application jobspec is a definition of the core scheduling requirements of a job (e.g., the path to the binary, the scale of the job, and the compute and network resources desired). As an example, LuxIO accepts the Flux definition of a jobspec from Flux RFC 14 [19]. In our prototype implementation, we mostly focused on the application jobspec and its respective Darshan I/O trace. LuxIO processes and aggregates the I/O characteristics which these sources provide into an *I/O Trait* structure (step 1). The I/O Traits are put through the I/O Identity Builder to create an I/O Identity (step 2) by deriving new features and/or removing unnecessary ones. The fitness (normalized euclidean distance relationship [20]) of the I/O Identity to the different I/O behavior classes is then measured by the I/O Fitness Calculator (step 3). The I/O Identity Builder and I/O Fitness Calculator enable LuxIO to achieve *detailed yet performant application I/O behavior modeling* via an efficient and complete understanding of the I/O behavior of an application and how it relates to known I/O behaviors. The Classification Mapper then performs a conversion from these fitnesses and the list of available SSLOs (step 4) to an SSLO (step 5), which represents the most desirable characteristics of storage to optimize performance for the application (a sort of storage class); this mapper is an indirect tuning mechanism which enables LuxIO to achieve *I/O-aware abstraction of storage system complexity* via a relationship of application I/O behavior classes to SSLOs using metrics such as coverage and satisfaction. These metrics represent how well an I/O behavior class or application is covered by a particular SSLO, respectively. The Deployment Constructor then uses the SSLO to

[1] https://github.com/scs-lab/luxio

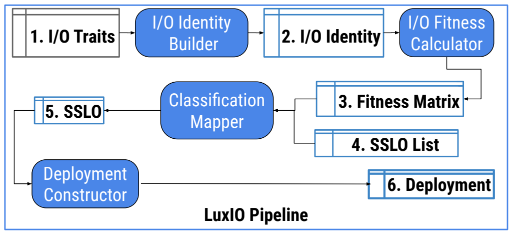


Fig. 1: LuxIO High-Level Architecture

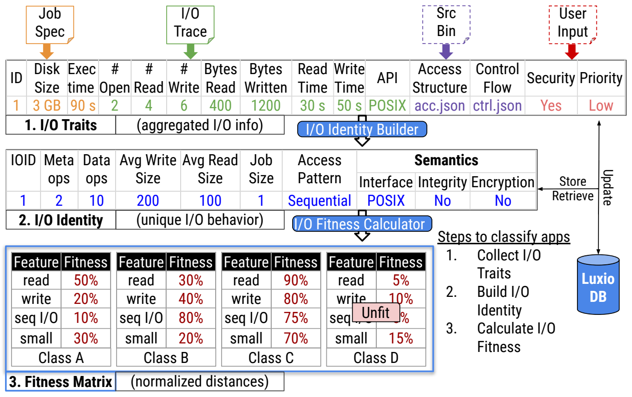

Job Spec
I/O Trace
Src Bin
User Input
ID | Disk Size | Exec time | # Open | # Read | # Write | Bytes Read | Bytes Written | Read Time | Write Time | API | Access Structure | Control Flow | Security | Priority
1 | 3 GB | 90 s | 2 | 4 | 6 | 400 | 1200 | 30 s | 50 s | POSIX | acc.json | ctrl.json | Yes | Low
1. I/O Traits
(aggregated I/O info)
I/O Identity Builder
IOID | Meta ops | Data ops | Avg Write Size | Avg Read Size | Job Size | Access Pattern | Semantics: Interface | Integrity | Encryption
1 | 2 | 10 | 200 | 100 | 1 | Sequential | POSIX | No | No
2. I/O Identity
(unique I/O behavior)
I/O Fitness Calculator
Store
Retrieve
Update
Luxio DB
Steps to classify apps
1. Collect I/O Traits
2. Build I/O Identity
3. Calculate I/O Fitness
Feature | Fitness
read 50% write 20% seq I/O 10% small 30% Class A
read 30% write 40% seq I/O 80% small 20% Class B
read 90% write 80% seq I/O 75% small 70% Class C
read 5% write 10% seq I/O % small 15% Class D
Unfit
3. Fitness Matrix
(normalized distances)

Fig. 2: Application I/O Behavior Modeling (1-3)

create a deployment schema (description of desired storage and how to configure it) for the job (step 6). The Deployment Constructor enables LuxIO to achieve *constraint-based resource provisioning*. It uses models to determine the optimal deployment based on one or more of the following four constraints: 1) resource utilization, 2) optimal performance, 3) interference, and 4) resource cost. Finally, a jobspec is ornamented with the deployment and passed to the job scheduler. LuxIO also aims to achieve a *standardized set of semantic-rich user interactions* via its data representations, such as the decorated jobspec containing the recommended storage deployment schema.

## B. *Application I/O Behavior Modeling*

LuxIO's Application I/O Behavior Model is shown in Figure 2. In step 1, LuxIO collects I/O Traits. In order to have an I/O behavior model which is as complete as possible, LuxIO takes inputs from multiple sources and aggregates them together into the I/O Traits structure. The different input sources include a jobspec, an I/O or execution trace, source code, an application binary, and user input. Of all the input sources possible, LuxIO only requires a jobspec and a Darshan trace. Darshan profiling is turned on by default on most supercomputers [21], so this is a realistic expectation. With more information, Luxio will be able to better characterize the I/O behavior of the application, but analyzing more information has greater complexity (either in time or in resolving conflicts). In step 2, the I/O Identity Builder creates an I/O Identity from the I/O Traits, deriving important features implied by the input. These derived features include common I/O metrics such as average write bandwidth (derived from total bytes written divided by time spent writing) and total metadata operations (sum of each metadata operation counter). In step 3, the I/O Fitness Calculator determines the relationship of the application to known I/O behavior classes using its I/O Identity. If fitness exceeds a configurable distance threshold, a new class will be created for that application in order to learn unknown behavior. The Classification Mapper will classify the application into one of the I/O behavior classes. Application classification is necessary for various reasons. First, classification allows LuxIO to speed up auto-configuration, because it can reuse the storage requirements (i.e., SSLOs) of similar applications rather than relearning them. Second, classification allows making well-informed storage resource sharing decisions via knowledge of how application classes interfere with each other. Third, having a measurement of similarity to reference applications can provide a behavioral approximation which is useful to the user in other contexts.

The methodology for building I/O behavior classes is as follows: First, a representative amount of profiling data should be collected from the site (enough to account for yearly seasonality, for example 20% of jobs in a year randomly-sampled). Second, a derived set of features that can be used to characterize I/O behavior is constructed from the profiling data (the same derived features as for I/O Identities in the previous paragraph). Third, an ensemble of models is built to predict I/O performance characteristics, such as read time, write time, and metadata time. The ensemble approach captures the diversity of I/O behavior because each model may perform better in certain scenarios. The ensemble model constructed is a stacked model that contains both linear (Lasso and Ridge) and tree-based (Ada Boost, Random Forest, XGB Regressor) models. Lasso [22] provides a simple model which works better with a larger feature set and low correlation, while Ridge [23] provides a more complex model which works better with a smaller feature set or high correlation. XGB [24], [25] is a high-performing model, but requires a large quantity of data to reach reasonable performance compared to other tree-based models; Ada Boost [26] is especially robust to overfitting for low-noise datasets; Random Forest [27] works well for noisy data, but may have overfitting problems which can cause issues when applied to data from general-purpose computing environments. The ensemble model utilizes performance-related features (e.g., number of I/O operations, average request size) rather than semantic features (e.g., interface used) to predict performance. XGB Regressor [25] is used as the meta-learner to combine the outputs from each of the different models in the ensemble, due to its low execution time and highly accurate results. Lastly, RayTune [28], a widely popular distributed hyperparameter tuner, is used to tune the hyperparameters of the model. It uses the AxSearch algorithm to determine values for hyperparameters such as the number of features, subsampling method, and maximum tree depth. Fourth, the feature set is reduced using Recursive Feature Elimination [29] to minimize noise introduced by low impact features and improve efficiency of search [30]. This process recursively removes features insignificant to prediction until a targeted number of features is reached [29]. The target number of features is determined by RayTune in the previous step. Fifth, I/O behavior classes are determined by first normalizing the different features and then applying a clustering algorithm. Normalization applies a logarithm on the feature value to reduce the effect of outlier data points, and a Min-Max Scaler to transform the data into the range [0,1]. K-Means clustering is applied to the transformed dataset and selected feature set. The optimal number of clusters is identified using a combination of the elbow method [31] and Davies-Bouldin index [32] - both widely used heuristics to assess the quality of unsupervised ML models.

**Case Study**: As an example of the LuxIO methodology for creating I/O behavior classes, three years of Darshan trace data of the Theta and Mira supercomputers, extracted from the Argonne Leadership Computing Facility (ALCF) public data portal [33], were used to

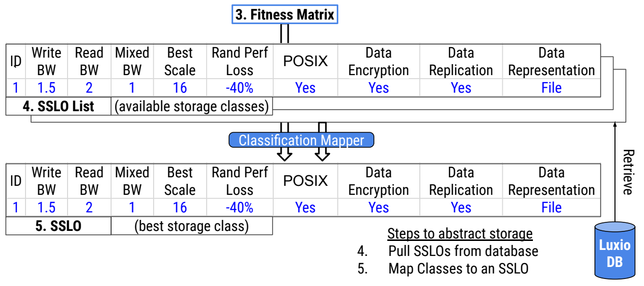
```
{
  "APP CLASS": {
    "application name": "VPIC",
    "io behavior class": "512-read-heavy-balanced-mpiio",
    "performance requirements": {
      "client procs": 512, // [256, 640]
      "read bandwidth": 0, // [0, 1]
      "write bandwidth": "1.31E+3 MB/s", //[1e+2, 1e+5]
      "read ops total": 0, // [0, 1e+1]
      "write ops total": 2.7e+3, // [1e+0, 1e+5]
      "metadata ops total": 2.86e+3, // [1e+2, 1e+4]
      "read ops per second": 0, // [1e+0, 1e+1]
      "write ops per second": "2.7e+1 ops", // [1e+0, 1e+2]
      "metadata ops per second": "7.0e+0 ops", // [1e+0, 1e+1]
      "bytes read": 3.23e+11, // [1e+9, 1e+12]
      "bytes written": 3.46e+8 // [1e+6, 1e+9]
    },
    "access pattern": {
      "small io ops": 1.4e+3, // [1e+2, 1e+5]
      "large io ops": 8e+2, // [1e+1, 1e+3]
      "seq io percent": 50 // [0, 1e+2]
    },
    "semantics": {
      "interface": "mpiio",
      "POSIX": true,
      "compression": false,
      "encryption": false
    }
  }
}
```

Fig. 3: Example I/O Identity, VPIC simulation

identify classes of I/O behavior pertaining to applications run by Argonne National Laboratory (ANL) between 2017 and 2019. The dataset includes a total of 542,029 entries. After removing jobs that spent less than 5% of their time performing I/O, the dataset was pruned down to 83,714 entries. In the resulting dataset the average job length was 1,573 seconds, the average number of processes was 49,132, the average number of I/O operations was 427,825,400, and the average number of bytes of data written was ~380 GB. Tuning the ensemble model and eliminating features reduced the original 100 features in the traces to 24. Examples of eliminated features include number of MPI-IO syncs, POSIX file alignment, and POSIX mode. On the other hand, features kept include the number of client processes, number of opens, total number of bytes read/written, number of read/write operations, the number of small I/O requests (less than 10 KB), percentage of I/O which is sequential, and others. The complete list of features is in the data section of the LuxIO GitHub repository.

**Model Validation**: In order to validate the accuracy of the predictions, the ensemble model was evaluated using the Argonne dataset, with 80% of the dataset for training and 20% for testing. We found that the stacked model had a better Mean Absolute Percentage Error (MAPE) than the individual component models. In the worst case, the model achieved an MAPE of 35% for the read time, 32% for the write time, and 27% for the metadata time (more detail in LuxIO GitHub repository). This resulted in a fast (under 100 ms), low-resource-overhead model that was able to predict I/O time measurements to within about 30% of the true value, which was considered sufficient for the purposes of LuxIO. This was deemed acceptable because the prediction will typically give an estimate with the same units (e.g., hours, minutes, seconds) as the actual I/O performance, even for highly variable workloads.

**Application Example**: Figure 3 shows an example I/O Identity for Vector Particle-In-Cell (VPIC). This I/O Identity demonstrates various features which LuxIO derives from I/O Traits. The names of the I/O Identities are also used for I/O behavior classes and follow the format "[job scale]-[most common operation type]-[metadata operation load]-[access pattern]-[interface]". Important categories of features include information about the scale of the application, the kind of I/O being performed, the different access patterns, and the size of the I/O operations performed. For example, LuxIO I/O Identities specify


3. Fitness Matrix
Classification Mapper
Retrieve
Steps to abstract storage
4. Pull SSLOs from database
5. Map Classes to an SSLO
Luxio DB

| ID | Write BW | Read BW | Mixed BW | Best Scale | Rand Perf Loss | POSIX | Data Encryption | Data Replication | Data Representation |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1.5 | 2 | 1 | 16 | -40% | Yes | Yes | Yes | File |
| **4. SSLO List** | (available storage classes) | | | | | | | | |

| ID | Write BW | Read BW | Mixed BW | Best Scale | Rand Perf Loss | POSIX | Data Encryption | Data Replication | Data Representation |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1.5 | 2 | 1 | 16 | -40% | Yes | Yes | Yes | File |
| **5. SSLO** | (best storage class) | | | | | | | | |

Fig. 4: Storage Service Objective Definition (4-5)

the access pattern of the application, which includes information about the number of large and small I/O operations and the percentage of sequential I/O in the application. This example of classification demonstrates the kind of detail that LuxIO aimed to achieve in objective (1).

## C. Abstracting Storage System Complexity

In order to abstract storage system complexity, LuxIO defines the SSLO abstraction. This abstraction generalizes the performance characteristics and semantics of storage deployments (e.g., read/write bandwidth, interfaces supported) under known I/O behaviors. Since multiple storage deployments possessing similar characteristics can fulfill the same SSLO, there can be different resource provisioning strategies based on the resources available in the system and user-desired heuristics. LuxIO selects an appropriate SSLO in steps 4 and 5 of Figure 4. In step 4, LuxIO consults a database for existing SSLOs, representing both currently-deployed storage services and new deployments. In step 5, the Classification Mapper selects an SSLO which is appropriate for the I/O behavior class of the application. This is done instead of unique configuration in order to make storage configuration performant.

The Classification Mapper takes as input the fitness matrix from the I/O Fitness Calculator and the list of SSLOs. It then maps the classes which represent the application I/O behavior to a single SSLO via an algorithm that utilizes fitness to a particular I/O behavior class and the coverage of that class by a particular SSLO. Coverage values are known relationships between I/O behavior classes and SSLOs. They are determined via heuristic associations between features of the I/O behavior classes and features of the SSLOs. For example, features relating to the sequentiality of accesses in the application will correspond to the sequential sensitivity feature in the SSLO. The mapping algorithm is presented as Algorithm 1. The I/O behavior class fitness is multiplied by the coverage to get the score along a path, and the satisfaction of a particular SSLO is then determined to be that of the maximum scoring path with that SSLO as a terminal point. This SSLO will be the output of the mapping step. In the event that the application is too far away from known SSLOs, the default SSLO will be applied in order to get feedback on application performance.

The methodology for determining the SSLOs has several steps. First, a comprehensive stress test of the utilized storage systems under various workloads and device types should be performed. To capture a wide variety of I/O workloads and their unique characteristics, LuxIO suggests a stress test based on CORAL [34], consisting of several I/O benchmarks and kernels (VPIC [35], HACC-IO [36], Enzo [37]) executed in a combined script. Tests should be run with various client scales, storage hardware, and storage configurations. Second, a Min-Max Scaler should be applied to put the resultant

**Algorithm 1** Mapping I/O Identities to SSLOs

```
 1: procedure MAPPER(IOID, IOClasses, SSLOs, thresh)
 2:     Let $S$ be an arr with len IOClasses.len * SSLOs.len
 3:     for IOClass $\in$ IOClasses do
 4:         for SSLO $\in$ SSLOs do
 5:             $f \leftarrow fitness$(IOID,IOClass)
 6:             $c \leftarrow coverage$(IOClass,SSLO)
 7:             $S$[IOClass,SSLO] $\leftarrow f * c$
 8:         end for
 9:     end for
10:     return highest-ranked (SSLO, IOClass) pair in $S$
11: end procedure
```

```
{
  "SSLO": {
    "name": "read-large-heavy-sequential-mpiio",
    "performance guarantees": {
      "write bandwidth": "2.362232e+3 MB/s", // [1, 1e+4]
      "read bandwidth": "4.187593e+3 MB/s", // [1e+2, 1e+4]
      "mixed bandwidth": "2.791729e+3 MB/s", // [1e+2, 1e+4]
      "io operation latency": "5e-06 sec", // [1e-7, 1e-5]
      "metadata operations per second": 5.18496e-1 // [1e-3, 1e+1]
    },
    "access pattern": {
      "random write performance sensitivity": 0.33, // [0.24, 0.45]
      "random read performance sensitivity": 0.36, // [0.124, 0.48]
      "small read request performance sensitivity": 0.7, // [0.6, 0.8]
      "small write request performance sensitivity": 0.6 // [0.5, 0.7]
    },
    "semantics": {
      "posix": true,
      "data encryption": false,
      "data replication": false,
      "data compression": false,
      "data representation": "file" // ["file", "nosql", "rdbms", "kvs"]
    }
  }
}
```

Fig. 5: An example of an SSLO, including storage data categories

stress test data in the range [0,1]. Finally, the data should be grouped using K-Means clustering to create the SSLOs.

In order to provide an example of the LuxIO SSLO construction methodology, we performed a comprehensive stress test of OrangeFS, because of its complex configuration space and ability to run in userspace. This stress test ran over several weeks on a production cluster (see section IV-A) across all available storage devices, and collected 16,887 data points of performance measurements such as I/O time, read/write bandwidth, and latency. Figure 5 provides an example SSLO generated from the OrangeFS usecase, which showcases the different aspects needed to define an SSLO. SSLOs are named according to the following format: "[performant operation type]-[performant operation size]-[metadata throughput]-[I/O pattern preference]-[interface supported]". SSLOs have three categories of data: a) minimum performance guarantees of the chosen storage deployment, b) performance variability of the storage system across various access patterns (i.e., storage sensitivity), and c) different non-performance-related semantics required by the application. For example, minimum performance guarantees includes read, write and mixed bandwidth requirements, I/O operation latency, and performance for metadata operations. Semantics are typically user-specified, and might include the security or logging configurations of OrangeFS, or could include replication parameters such as osd_pool_default_size in Ceph. The example SSLO provided shows that LuxIO has captured the kind of storage behavior necessary to achieve objective (2).

## *D. Constraint-Based Resource Provisioning*

After an SSLO is selected for a job, a deployment schema meeting the requirements of the SSLO must be constructed based on the constraints of availability of resources in the cluster and time and cost requirements of the user, as shown in Figure 6 step 6. Deployment construction involves selection of resources and a storage configuration. Resource selection identifies physical resources

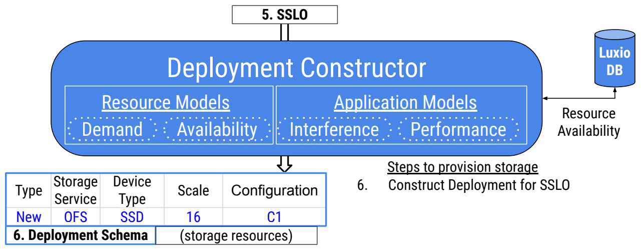


Fig. 6: LuxIO Deployment Construction

to use for a deployment with various user-chosen policies. These include maximizing resource utilization, maximizing performance, and minimizing interference with other applications. Where permitted by the user, the Deployment Constructor will attempt to share with an existing storage service, because using storage that another application has already deployed will eliminate time waiting in the job queue for the necessary resources and for the storage system to initialize.

**Resource and Configuration Selection**: To identify the resources and configuration which fit to a given SSLO, LuxIO uses a model which predicts performance characteristics based on the workload characteristics (e.g., access pattern, scale, latency and bandwidth requirements), storage system type (e.g., OrangeFS, Lustre), network type (e.g., ethernet, infiniband), device type (e.g., HDD, NVMe), and number of devices. The model is an XGB Regressor that is trained and tuned on the stress test data using Bayesian Optimization. The error of this model was evaluated using 80% of the data as a training set and 20% as a testing set, and it was found that the model gets an MAPE of 9.995%. Random Forest and Ada Boost models were also tested for this purpose, but XGB Regressor was selected because it performed the best of these three. LuxIO uses this model to identify the minimum number of storage devices required to achieve the required performance characteristics of the SSLO using nonlinear least squares minimization [38]. Using this approach, LuxIO constructs new candidate deployment schemas for every device and network type and loads the set of existing shareable deployments from the LuxIO database. LuxIO obtains information about the interference between I/O behavior classes and the application scale during the stress testing phase. It does this by running pairwise tests for each pair of I/O behavior class representative applications (i.e., centroid application in the class) at various scales. Each pairwise test consists of running each application in the pair, both on the same node and on isolated nodes. This data is then used to generate a table of interference factors [8] for each application pair, calculated as the slowdown experienced by each representative relative to its I/O performance when running exclusively. LuxIO is designed so that users that opt into storage resource sharing may experience significantly improved turnaround time, but take the risk of some application performance slowdown within their specified threshold of interference. This approach is important to address user-defined fast scheduling constraints.

LuxIO employs one of the following policies to decide on the final set of resources to provision: a) *Maximizing performance* selects the resource set which will yield the best performance for the application, regardless of scheduling delay, and disallows future resource sharing. b) *Minimizing interference* favors storage resource sharing and selects the existing deployment which will cause minimum interference with the application. If there are no satisfactory existing deployments available, defaults to maximizing performance policy with storage resource sharing allowed. c) *Maximize resource utilization* favors deployment construction and selects the resource set with the lowest overall price (i.e., resource cost), or defaults to the minimizing interference policy if no resources are available within a user-defined time window. These constraints enable the sort of provisioning LuxIO aimed at in objective (3).

**Pricing Model**: In order to determine cost of different resources for the purposes of achieving better system resource utilization, LuxIO employs various pricing models. These assign a price to the different tiers of storage resources (e.g., Non-Volatile Memory express (NVMe), SSD, Persistent MEMory (PMEM)) to determine the most suitable tiers a user is able to utilize based on their budget. The Availability Price Model is represented by the formula 1, where the price of each tier $t_cost$ is a constant tier cost provided by the system administrator, and device availability is accounted for with $t_count$.

The Demand Price Model is designed such that resource price has a direct relationship with popularity, and there are bounds on the minimum and maximum price. These two bounds, along with the rate of growth of the sigmoid function, are the three hyperparameters of the Demand Price Model which need to be configured so that the model can run. The Demand Price Model sets the adjusted price of the tier as in formula 2, which returns a number between a configurable minimum and maximum value. The input to the function is the demand on the tier, which is a value which increases when the tier was accessed within a time interval $t_i$ and decreases when the tier was not accessed within a time interval $t_j$. The Demand Price Model can determine the cost of a full deployment by summing the prices of the resources required for that deployment. This model accounts for popularity as well as availability when pricing resources, so a resource accessed more often will be more expensive. LuxIO suggests that future production machines should incentivize storage resource sharing (and therefore increase resource utilization) by giving users who opt into storage resource sharing a core-hour discount using the pricing models.

$$AvailPrice(t) = \frac{t_{cost}}{t_{count}} \tag{1}$$

$$DemandPrice(t) = AvailPrice(t) * sigmoid(dem(t)) \tag{2}$$

## E. Interacting with LuxIO

Current job scheduling methods are to specify which storage devices are required for the application. This is functional, but it is also low-level. The user must know their storage requirements in order for the job scheduler to understand them. There are no standards for a higher-level interface which would provide for specifying I/O requirements directly to the job scheduler or storage system. In order to properly provision and configure storage systems, LuxIO needed to create a standard for specifying I/O requirements in a fashion that could be conveyed to the job scheduler and storage systems. The specifications for data in the LuxIO service include I/O Traits, I/O Identities, SSLOs, and storage deployment schemas. These JSON specifications represent a standard methodology for detailing the requirements of I/O to the job scheduler or storage system. Taken together, these I/O requirement specifications serve to enable simple communication about the natures of I/O behavior and storage service provisioning as encapsulated by the I/O behavior and storage quality and semantic models. These I/O requirement specifications are formally detailed in the specification section of the LuxIO GitHub repository. It achieves the type of communication standard that LuxIO aimed at in objective (4).

General use of the LuxIO service takes two forms: non-interactive and interactive. In the non-interactive case, the user specifies their access to LuxIO via a wrapper around the Flux job scheduler, which automatically employs LuxIO in order to determine and deploy appropriate storage for the job. This wrapper includes flags for passing along the various arguments that LuxIO takes (e.g., "`flux jobspec.yaml --use-LuxIO --LuxIO-flags='-t trace.darshan'`"). The ornamented jobspec may rarely become infeasible at scheduling time, and if that happens, then the job will be put back into the LuxIO pipeline up to a configurable number of times before failing. The ornamentation on top of the user-submitted jobspec is a storage deployment schema. This schema can either recommend a new deployment or a shared deployment, and contains storage-deployment directives and scheduler data staging hooks [39]. Staging hooks effectively cause the scheduler to load specified stage-in files to the storage before scheduling the job. They are necessary to ensure that data is properly staged into and out of the service, as ephemeral storage does not equate to ephemeral data.

In the interactive case, the user employs the LuxIO Command Line Interface (CLI) tools to directly access various I/O requirements. The primary information provided by LuxIO is the same storage deployment schema as in the non-interactive case. Alternatively, LuxIO can provide: a) the I/O Identity of the application, b) insight into the similarity of the application to common reference applications, c) the SSLO which represents the storage requirements of the application, d) the suggested hardware resource set for the application, and e) a storage system configuration file which satisfies the SSLO. Each of these data specifications gets stored in the LuxIO database as LuxIO derives them for future reference.

## IV. Evaluations

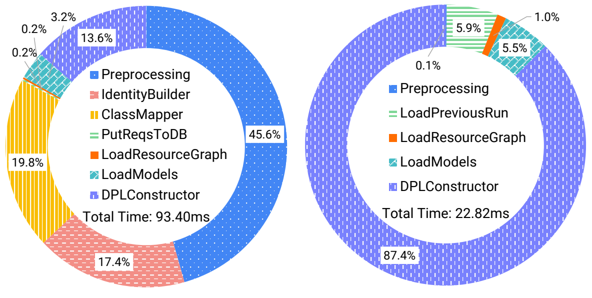
### A. Testbed

**Hardware**: Tests were performed on the multi-tiered Ares research cluster at Illinois Institute of Technology [40]. This cluster has 32 compute nodes and 32 storage nodes. Each compute node has a dual Intel® Xeon Scalable Silver 4114 processor and 96 GB of RAM, along with an NVMe PCIe $\times 8$ drive and SATA SSD. The storage nodes, meanwhile, each have a dual AMD Opteron 2384 @ 2.7 GHz processor with 32 GB of RAM, along with a SATA SSD and a traditional Hard-Disk Drive (HDD). For PMEM, we used a community-standard emulation technique [41].

**Software**: LuxIO is built on top of a few other technologies, consisting of Flux [42], Python 3, Darshan version 3.3.1, and msgpack version 1.0.2. LuxIO uses Redis version 6.2.6 for its database. We use Darshan as it is the most common application I/O profiler used in supercomputing facilities. We use Redis as it is a key value store that performs well for small size key-value data. Msgpack was chosen, as it is a well-known efficient binary serialization format. All the tests were run on top of OrangeFS version 2.9.8 as the PFS, with operating system caching disabled. They also use the Argonne and OrangeFS usecases for I/O classes and SSLOs.

### B. Overhead Analysis

Identifying the time cost of the internal components of LuxIO is important to understanding the impact of the LuxIO pipeline on job scheduling. To do this, we process a single, previously unknown jobspec using LuxIO twice and measure the execution time of the different internal steps each time. The results are shown in Figure 7. Overall, we see that a fully unknown job requires 93.40 ms to pass through LuxIO and additional executions of the job require only 22.82 ms. Two core observations can be made from this evaluation. First, these results show that LuxIO overhead is smaller than that of job scheduling and deployment, which can run in the 400 millisecond range [42]. This



(a) Provisioning a new job (b) Provisioning a known job

Fig. 7: Anatomy of operations for LuxIO

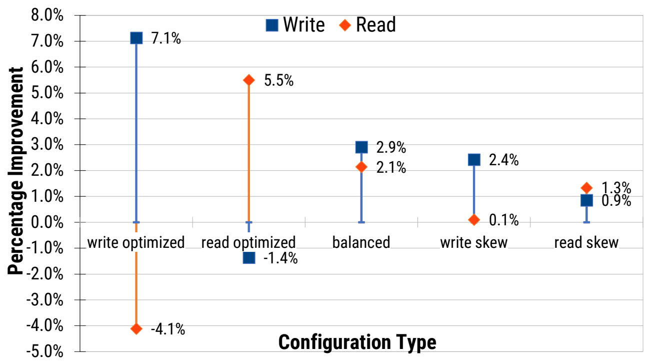


Fig. 8: Percentage variability in performance of diverse OrangeFS configurations over default for read and write workloads

means that LuxIO runtime is at worst 23.4% of that of existing job scheduling pipelines, but amortized over time can become as low as 5.7%. Second, LuxIO spends the majority of its runtime in the preprocessing step, which extracts the I/O traits of the application from the jobspec and input data. This step accounts for 45.6% of the LuxIO runtime and motivates LuxIO's usage of an external database to store the I/O information of previously seen applications; previously seen applications defer to LuxIO's secondary bottleneck of deployment construction, which accounts for 87.4% of the time to process a known jobspec.

## C. *Performance Variability Across Storage Configurations*

One of LuxIO's core premises is that the performance of an application is affected by how the underlying storage is configured. In order to quantify this variation, we evaluated read and write workloads on a few LuxIO-suggested OrangeFS configurations. We made sure that our applications ran exclusively during these evaluations to eliminate resource contention and pollution of performance results. These configurations and their percentage of improvement in performance over the configuration that ships as default OrangeFS can be seen in Figure 8. From this figure, we observe three things. First, the write-optimized configuration and read-optimized configuration are each best for their respective workload type and suffer performance loss if deployed for the opposite workload type. Correct configuration can lead to an improvement of 5-10% over the default, while incorrect configuration can lead to 5% loss in performance over the default. Second, the balanced configuration shows improvement for both read and write workloads, which shows an optimal choice for mixed workloads. Third, there are additional configurations which may be difficult to select manually, such as the write-skew and read-skew configurations (so named because they give

| App Name | # Procs | Amount Written | Amount Read | # I/O Ops | Request Size | Sequential | AppClass | Fitness | Coverage |
|---|---|---|---|---|---|---|---|---|---|
| VPIC | 640 | 3.2TB | 0 | Low | Large | Yes | A | 0.4126 | 0.6605 |
| BD-CATS | 640 | 0 | 3.2TB | Low | Large | Yes | B | 0.4638 | 0.5504 |
| HACC | 640 | 450GB | 450GB | Low | Large | Yes | C | 0.684 | 0.55 |
| ENZO | 640 | 40GB | 8MB | Medium | Small | Yes | D | 0.8221 | 0.6111 |

(a) Workloads Used for End-to-End

| Conf | TroveSyncMeta | TroveSyncData | TroveMaxConcurrentIO | TCPBufferReceive | ... |
|---|---|---|---|---|---|
| C1 | no | yes | 8 | 212992 | |
| C2 | no | yes | 32 | 212992 | |
| C3 | yes | no | 16 | 106496 | |
| C4 | yes | yes | 16 | 106496 | |

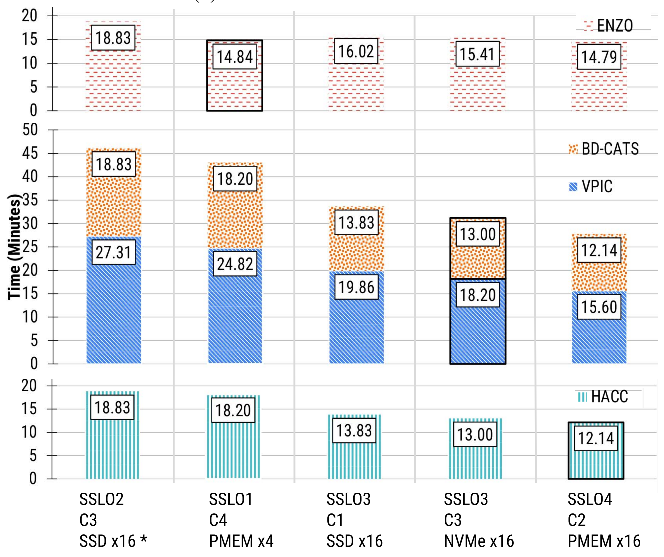
(b) Configurations Used for End-to-End

| Class | Interfaces | Scale | I/O Type | | | Access Pattern | | | Size | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | # Client Procs | Data Ops | | # Metadata Ops | # Small I/O Ops | # Large I/O Ops | % Seq I/O | # Bytes Read | # Bytes Written |
| | | | # Reads | # Writes | | | | | | |
| A | POSIX-IO/MPI-IO | 16K | 546K | 2M | 950K | 1M | 398K | 0.5 | 15G | 66G |
| B | POSIX-IO/MPI-IO | 256 | 723 | 11K | 16K | 1K | 1K | 0.98 | 472M | 2G |
| C | POSIX-IO/MPI-IO | 512 | 265M | 21M | 286M | 286M | 7K | 0 | 301G | 330M |
| D | POSIX-IO/MPI-IO | 32K | 306M | 869K | 306M | 306M | 1M | 0 | 60G | 42G |

(c) I/O Behavior Classes Used for End-to-End

| SSLOs | I/O Performance | | | Access Pattern | | Scale | Semantics | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Write BW | Read BW | Mixed BW | Randomness Perf Loss | Small Req Perf Loss | Best Client Scale | Posix | Data Encryp | Data Replica | Data Compress | Data Repr |
| SSLO1 | 1.8GBps | 2.9GBps | 0MBps | -0.26 | -0.99 | 8.00 | Y | N | N | N | file |
| SSLO2 | 2.2GBps | 3.1GBps | 2.1GBps | -0.12 | -0.95 | 4.00 | Y | N | N | N | file |
| SSLO3 | 2.7GBps | 3.9GBps | 2.6GBps | -0.11 | -0.90 | 8.00 | Y | N | N | N | file |
| SSLO4 | 3.1GBps | 4.3GBps | 3.6GBps | -0.03 | -0.90 | 4.00 | Y | N | N | N | file |

(d) SSLOs Used for End-to-End



(e) End-to-end evaluations of LuxIO service. Separated into ENZO, VPIC+BDCATS, and HACC workloads

Fig. 9: End-to-End evaluation of LuxIO service. Columns with black borders indicate LuxIO's chosen deployment, SSLO# indicates an SSLO, C# indicates a set of configuration details, and <device> ×# represents a certain number of devices of the specified type. The * indicates a baseline deployment.

slightly better performance for write and read workloads, respectively). LuxIO utilizes these in cases with workload types and ideas that were not considered in simple manual selection. The performance of LuxIO's classification is demonstrated in the end-to-end evaluation.

## D. *End-to-End Profiling of LuxIO Service*

LuxIO provisions resources depending on the I/O characteristics of the applications. These evaluations demonstrate the effectiveness of provisioning customized storage services through LuxIO. They focus

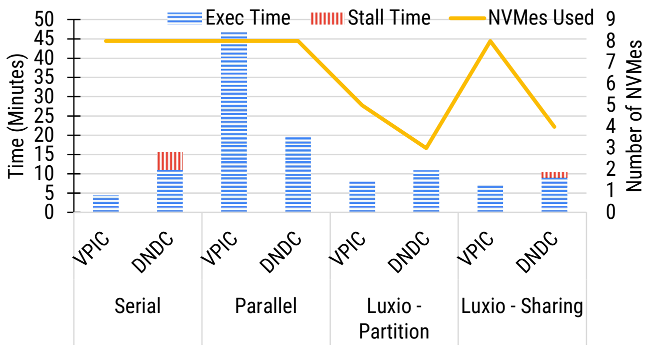


Fig. 10: Aggregated runtime of different resource sharing models. Serial means waiting to run on the same resources, Parallel means sharing resources, Partition means running with resource partitioning, and Sharing means running with LuxIO interference staggering and resource sharing policies.

on four real-world applications: VPIC, BD-CATS, HACC, and ENZO. These applications are detailed in Table 9a. VPIC [35] is a plasma modeling code, its I/O kernel consists of a large amount of data written with large request sizes, specifically 32 bytes per particle across 8 million particles per process in each collective write phase. BD-CATS [43] is a cosmological and plasma physics clustering analysis system with an I/O kernel which consists of a large amount of data being read with large request sizes; in our case, this data is being read from the VPIC program. Hardware Accelerated Cosmology Code (HACC) [36] is a cosmological simulation using N-body techniques to simulate fluid structure formation; its I/O kernel involves a gigabyte-order write phase and a read phase, each involving large request sizes of 20 million particles per process. ENZO [37] is an adaptive mesh refinement simulation code; its I/O kernel consists of gigabyte-order writes followed by megabyte-order reads, with a medium number of small request sizes. Enzo processes data on a $128 \times 128 \times 128$ grid which is checkpointed every 10 seconds. A summary of the details of the best-fitting I/O behavior classes of our test workloads is provided in Figure 9c. For example, it can be seen that HACC fits best to class C which has a 512 client process scale and leans towards reads but also has some writing mixed in. This class is not a perfect fit to HACC, but it fits well enough that HACC shows an improvement in performance when mapped to this class. The always-on PFS (i.e., RSS) on our cluster is represented by *SSLO2+C3+SSDx16* and consists of an OrangeFS running over 16 SATA SSDs using the default OrangeFS configuration. The relevant OrangeFS configurations are represented in figure 9b, with important distinguishing factors including TroveSyncMeta, TroveSyncData, TroveMaxConcurrentIO, and TCPBufferReceive. VPIC, BD-CATS, and HACC spend the majority of their runtime (90%) in I/O, and perform mainly large I/O accesses ($\geq 1$ MB). ENZO spends roughly 40% of its runtime in I/O, and performs mainly small I/O accesses ($<10$ KB). Each application is executed at a 640-process scale over an exclusive deployment of a PFS.

**LuxIO Satisfaction Evaluation**: In order to show that LuxIO is able to capture the variety in these applications despite them not being part of the training set, we also include the fitnesses and coverage of these applications in Figure 9a. ENZO achieves the best fitness of 82.2%, while the worst fitness is the 41.3% of VPIC. The satisfaction of these requirements ranges from 25.5% for BD-CATS to 50.4% for ENZO. We can see that the Argonne models can be applied across sites, although it is always possible to contrive a workload which does not fit well to any suggested model. As a general guideline, retraining of the I/O behavior models is expected to be needed every 10 years (to account for paradigm shifts in supercomputing) or for any site with a specialized purpose, and retraining of the SSLOs is expected to be needed whenever I/O behavior models are retrained or for any site not using OrangeFS.

**LuxIO Recommendation Evaluation**: Each application is passed through the LuxIO pipeline to identify its optimal SSLO, and we compare the runtime of these applications on the different SSLOs in order to show performance improvements. A summary of the relevant SSLOs for the end-to-end evaluation is contained in Figure 9d. From Figure 9e, we can make the observation that the deployments that LuxIO recommends perform significantly (20% - 50%) faster over the always-on PFS. This is because the SSLOs selected satisfy the requirements of the known characteristics of the applications, allowing for the construction of faster storage deployments for these applications. Second, it can be seen that different PFS configurations running over the same storage type and scale can lead to significantly different performance. For VPIC & BD-CATS, LuxIO recommends an SSLO that supports a high read/write bandwidth (SSLO3). The final deployment constructed from this SSLO consists of 16 NVMes and OrangeFS configuration 3 (i.e., C3). When compared to SSLO3+C1+SSDx16, which contains 16 SSDs and configuration C1, a performance improvement of 14% is made. Note that the performance improvement from a different set of resources and configuration is more than that from configuration alone (e.g., in evaluation section IV-C). Third, also focusing on VPIC & BD-CATS, it can be seen that another SSLO is available that could have resulted in 10% better performance (SSLO4). However, this SSLO constructs a deployment of 16 PMEM devices, which are reserved for applications with random access patterns and small I/O, rather than sequential access patterns with large I/O. While PMEM is faster than NVMe, applications with more favorable characteristics for PMEM should be prioritized. In this evaluation, LuxIO performs a multi-objective optimization, achieving the best performance while minimizing the scheduling budget cost of resources. As can be seen, LuxIO avoids suggesting PMEM devices, since it would violate the budget available for this job. It is notable that all of LuxIO's recommended deployments provide better performance than the default, so that even when a suboptimal deployment is chosen in order to improve resource utilization, the application still sees performance improvement. Lastly, ENZO's evaluation showcases the value of the cost model used in LuxIO. While both SSLO1 and SSLO4 provide similar performance, the SSLO1 deployment uses 4 PMEM devices while SSLO4 uses 16 PMEM devices. In this instance, LuxIO chooses the deployment with a smaller resource requirement, leading to higher overall resource utilization. Overall, we observed that LuxIO can reduce the execution time of application I/O by an average of 43% when compared to an always-on deployment of the same storage system.

## E. *Resource Sharing in LuxIO*

Figure 10 shows how LuxIO performs when multiple applications are run on the same resources. For this evaluation, applications are allowed to share compute nodes and perform I/O to the same NVMe drives. We also use a new application for this evaluation: DeNitrification-DeComposition (DNDC). DNDC is a simulation for carbon and nitrogen models which involves reading 64,000 small files, each of a few KBs [44]. This evaluation utilizes four methods of deployment: one where applications wait for each other before running on shared resources (serial), one where applications run simultaneously on shared resources but without LuxIO to determine the optimal resources for each (parallel), one where LuxIO is used to determine an optimal partition of resources between the applications but without sharing (partition), and a final method where applications run simultaneously on shared resources with LuxIO indicating which applications should run on which resources and staggering runtime to minimize interference (sharing). We can make several observations from this figure. First, in the serial case, turnaround time is longer compared to LuxIO provisioning strategies; although running a single application exclusively would be faster due to having exclusive use of the system resources, this would neglect to consider stall time from waiting for other applications. Second, in the parallel case, we see an increase in the total time to run applications due to resource contention. Finally, in the partition and shared cases, we see an improvement in overall performance and resource utilization. The sharing case could pair 2 DNDC instances to one VPIC and therefore require 25% less resources than the serial or parallel cases. The improvement in turnaround time is 5% over the serial for the LuxIO partition case, and 12.8% over the serial for the LuxIO sharing case. The advantage over parallel for these two cases is much more significant, at over 70%.

## V. Related Work

### A. Application I/O Characterization

Some works have proposed employing sophisticated Machine Learning (ML) techniques to alleviate the issues with static analysis in characterizing I/O. For example, the study by Bez et al [45] explains and evaluates multiple methodologies for the prediction of I/O access pattern, including decision trees, random forest, and neural networks. These predictors have the advantages over statistical analysis of analyzing while the application is running and not requiring user intervention. However, they can lack semantic detail, abstracting complex I/O behaviors as simply access patterns. LuxIO models are designed to provide additional details, such as average I/O bandwidth, total I/O size, and sequentiality.

### B. Configuring and Classifying Complex Storage

The objective of storage auto-configuration is to discover an optimally-performing configuration for a given application on given hardware. Auto-configuring storage for an application entails profiling the application on various storage configurations and selecting the configuration with the best performance. Configuration of storage can be done offline or online [46], where the offline case determines which parameters to use for configuring storage and the online case changes an existing deployed storage dynamically. Often, the parameter space for auto-configuration is very large, which necessitates dimensionality reduction [47] in order to reach feasible search times. Typical approaches make use of artificial intelligence techniques such as genetic algorithms [48] or a Latin Hypercube Design [49] in order to simplify the search, which would take too long using simple random search. The major flaw in existing auto-configuration approaches is that they tend to be prohibitively expensive even with the various techniques used to reduce the configuration search space, and complex due to the profiling either of one application across multiple storage types or multiple applications across one storage type. Ultimately, if optimality of configuration cannot be guaranteed in a satisfactory time period, then simply providing better performance than default should be prioritized [50]. This is a key justification for LuxIO, which prefers to use classification over the perfect optimality of auto-configuration.

A heuristic classification of storage is necessary for LuxIO's idea of using "good-enough" storage, rather than always configuring optimally. To the best of our knowledge, there are no comprehensive classifications of storage available before LuxIO. The closest existing works have come is to build loose heuristic classifications of storage properties, as in the survey by Macedo et al [51], or to characterize an individual deployment of a storage system, such as in the paper by Inacio et al [52].

### C. Automatic Storage Provisioning in HPC

Automatic provisioning in HPC is primarily seen in burst buffer scheduling. DataWarp [53] sets up several burst buffers and assigns applications to them based on remaining capacity and without concern for application interference. Harmonia [8] allows for various constraints in its provisioning, and ultimately can reduce interference where required. These systems are useful, but treat storage provisioning separately from the job and do not consider compute-node-local storage resources. While this works well in high-contention scenarios, and LuxIO techniques would be able to apply to this sort of situation, the usecase is ultimately different as LuxIO is aimed at improving utilization in a situation where node-local storage is ubiquitous. For LuxIO, resource utilization is the primary concern rather than resource contention.

## VI. Conclusions and Future Work

We have presented LuxIO, a service which provides intelligent storage resource provisioning and auto-configuration. LuxIO-deployed storage demonstrates up to $2\times$ improvement in performance over baseline deployments. Experimental and analytical overhead analyses show that LuxIO only imposes negligible overhead on the job scheduling pipeline. In addition, LuxIO resource sharing policies lead to more than 5% improvement in turnaround time, while improving resource utilization by approximately 25% in select workloads.

LuxIO currently uses a trace-based approach due to availability and easy-to-process nature of traces. This is limiting in cases where I/O traces are unavailable or outdated, or when application behavior varies significantly across runs. In order to overcome this, LuxIO hopes to explore supplemental information sources in future work, such as more detailed application binary and source code analysis techniques. I/O characterization of a job or an application can change over time. In future work, LuxIO hopes to explore utilizing malleable storage and techniques that capture the temporal sequence of operations in order to handle this dynamism, which will also benefit the I/O trace concern.

## VII. Acknowledgement

This research is supported by the National Science Foundation under Grants OCI-1835764, CSR-1814872, and CSSI-2104013. Argonne I/O trace data was generated from resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

## References

[1] Oracle, "Lustre* software release 2.x operations manual," Oracle, Tech. Rep., 2017. [Online]. Available: https://doc.lustre.org/lustre_manual.pdf

[2] A. Kougkas, H. Devarajan, J. Lofstead, and X.-H. Sun, "Labios: A distributed label-based i/o system," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019, pp. 13–24.

[3] X. Ji, in Wuxi, B. Yang, T. Zhang, X. Ma, X. Zhu, X. Wang, N. El-Sayed, J. Zhai, W. Liu, and W. Xue, *Automatic, Application-Aware I/O Forwarding Resource Allocation.* Usenix, 2019. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/ji

[4] O. Yildiz, D. Morozov, B. Nicolae, and T. Peterka, "Dynamic heterogeneous task specification and execution for in situ workflows," in *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*, 2021, pp. 25–32.

[5] Z. Liu, R. Lewis, R. Kettimuthu, K. Harms, P. Carns, N. Rao, I. Foster, and M. E. Papka, "Characterization and identification of hpc applications at leadership computing facility," in *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020, pp. 1–12.

[6] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Transactions on Storage (TOS)*, vol. 7, no. 3, pp. 1–26, 2011.

[7] H. Devarajan, A. Kougkas, P. Challa, and X.-H. Sun, "Vidya: Performing code-block io characterization for data access optimization," in *2018 IEEE 25th International Conference on High Performance Computing, Data, and Analytics*. IEEE, 2018.

[8] A. Kougkas, H. Devarajan, X.-H. Sun, and J. Lofstead, "Harmonia: An interference-aware dynamic i/o scheduler for shared non-volatile burst buffers," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 290–301.

[9] K. Shoga, "Monitoring and data integration at LLNL," Nov 2018. [Online]. Available: https://eehpcwg.llnl.gov/assets/sc18_workshop_thermo_shoga.pdf

[10] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 307–320.

[11] OrangeFS, "OrangeFS configuration file," 2020. [Online]. Available: http://docs.orangefs.com/configuration/admin_ofs_configuration_file/

[12] Oracle/Intel, "Lustre software release 2.x - operations manual - lustre_manual.pdf," 2022. [Online]. Available: https://doc.lustre.org/lustre_manual.pdf

[13] C. authors and contributors, "Configuration – ceph documentation," 2022. [Online]. Available: https://docs.ceph.com/en/latest/rados/configuration/

[14] F. Tessier, M. Martinasso, M. Chesi, M. Klein, and M. Gila, "Dynamic provisioning of storage resources: A case study with burst buffers," in *IPDPSW 2020-IEEE International Parallel and Distributed Processing Symposium Workshops*, 2020.

[15] M. R. Wyatt, S. Herbein, T. Gamblin, A. Moody, D. H. Ahn, and M. Taufer, "Prionn: Predicting runtime and io using neural networks," in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 1–12.

[16] U. Schwiegelshohn and R. Yahyapour, "Fairness in parallel job scheduling," *Journal of Scheduling*, vol. 3, no. 5, pp. 297–320, 2000.

[17] S. Gugnani, T. Li, and X. Lu, "Nvme-cr: A scalable ephemeral storage runtime for checkpoint/restart with nvme-over-fabrics," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 172–181.

[18] T. P. Morgan, "Livermore converges a slew of new ideas for exascale storage," 2021. [Online]. Available: https://www.nextplatform.com/2021/03/09/livermore-converges-a-slew-of-new-ideas-for-exascale-storage/

[19] F. Team, "14/canonical job specification – flux 0.13.0 documentation," 2021. [Online]. Available: https://flux-framework.readthedocs.io/projects/flux-rfc/en/latest/spec_14.html

[20] A. Singh, A. Yadav, and A. Rana, "K-means with three different distance metrics," *International Journal of Computer Applications*, vol. 67, no. 10, 2013.

[21] NERSC, "Darshan i/o profiler." [Online]. Available: https://docs.nersc.gov/tools/performance/darshan/

[22] J. Ranstam and J. Cook, "Lasso regression," *Journal of British Surgery*, vol. 105, no. 10, pp. 1348–1348, 2018.

[23] D. W. Marquardt and R. D. Snee, "Ridge regression in practice," *The American Statistician*, vol. 29, no. 1, pp. 3–20, 1975.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[25] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[26] D. P. Solomatine and D. L. Shrestha, "Adaboost. rt: a boosting algorithm for regression problems," in *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No. 04CH37541)*, vol. 2. IEEE, 2004, pp. 1163–1168.

[27] Y. Liu, Y. Wang, and J. Zhang, "New machine learning algorithm: Random forest," in *International Conference on Information Computing and Applications*. Springer, 2012, pp. 246–252.

[28] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A research platform for distributed model selection and training," *arXiv preprint arXiv:1807.05118*, 2018.

[29] scikit-learn developers, 2022. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html

[30] K. Yan and D. Zhang, "Feature selection and analysis on correlated gas sensor data with recursive feature elimination," *Sensors and Actuators B: Chemical*, vol. 212, pp. 353–363, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925400515001872

[31] M. Syakur, B. Khotimah, E. Rochman, and B. D. Satoto, "Integration k-means clustering method and elbow method for identification of the best customer profile cluster," in *IOP conference series: materials science and engineering*, vol. 336, no. 1. IOP Publishing, 2018, p. 012017.

[32] J. Xiao, J. Lu, and X. Li, "Davies bouldin index based hierarchical initialization k-means," *Intelligent Data Analysis*, vol. 21, no. 6, pp. 1327–1338, 2017.

[33] A. L. C. Facility, "ALCF public data," 2021. [Online]. Available: https://reports.alcf.anl.gov/data/index.html

[34] L. L. N. Laboratory, "Coral benchmark codes — advanced simulation and computing," 2021. [Online]. Available: https://asc.llnl.gov/coral-benchmarks

[35] K. Wu, S. Byna, B. Dong *et al.*, "Vpic io utilities," Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), Tech. Rep., 2018.

[36] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann, "Hacc: Extreme scaling and performance across diverse architectures," in *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013, pp. 1–10.

[37] G. L. Bryan, M. L. Norman, B. W. O'Shea, T. Abel, J. H. Wise, M. J. Turk, D. R. Reynolds, D. C. Collins, P. Wang, S. W. Skillman *et al.*, "Enzo: An adaptive mesh refinement code for astrophysics," *The Astrophysical Journal Supplement Series*, vol. 211, no. 2, p. 19, 2014.

[38] T. S. community, "scipy.optimize.least_squares – scipy v1.7.1 manual," 2021. [Online]. Available: docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.least_squares.html

[39] F. Team, "flux-sched/jobspec.yaml at master · flux-framework/flux-sched · github," 2022. [Online]. Available: https://github.com/flux-framework/flux-sched/blob/master/t/data/shell/node-local/jobspec.yaml

[40] S. Lab, "Resources — scs lab - illinois institute of technology," 2022. [Online]. Available: http://www.cs.iit.edu/ scs/resources.html

[41] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A cross media file system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 460–477.

[42] D. H. Ahn, N. Bass, A. Chu, J. Garlick, M. Grondona, S. Herbein, H. I. Ingólfsson, J. Koning, T. Patki, T. R. Scogland *et al.*, "Flux: Overcoming scheduling challenges for exascale workflows," *Future Generation Computer Systems*, 2020.

[43] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, P. Dubey *et al.*, "Bd-cats: big data clustering at trillion particle scale," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12.

[44] I. for the Study of Earth Oceans and S. U. of New Hampshire, "Home page," 2022. [Online]. Available: https://www.dndc.sr.unh.edu/

[45] J. L. Bez, F. Z. Boito, R. Nou, A. Miranda, T. Cortes, and P. O. Navaux, "Detecting i/o access patterns of hpc workloads at runtime," in *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2019, pp. 80–87.

[46] A. Mahgoub, P. Wood, A. Medoff, S. Mitra, F. Meyer, S. Chaterji, and S. Bagchi, "{SOPHIA}: Online reconfiguration of clustered nosql databases for time-varying workloads," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 223–240.

[47] H. Dou, P. Chen, and Z. Zheng, "Hdconfigor: Automatically tuning high dimensional configuration parameters for log search engines," *IEEE Access*, vol. 8, pp. 80 638–80 653, 2020.

[48] E. Zadok, A. Arora, Z. Cao, A. Chaganti, A. Chaudhary, and S. Mandal, "Parametric optimization of storage systems," in *7th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, 2015.

[49] P. Jamshidi and G. Casale, "An uncertainty-aware approach to optimal configuration of stream processing systems," in *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2016, pp. 39–48.

[50] A. Fekry, L. Carata, T. Pasquier, A. Rice, and A. Hopper, "To tune or not to tune? in search of optimal configurations for data analytics," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 2494–2504.

[51] R. Macedo, J. a. Paulo, J. Pereira, and A. Bessani, "A survey and classification of software-defined storage systems," *ACM Comput. Surv.*, vol. 53, no. 3, may 2020. [Online]. Available: https://doi.org/10.1145/3385896

[52] E. C. Inacio, J. Nonaka, K. Ono, M. A. Dantas, and F. Shoji, "Characterizing i/o and storage activity on the k computer for post-processing purposes," in *2018 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2018, pp. 00 730–00 735.

[53] D. Henseler, B. Landsteiner, D. Petesch, C. Wright, and N. J. Wright, "Architecture and design of cray datawarp," *Cray User Group CUG*, 2016.