A Comprehensive I/O Knowledge Cycle for Modular and Automated HPC Workload Analysis

Zhaobin Zhu*, Sarah Neuwirth*† and Thomas Lippert*†
*Institute of Computer Science, Goethe University Frankfurt, Germany
{zhu, s.neuwirth}@em.uni-frankfurt.de
†Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Germany

***Abstract*—On the way to the exascale era, millions of parallel processing elements are required. Accordingly, one major challenge is the ever-widening gap between computational power and underlying I/O systems. To bridge this gap, I/O resources must be used efficiently, thus a profound I/O knowledge is required. In this work, we analyze state-of-the-art approaches that can be applied to improve the general I/O understanding and performance. Based on our analysis, we present an automated, modular, tool-agnostic I/O analysis workflow and a prototype implementation that can be used to generate, extract, store, analyze, and use I/O knowledge in a structured and reproducible way.**

***Index Terms*—I/O, HPC, Workflow, Performance Analysis, Knowledge Sharing, I/O Understanding, I/O Optimization.**

I. Introduction

For the execution of scientific applications and workflows on high-performance computing (HPC) and modular supercomputing [1] environments, up to several millions of parallel processing elements can be involved. Consequently, applications can process and generate up to several petabytes of data. Typically, computing resources on HPC systems are distributed via resource managers such as Slurm. To request and allocate resources, HPC applications are submitted as batch jobs and are exclusively executed on the allocated resources. However, due to the ever-widening gap between compute power and I/O systems, and the shared nature of I/O subsystems, access to the I/O resources has become a major bottleneck [2]–[4] and the observed I/O performance at the application-level can be much lower than the theoretical peak bandwidth.

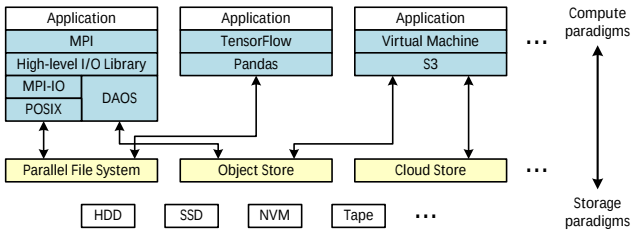

Fig. 1: Overview of the parallel I/O architecture.

As shown in Figure 1, the changing landscape of emerging hybrid HPC workloads has even further intensified the complexity of parallel I/O systems. There is a wide range of software and hardware components involved across different abstractions for performing parallel I/O. A typical I/O stack consists of high-level libraries, middleware, operating system functions, parallel file systems, and storage hardware [5]–[8]. Each of these layers offer corresponding configuration or optimization options [7]–[9]. To simplify data management and enable parallel I/O, scientific users often rely on high-level I/O libraries [10], [11] such as HDF5 [12], ADIOS [13], or PnetCDF [14]. Typically, these libraries support parallel access to the data and are built atop MPI-IO, where MPI-IO in turn uses POSIX to communicate with the underlying storage system [2]. Accordingly, such high-level I/O libraries also provide the ability to specify configurations for the lower layers and thus are used to improve I/O performance.

A recent study [15] has shown that HPC storage systems may no longer be dominated by write I/O. Emerging HPC workloads now also encompass advanced and big data analytics, machine learning, deep learning, and data-intensive workflows. Due to diverse I/O behavior of HPC applications and the intensifying complexity of the I/O stack, a crucial problem is the lack of I/O knowledge and the abundance of tunables that can be accessed via different APIs [4], [16], [17]. Since most users lack a deeper understanding of the underlying I/O subsystem [16], tunable options such as the parallel file system (PFS) striping settings, MPI-IO hints, and I/O library optimizations are oftentimes not used. Even though there is a vast amount of performance analysis, measurement, and visualization tools [18]–[20], users need to choose the best-suited one for their specific use case, which is essential for connecting a performance bottleneck with its tuning solution. Therefore, most users are forced to rely on the default I/O settings, which can result in wasting valuable I/O resources.

Given the missing ability to easily share insights with other users observing similar application or environment characteristics, I/O knowledge is often disregarded after a one-time use. To continuously grow the I/O knowledge base of the HPC community and to establish a standardized and tool-independent approach, we propose the following contributions:

1) A ***generic workflow*** that can be universally applied and modularly extended to improve the basic I/O understanding for single application runs and HPC system workloads. The workflow is software and hardware agnostic.
2) A ***prototype implementation*** that showcases the applicability of the proposed workflow. It enables scientific users to easily analyze and optimize the I/O behavior of their applications in an automated manner without any deeper understanding of the parallel I/O and storage system.

II. Analysis of Related Work

To address the aforementioned problems, a vast amount of European and international tools and frameworks already exist to collect and analyze the I/O behavior of applications. The analysis of existing work is crucial to design a generic workflow to drive the I/O knowledge cycle.

A. Performance Analysis and Visualization Tools

1) Scalable I/O for Extreme Performance (SIOX): *SIOX* [21] is a multipurpose environment for capturing system activities and gaining knowledge from the captured information, with a particular focus on I/O. To obtain an overview of all I/O calls on a HPC file system and to use them for I/O optimization, standardized interfaces are first created to collect performance data from all abstraction levels. The data is then compressed and stored permanently. Finally, collected performance data are analyzed and correlated with observed access patterns to gain knowledge about system characteristics and causal relationships.

2) DXT Explorer: Since Darshan [22] is one of the most widely used I/O profiling tool, a special tool was developed to narrow the gap between trace analysis and the actual applying of tuning parameters. *DXT Explorer* [3] is an interactive log analysis tool, which uses Darshan's extended tracing module (DXT) [23]. By visualizing the I/O behavior of the application, performance bottlenecks can be identified, and optimization parameters can be applied. Since the tool requires DXT log as input for the analysis, DTX Explorer is only available for Darshan and therefore is not compatible with other tracing and profiling tools, as well as output of established benchmarks.

3) SCTuner: To optimize both the I/O library and the underlying I/O stack at application runtime, *SCTuner* [4] was proposed as an auto tuner integrated into the I/O library itself. To profile the behavior of individual I/O subsystems with different configurations across I/O layers, a statistical benchmarking method was introduced. For this purpose, a group of IOR [24] benchmark experiments for specific I/O patterns with a set of tuning parameters like nodes, core per node, burst size, aggregators, and buffer size across I/O layers were first conducted. Subsequently, the benchmark results are normalized so that the used configuration can be mapped to a relative performance. For the optimization, an I/O pattern extractor was implemented in HDF5. When a file is opened for the first time, it extracts information about its I/O patterns as well as the number of compute nodes used, the number of MPI ranks, and the underlying configuration of the file system. In case of a parallel I/O call, information including burst size and the start offset of each burst as well as the total data size are extracted. Finally, when an I/O call is completed, the information is passed to the tuner. For the optimization at runtime, Tang et al. plan to use techniques such as online gradient descent to implement an online performance tuner.

4) H5Tuner: Due to the optimization possibilities on parallel file systems and I/O middleware, while hiding the complexity of the I/O stack from the developers, Behzad et al. proposed an autotuning system for optimizing I/O performance, I/O performance modeling, I/O tuning, and I/O patterns [17]. The framework *H5Tuner* is able to dynamical set the parameters of different levels of the I/O stack through HDF5 initialization function. For applying the configuration provided by H5Tuner to the autotuning system, the authors utilize tracing tools to extract the application's I/O kernel and execute the kernel with a preselected training set of tunable parameters.

B. User-Centric System Fault Identification using IO500

Since I/O performance is difficult to predict unless the entire resources are exclusively assigned to a single user, an IO500-based workflow has been proposed by Liem et al. [25]. The IO500 benchmark [26] is nowadays an established I/O benchmark and used to identify performance boundaries for optimized and suboptimal applications. The benchmark consists of data and metadata benchmarks [27]. This approach is based on the idea of exploring the IO500 benchmark to provide users realistic expectations regarding their application's I/O and optimization strategy based on the system's performance. To formulate a two-dimensional bounding box for user expectation, four benchmark scenarios from IO500, i.e., *IOR-hard*, *IOR-esay*, *mdtest-hard* and *mdtest-easy* are used. Afterward, the application's I/O performance is mapped into the bounding box and tuned if necessary during the last step in a certain dimension. In addition to the performance assessment through the visual representation of the bounding box, anomalies can also be detected through this approach.

C. Discussion

In terms of I/O optimization, profound knowledge is required to make the right decision in the optimization process. Given the lack of I/O understanding among the scientific community, these tools support the user in different ways. Each of them has strengths and weaknesses and is therefore only suitable for certain use cases. Due to the different aims and implementations, the approaches are not compatible with each other and do not offer the ability to share the obtained knowledge, i.e., the obtained knowledge is discarded after the one-time use for a specific use case. Given the importance of knowledge sharing in the HPC community, especially with the raise of emerging workloads, supporting the exchange of the I/O knowledge obtained and thus helping users and system engineers is the main focus of this work.

III. The I/O Knowledge Cycle Workflow

Whereas the applied approaches and goals of the previously discussed work differ greatly from one another, certain phases can be identified in all approaches. In general, at the beginning of the workflow, information is generated in different ways, i.e., a knowledge generation phase takes place. While in [17], [23], and [21] the knowledge is generated based on traces, [25] and [4] use benchmarks for the knowledge generation, i.e., different approaches use different data sources. Given the different foci of the analyzed approaches, a phase is differentiated by where the relevant knowledge is extracted

and persisted in different ways, e.g., as a simple log file in the system or as structured data in a specific database.

Before the actual knowledge can be used, it must be understood, i.e., analyzed. Depending upon the approach, more or less support is provided for the analysis. In SIOX [21], the knowledge is provided over a reporter and the GUI to the database, whereas DXT [23] has a more mature GUI, where the collected information can be visualized. Similarly, [25] visualizes the knowledge through a simple web GUI. In contrast, [4] and [17] offer less support for the analysis and the collected knowledge needs to be visualized manually first.

Finally, all approaches apply the obtained knowledge in a number of ways. While the last phase in [17] and [4] optimizes the I/O performance by tuning HDF5 parameters, the focus in [23], [21] and [25] is to create a better I/O understanding and thus improve the I/O performance of an application.

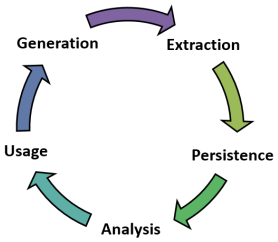

Fig. 2: Overview of the proposed I/O knowledge cycle.

From this, it can be deduced that basic knowledge must be established and analyzed first in order to further apply it, hence creating an iterative cyclic process. We call this iterative approach ***I/O knowledge cycle***. Since each phase is interdependent and has an enormous impact on the quality of the results in the subsequent phase, it is particularly important to consider the use of I/O knowledge as a *generic workflow*. To keep our workflow as generic as possible and to support both existing and new approaches, our workflow can be divided into five generic phases, as depicted in Figure 2:

1) ***Generation Phase***: To build a knowledge base, the I/O information needs to be collected first in a structured way, for example via benchmarks or simulations, but also via monitoring tools. However, the collected data often contains too much information, it is often generated for a single use and then discarded after the usage for performance tuning or anomaly detection. Since the generation phase is crucial for the subsequent phases, the generation needs to be carried out in a verified environment so that the knowledge is reproducible and representative.
2) ***Extraction Phase***: To transform the collected information into knowledge, metrics of interest are systematically extracted from the corresponding output or log file, according to the methodologies applied for the knowledge generation. Since various methodologies are allowed, various metrics can be extracted from different sources. At the end of this phase, the captured knowledge can be mapped to a data structure or a model depending on its further use. Regarding machine learning based I/O optimization and I/O predication, promising training models and predication models can be established from the collected knowledge. In terms of use cases demonstrated in this work, the obtained knowledge, i.e., performance metrics and system information are mapped to a Python object called *Knowledge*.
3) ***Persistence Phase***: To be able to apply the obtained knowledge and share it with the HPC community, corresponding knowledge is stored as structured data sets in this phase. For this, obtained knowledge can be saved, e.g., as a CSV file or as a database entry. Depending on the methodology, a mapping between the knowledge and database scheme is also required in this phase.
4) ***Analysis Phase***: In the fourth phase, the gained knowledge is further evaluated and analyzed. Typically, in order to better understand the impact of different I/O performance factors (see Figure 3), metrics of interest are presented in a well-organized way or are visualized as simple plots, interactive charts, or even complex dashboards. Therefore, in the analysis phase, simple use cases such as anomaly detection can be derived.
5) ***Usage Phase***: In the last phase, the gained knowledge is used for its actual purpose or for generating new knowledge. The I/O knowledge can be used for different use cases such as I/O optimization, performance prediction, anomaly detection, and I/O pattern analysis, but also for workload generation. Thus, this iterative cyclic process is either re-relaunched or terminated. Depending on the implementation and the use case, our analysis workflow can be used in both online and offline fashion.

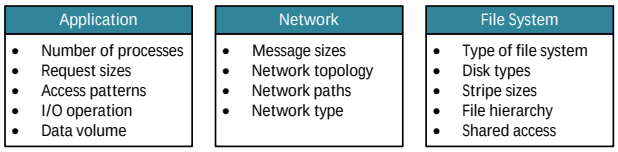

Fig. 3: I/O performance impact factors.

IV. High-Level System Design

Since different tools and technologies are applied in different phases, a *modular architecture* is chosen to implement the knowledge cycle as a generic workflow framework. Figure 4 illustrates the high-level architecture. While the generation and extraction of the knowledge usually is performed on HPC and supercomputer environments, the analysis of the knowledge can be done by a web-based tool on a local computer. Hence, independent phases can be isolated from each other, which enables a flexible and modular architecture. Also, the storing of knowledge can be done in two different environments. The data can be persisted either in (1) a local database or (2) globally by using a public database. Accordingly, the separation of databases gives us the flexibility to allow our tools to be applied in both public and private or combined environments. Therefore, the user can decide whether or not to share the knowledge and which knowledge to share.

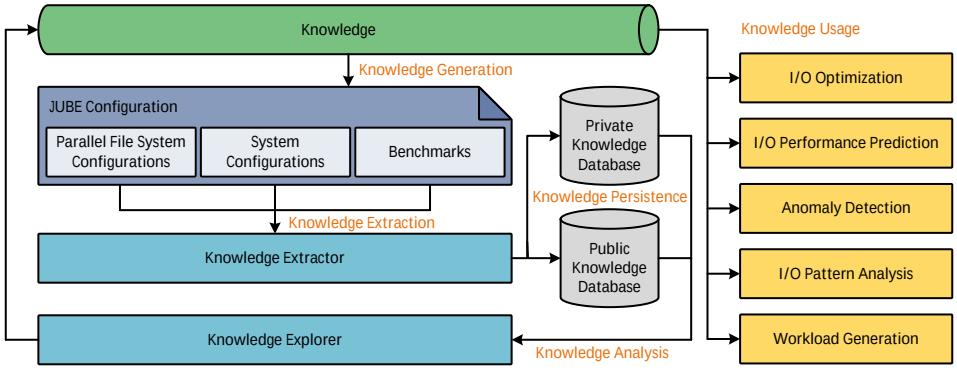

Fig. 4: A modular high-level architecture for various knowledge use cases.

Knowledge can be used and applied in different ways, as indicated in Figure 4. Consistent with our highly modular architecture, further modules such as the optimization module can be integrated in the future with minimal effort. In the following, five example use cases are briefly discussed:

I/O optimization: Given the complexity of the parallel I/O stack and the lack of optimization knowledge, automated tools can help the user to exploit I/O resources more efficiently. To achieve near-optimal use of I/O and storage resources, the I/O knowledge collected in our workflow can be applied in an offline fashion as well as an online fashion for I/O optimization by using an I/O pattern extractor. For example, in the offline mode, the users can be suggested with suitable configurations via a recommendation module, which can be applied manually for individual runs. The offline optimization process can be further automated through additional modules, including high-level libraries such as presented in [4], [12].

I/O performance predication: Since performance estimation requires a lot of expertise, machine and deep learning approaches can be applied to predict the I/O performance. Typically, the actual performance often differs from the theoretical performance and its accuracy heavily depends on the training data sets. Using our generic workflow, representative and reproducible data sets can be created for predictive modeling and then used to predict I/O performance. Furthermore, by integrating approaches such as [25] in our workflow, upper and lower performance boundaries can be determined and thus provide the user with a realistic expectation.

Anomaly detection: To ensure the reliability and reproducibility of the system, anomalies must be detected. While anomalies can be caused by several aspects such as workload distribution, application errors, hardware failures, and incorrect system configuration, the majority of them affect the overall performance. As the visualization of the I/O knowledge and performance bounding box are central parts of our workflow, anomalies can be detected in a simple and straightforward way. To identify possible causes, our workflow offers the ability to extract additional information such as file system information, and overall system statistics and configuration. It is planned to collect further information from workload managers such as Slurm, thus providing context between anomaly and causes.

I/O pattern analysis: A deep understanding of the I/O pattern helps to better exploit resources as well as improve the requirements for HPC storage resources as outlined by Layton [28]. Since our generic workflow is tool agnostic, it allows the integration of user-specific monitoring, profiling, and characterization tools. Thus, I/O patterns can be clearly identified and the performance impact can be better understood. For the validation of the correlation between I/O patterns and performances, benchmarks can provide further insights.

Workload generation: Considering the iterative large-scale I/O performance evaluation process [2], the workload, specifically the generation of the workload, plays a crucial role and therefore strongly influences the evaluation results. For this purpose, the knowledge obtained from our generic workflow can be used to, e.g., generate new benchmark configurations, but also synthetic workload for simulation and thus drive the simulation or initialize new evaluation processes.

V. Early Prototype Implementation

In the following, we present an early prototype implementation of the high-level architecture presented in Figure 4.

A. Phase I: Knowledge Generation

For the knowledge generation, different experiments and application runs can be performed on a specific target system. To emulate the I/O patterns of scientific applications, we use different community benchmarks such as IOR, which provides a flexible way of measuring I/O performance with different configuration options. IOR provides access to shared files both independently and collectively, and supports the use of various I/O interfaces, such as POSIX, MPI-IO, and HDF5 [29].

Furthermore, to ensure reliability and reproducibility of systematic I/O benchmarking, we use JUBE [30] to initialize our workflow. JUBE is a generic, lightweight, configurable benchmarking environment that supports systematic, automated execution, monitoring and analysis of application execution. Accordingly, we define a set of I/O patterns as JUBE parameters in the JUBE configuration file and use them for the execution of IOR in the respective iteration. By using the JUBE configuration, a set of benchmarks is executed for a given I/O pattern. JUBE creates a subdirectory for each benchmark iteration and stores the corresponding output.

With the increasing importance and acceptance of the IO500 benchmark and its applicability, e.g., in [25], the IO500 benchmark has also been integrated with eleven additional test cases as a separate knowledge generator and to demonstrate the easy expandability of our workflow.

To cover real I/O patterns like checkpoint and restart for large simulations, our prototype implementation additionally integrates HACC-IO [31], which supports different I/O interfaces (i.e., POSIX, MPIIO) and file access modes (i.e., single-shared-file, file-per-process, and one-file-per-group).

Finally, in order to demonstrate the modularity and expandability of our proposed workflow, we support Darshan as an additional data source. I/O characteristics can either be recorded for a particular scientific application or system-wide with Darshan. This process can also be automated via the JUBE environment. In terms of the bursty I/O workloads of modern HPC applications, Darshan and corresponding application can be used in the generation phase.

B. Phase II: Knowledge Extraction

To extract the previously generated knowledge from the output of the benchmark runs, we implement a Python-based tool called *knowledge extractor*. It can be run manually or automatically, i.e., executed in a workflow together with the knowledge generation phase. For automated execution, the knowledge extractor is defined in the JUBE configuration file as an application to be executed in addition to the benchmarks such as IOR and IO500. After the knowledge generation phase is completed, the extractor is started sequentially. By default, the tool expects the path of the output as a parameter. If the path is not specified, our tool automatically searches in the JUBE workspace for available benchmark results.

Essentially, the tool extracts different benchmark statistics and transforms the metrics of interest into a knowledge object. Our *knowledge object* currently consists of the parameters used, i.e., parameters describing the I/O pattern and the obtained benchmark results. IOR, for example, offers the ability to define the number of iterations of read and write operations, providing individual read and write operations. Therefore, the summary over the defined number of iterations is included in the results for a knowledge object.

In addition, the Python-based extractor can identify the used parallel file system settings and the system settings at runtime. For example, for BeeGFS, the file system settings *Entry type*, *EntryID*, *Metadata node*, *Stripe pattern* details can be collected. The support of other popular parallel file systems is planned for future releases. For the system statistics including processor cores, processor architecture, processor frequency, but also the cache and memory sizes, the extractor uses the data from `/proc/`. These statistics are also included in the knowledge object. Since we want to fully integrate Darshan support in the future, Darshan logs must be extracted during the knowledge extraction phase accordingly. To enable this, PyDarshan [32] is also integrated in the extractor and can interpret Darshan log as well.

C. Phase III: Knowledge Persistence

For persisting our knowledge object, we are using the DB-API 2.0 interface for SQLite [33] databases to perform database specific operations. Regarding the data security, our tool provides the ability to store the obtained knowledge either directly as a local SQLite database or by specifying a SQL connection URL remotely.

To persist knowledge objects, we currently use four database tables: *performances*, *summaries*, *results*, and *filesystems*. In the *performances* table, the I/O patterns, including benchmark-specific configurations, e.g., the IOR configuration such as *API*, *testFileName*, *filePerProc*, *start/end time*, are stored and each knowledge object is identified by a unique ID. For each knowledge an entry in the table *summaries* exist, i.e., a summary is uniquely assigned to a knowledge object by the foreign key *performance_id*. Since IOR supports different I/O interfaces, a summary is stored for each operation with the corresponding interface over the specified iteration. Each summary contains performance statistics such as max/mean/min bandwidth and the number of operations. In order to provide a rich set of visualization options, we have decided to store individual results, instead of storing only the summary of the performance statistics. The relationship between results and summaries is established by the foreign key *summaries_id*. At least one detailed result exists for each summary. In addition, depending on the file system used, a knowledge object can be extended by available user-level file system information such as *chunk size*, *number of storage target*, *RAID scheme*, *storage pool*. The information is then stored as an entry in the *filesystems* table.

As various benchmarks are used in IO500 for different test cases, e.g., mdtest, IOR, and find, we decide to first separate our knowledge object from the knowledge object used in IO500. Thus, IO500 related results are stored in separate tables such as, *IOFHsScores*, *IOFHsTestcases*, *IOFHsOptions*, *IOFHsResults*, and *IOFHsRuns*. While for each IO500 run an entry *IOFHsRuns* table and *IOFHScorces* table is created, the number of performed test case may vary depending on the setting. Accordingly, *IOFH_id* is applied as foreign keys for mapping to individual IO500 runs. In addition to the score, for each test case applied, options and the corresponding result are stored in *IOFHsOptions* table and *IOFHsResults* table. Also, system information extracted in the previous phase belongs to IO500 knowledge object and must be persisted accordingly. To assign the single system information to an IO500 knowledge object, the *IOFH_Id* is used.

D. Phase IV: Knowledge Analysis

To analyze the knowledge gained in previous phases, our implementation includes the *knowledge explorer*, a web-based analysis tool. Users can either use global data, i.e., knowledge stored in our database, or local data, i.e., manually uploaded knowledge objects. The knowledge explorer can be used to analyze and visualize performance statistics of an individual benchmark or application run or aggregated over multiple iterations and runs. To analyze a single run, our tool offers

the knowledge viewer feature. By selecting the command used for the benchmark, all related benchmarks and file system information, as well as the corresponding benchmark summary are displayed immediately. Because a benchmark run typically includes several iterations, our knowledge explorer offers the ability to display detailed performance statistics for each operation and iteration. For both single or multiple benchmark runs, the tool provides the ability to visualize results as an interactive graph and export it as an image file.

An essential feature is the comparison between different knowledge objects. To use the comparison feature, our tool offers the ability to select any number of knowledge objects and compares them based on defined metrics. Therefore, the user can select the axes of the chart at runtime to perform the analysis over various aspects. For the y-axis applied option and for x-axis focused metrics can be selected. Also, when selecting a knowledge object, an overview chart is automatically created at the same time, where the individual knowledge object are displayed on the basis of their throughput with corresponding min, max, mean as a boxplot.

For IO500, we provide an extra viewer in our knowledge explorer. This is similar to the viewer we have designed for IOR knowledge object, with the differences that it can additionally visualize score value and different test cases for each IO500 execution. Since our tool is currently under construction, we are already using a different library for the visualization for the IO500 viewer. To find similar knowledge object and perform fine-grained evaluations, we also support filtering and sorting of knowledge object in the comparison view. Screenshots of the previously mentioned features can be found on our companion repository as well.

E. Phase V: Knowledge Usage

As can be seen in Figure 2, in the last phase of our generic workflow, the obtained knowledge can be applied to different use cases, including I/O optimization, performance predication, anomaly detection, I/O pattern analysis, workload generation, and even the generation of new knowledge.

To demonstrate the applicability, we test our prototype implementation on the FUCHS-CSC cluster at Goethe University. This cluster provides a total of 198 nodes, each with 2x Intel Xeon E5-2670 v2 and 128 GB RAM. Accordingly, 20 cores per node and a total of 3960 cores can be allocated. In addition, FUCHS-CSC supports BeeGFS as parallel file system and through InfiniBand (FDR) offers an aggregated bandwidth of 27 GB/s [34]. In this work, we are focusing on two use cases.

1) Example I – New Knowledge Generation: As depicted in Figure 4, knowledge can be used to generate further knowledge. Hence we performed a series of benchmark experiments using IOR as explained in Section V-A. Up to 4 nodes, i.e., 80 cores, are allocated and BeeGFS is used for the execution. For this use case, the command `ior -a mpiio -b 4m -t 2m -s 40 -F -C -e -i 6 -o /scratch/fuchs/zhuz/test80 -k` was run.

Since read or write are not explicitly specified, IOR executes the command once with read and once with write per iteration.

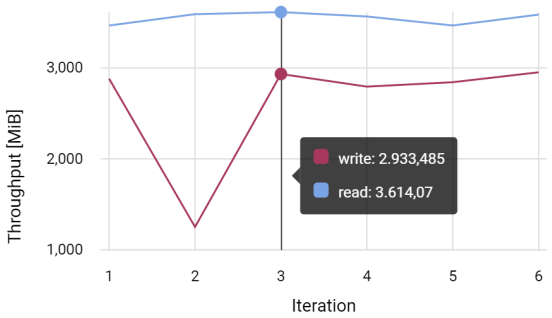

Fig. 5: Performance analysis through multiple iterations.

MPI-IO is used for parallel I/O and each task reads and writes 40 times 4MB contiguous bytes, with 2MB as *transfersize* and enabled flags *file-per-process*, *reorderTasksConstant* and *fsync*. To see the variation between each iteration, the iteration count is set to 6 for this experiment. After the results are collected and the relevant knowledge is stored in the database, new knowledge can be created.

For the generation of new knowledge, our web-based tool provides the functionality to generate new benchmark setups based on existing knowledge and can be extended to generate JUBE configuration additionally. The user can apply the generated command to re-run the workflow. First, the previously applied command is selected and then loaded from the corresponding configuration in the view and can be modified as required. Afterward, the new command can be created by clicking ”create configuration”. With the just created configuration, a new benchmark run can be started on the corresponding system and thus new knowledge can be generated. Due to the generic workflow, this process can be repeated as often as required.

2) Example II – Anomaly Detection: The I/O knowledge cycle can also be used to detect I/O anomalies. In this use case, I/O performance anomalies can be identified via the knowledge explorer in several ways.

As previously described, our tool supports the user through the visualization and comparison function in order to improve the understanding of I/O performance. In our previous experiment in Section V-E1, performance variation between the individual iterations of the benchmark run can be identified. As can be seen in Figure 5, the throughput in MiB and the number of ops for reads and writes over 6 iterations are visualized as an interactive chart. While the average throughput for write for iteration 1, 3, 4, 5, 6 is 2850 MiB, the throughput for iteration 2 is 1251 MiB, which is less than half the average throughput. Similarly, this phenomenon is evident when looking at the number of operations. Furthermore, other metrics like *closeTime*, *latency*, *totalTime*, *wrRdTime* can be displayed to support this observation and thus measurement errors can be excluded. Therefore, through our visualization, anomalies can be quickly and clearly identified.

A further ability to detect anomalies is the use of the bounding box [25] approach. Our tool supports the extraction of IO500 knowledge, thus the ability to create a bounding box

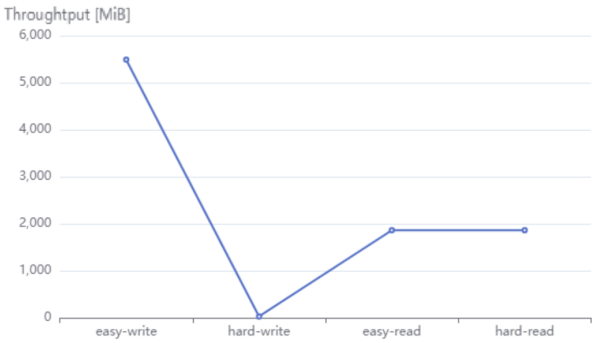

Fig. 6: Anomaly detection through IO500 boundary testcases.

to estimate the realistic I/O performance for a given system. To demonstrate this functionality, we use the benchmark setup previously described in Section V-E1.

The IO500 benchmark is run with 40 cores on FUCHS-CSC. The results are used to create a knowledge object and stored in our database for further analysis by the knowledge explorer. A bounding box can be created by applying certain test cases regarding the I/O performance of FUCHS-CSC. Since we currently do not support the visualization of the bounding box, we present the proposed approach of Liem et al. in a simplified way using an existing visualization example for IO500. Whereas in the original idea for the bounding box, medtest (easy, hard) and ior (easy, hard) are applied, we only utilize the *ior-easy* and *ior-hard* as a one dimensional bounding box for demonstration purposes. Therefore, as in Figure 6 illustrated, while the variance for *ior-easy* write and *ior-hard* write is quite large, the throughput for *ior-easy* read and *ior-hard* read remains the same. A possible cause for the bad *ior-easy* read result could be a broken node, which needs to be analyzed in more detail in the future.

VI. Outlook

Since our proposed analysis workflow consists of five phases, the workflow and corresponding tools can be individually improved and extended in each of these phases. In general, further approaches and tools can be considered in the future, e.g., for knowledge generation. Consequently, our extractor needs to be extended with appropriate interfaces. With regard to the more converging HPC community from different countries, it is also meaningful to integrate further parallel file systems such as Lustre [35], IBM Spectrum Scale [36], and OrangeFS [37] for our extractor and thus to enable a deep insight into the performance impact of parallel file systems. Regarding the persistence phase, in the future, we plan to create more unified knowledge object, which will support more benchmarks with different output formats.

Considering knowledge analysis and the use case anomaly detection, the GUI of the knowledge explorer will be extended. This allows a unified presentation of I/O knowledge, but also the support of additional chart types, including heat map and bounding box. Moreover, another important aspect is to support the ability to add knowledge manually through the web-based user interface. Thus, allows a unified analysis process. In order to use the knowledge for I/O optimization, the tool needs to be extended by further optimization modules such as I/O pattern extractor and recommendation module in the future. Furthermore, the knowledge objects can be used as training data for linear regression analysis to make I/O performance predictions.

VII. Conclusion

With the growing gap between computational power and underlying storage systems [2], [4], and the lack of I/O expertise [4], [16], I/O resources need to be used efficiently, and the required knowledge needs to be easily accessible. For this purpose, we present in this paper a generic workflow (section III), and a possible implementation (section V) that realizes each phase of the knowledge cycle.

Through our generic workflow, I/O knowledge can be generated, extracted, persisted, and analyzed in a structured and reproducible manner and therefore can be applied to specific usage scenarios such as anomaly detection, new knowledge generation, but also I/O optimization. Through the experiments and resulting knowledge, we can clearly show that our implementation is fully compatible with the I/O knowledge cycle. Hence, we are confident that our proposed workflow and corresponding implementation can provide a deeper I/O understanding to the user with minimal prior knowledge, thus enabling a more efficient use of the underlying I/O resources. Further information, including additional screenshots, are available on our companion GitHub repository: https://github.com/lalilalalalu/IO-Knowledge.git.

References

[1] E. Suarez, N. Eicker, and T. Lippert, "Modular Supercomputing Architecture: from Idea to Production," in *Contemporary high performance computing*, pp. 223–255, CRC Press, 2019.

[2] S. Neuwirth and A. K. Paul, "Parallel I/O Evaluation Techniques and Emerging HPC Workloads: A Perspective," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 671–679, IEEE, 2021.

[3] J. L. Bez, H. Tang, B. Xie, D. Williams-Young, R. Latham, R. Ross, S. Oral, and S. Byna, "I/O Bottleneck Detection and Tuning: Connecting the Dots using Interactive Log Analysis," in *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW)*, pp. 15–22, IEEE, 2021.

[4] H. Tang, B. Xie, S. Byna, P. Carns, Q. Koziol, S. Kannan, J. Lofstead, and S. Oral, "SCTuner: An Autotuner Addressing Dynamic I/O Needs on Supercomputer I/O Subsystems," in *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW)*, pp. 29–34, IEEE, 2021.

[5] S. Neuwirth, *Accelerating Network Communication and I/O in Scientific High Performance Computing Environments*. PhD thesis, 2019.

[6] S. El Sayed Mohamed, *Analysis of I/O Requirements of Scientific Applications*. No. FZJ-2018-04789, Jülich Supercomputing Center, 2018.

[7] K. Q. Prabhat and Q. Koziol, "High performance parallel I/O," *ANTYPAS, Katie; YAO, Yushu. Overview of I/O Benchmarking. California, USA, Quincey Koziol*, pp. 279–288, 2014.

[8] M. Seiz, P. Offenhäuser, S. Andersson, J. Hötzer, H. Hierl, B. Nestler, and M. Resch, "Lustre I/O performance investigations on Hazel Hen: experiments and heuristics," *The Journal of Supercomputing*, vol. 77, no. 11, pp. 12508–12536, 2021.

[9] 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)*, pp. 80–87, IEEE, 2019.

[10] H. Tang, S. Byna, N. A. Petersson, and D. McCallen, "Tuning parallel data compression and I/O for large-scale earthquake simulation," in *2021 IEEE International Conference on Big Data (Big Data)*, pp. 2992–2997, IEEE, 2021.

[11] W. Yu, S. Oral, J. Vetter, and R. Barrett, "Efficiency evaluation of cray XT parallel io stack," in *Cray User Group Meeting (CUG 2007)*, 2007.

[12] "The HDF Group - Hierarchical Data Format, version 5.." https://www.hdfgroup.org/solutions/hdf5/. Accessed: 2022-07-3.

[13] "ADIOS team at ORNL - The Adaptable I/O System." https://csmd.ornl.gov/adios/. Accessed: 2022-07-3.

[14] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A high-performance scientific I/O interface," in *SC'03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, pp. 39–39, IEEE, 2003.

[15] T. Patel, S. Byna, G. K. Lockwood, and D. Tiwari, "Revisiting I/O Behavior in Large-Scale Storage Systems: The Expected and the Unexpected," SC '19, ACM, 2019.

[16] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright, "Modular hpc i/o characterization with darshan," in *2016 5th workshop on extreme-scale programming tools (ESPT)*, pp. 9–17, IEEE, 2016.

[17] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, R. Aydt, Q. Koziol, M. Snir, *et al.*, "Taming parallel I/O complexity with auto-tuning," in *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, IEEE, 2013.

[18] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The Vampir Performance Analysis Tool-Set," in *Tools for High Performance Computing* (M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, eds.), (Berlin, Heidelberg), pp. 139–155, Springer Berlin Heidelberg, 2008.

[19] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011* (H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, eds.), (Berlin, Heidelberg), pp. 79–91, Springer Berlin Heidelberg, 2012.

[20] R. Dietrich, F. Winkler, A. Knüpfer, and W. Nagel, "PIKA: Center-Wide and Job-Aware Cluster Monitoring," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 424–432, 2020.

[21] J. M. Kunkel, M. Zimmer, N. Hübbe, A. Aguilera, H. Mickler, X. Wang, A. Chut, T. Bönisch, J. Lüttgau, R. Michel, *et al.*, "The SIOX architecture–coupling automatic monitoring and optimization of parallel I/O," in *International Supercomputing Conference*, pp. 245–260, Springer, 2014.

[22] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale I/O workloads," in *2009 IEEE International Conference on Cluster Computing and Workshops*, pp. 1–10, IEEE, 2009.

[23] C. Xu, S. Snyder, V. Venkatesan, P. Carns, O. Kulkarni, S. Byna, R. Sisneros, and K. Chadalavada, "DXT: Darshan extended tracing," tech. rep., Argonne National Lab.(ANL), Argonne, IL (United States), 2017.

[24] "HPC IO Benchmark Repository." https://github.com/hpc/ior. Accessed: 2022-07-6.

[25] R. Liem, D. Povaliaiev, J. Lofstead, J. Kunkel, and C. Terboven, "User-Centric System Fault Identification Using IO500 Benchmark," in *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW)*, pp. 35–40, IEEE, 2021.

[26] "IO500 Storage Benchmark." https://github.com/IO500. Accessed: 2022-07-8.

[27] J. Kunkel, G. F. Lofstead, and J. Bent, "The Virtual Institute for I/O and the IO-500.," tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2017.

[28] J. Layton, "Io pattern characterization of hpc applications," in *International Symposium on High Performance Computing Systems and Applications*, pp. 292–303, Springer, 2009.

[29] H. Shan and J. Shalf, "Using IOR to analyze the I/O performance for HPC platforms," tech. rep., Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2007.

[30] S. Lührs, "Automated benchmarking with JUBE," in *HPC Knowledge Meeting'20*, no. FZJ-2020-02622, Jülich Supercomputing Center, 2020.

[31] "HACC I/O Benchmark Summary." https://asc.llnl.gov/sites/asc/files/2020-06/HACC_IO_Summary_v1.0.pdf. Accessed: 2022-07-28.

[32] "Argonne National Laboratory. PyDarshan: HPC I/O characterization tool." https://www.mcs.anl.gov/research/projects/darshan/docs/pydarshan/index.html. Accessed: 2022-07-2.

[33] "What Is SQLite?." https://www.sqlite.org/index.html. Accessed: 2022-07-6.

[34] "The CPU Cluster FUCHS-CSC." https://csc.uni-frankfurt.de/wiki/doku.php?id=public:service:fuchs. Accessed: 2022-07-2.

[35] F. Wang, S. Oral, G. Shipman, O. Drokin, T. Wang, and I. Huang, "Understanding lustre filesystem internals," *Oak Ridge National Laboratory, National Center for Computational Sciences, Tech. Rep*, vol. 120, 2009.

[36] "{GPFS}: A {Shared-Disk} File System for Large Computing Clusters, author=Schmuck, Frank and Haskin, Roger," in *Conference on File and Storage Technologies (FAST 02)*, 2002.

[37] "OrangeFS Development Team - The OrangeFS Project." http://www.orangefs.org/. Accessed: 2022-07-3.