TOKIO on ClusterStor: Connecting Standard Tools to Enable Holistic I/O Performance Analysis

Glenn K. Lockwood, Nicholas J. Wright
Lawrence Berkeley National Laboratory
{glock, njwright}@lbl.gov

Shane Snyder, Philip Carns
Argonne National Laboratory
{ssnyder, carns}@mcs.anl.gov

George Brown, Kevin Harms
Argonne National Laboratory
gbrown@anl.gov, harms@alcf.anl.gov

***Abstract*—At present, I/O performance analysis requires different tools to characterize individual components of the I/O subsystem, and institutional I/O expertise is relied upon to translate these disparate data into an integrated view of application performance. This process is labor-intensive and not sustainable as the storage hierarchy deepens and system complexity increases. To address this growing disparity, we have developed the Total Knowledge of I/O (TOKIO) framework to combine the insights from existing component-level monitoring tools and provide a holistic view of performance across the entire I/O stack.**

A reference implementation of TOKIO, pytokio, is presented here. Using monitoring tools included with Cray XC and ClusterStor systems alongside commonly deployed community-supported tools, we demonstrate how pytokio provides a lightweight foundation for holistic I/O performance analyses on two Cray XC systems deployed at different HPC centers. We present results from integrated analyses that allow users to quantify the degree of I/O contention that affected their jobs and probabilistically identify unhealthy storage devices that impacted their performance. We also apply pytokio to inspect the utilization of NERSC's DataWarp burst buffer and demonstrate how pytokio can be used to identify users and applications who may stand to benefit most from migrating their workloads from Lustre to the burst buffer.

I. Introduction

The Total Knowledge of I/O (TOKIO) framework [1] connects data from component-level monitoring tools across the I/O subsystems of HPC systems. Rather than build a universal monitoring solution and deploy a scalable data store to retain all monitoring data, TOKIO connects to *existing* best-in-class monitoring tools and databases, indexes these tools' data, and presents the data from multiple connectors in a single, coherent view to downstream analysis tools and user interfaces.

At a high level, we propose three basic axioms of characterizing storage and I/O subsystems today:

- **I/O systems are complex systems that fail in complex ways.** High-performance parallel I/O is made possible by increasingly complex I/O subsystems which attempt to satisfy three orthogonal goals: (1) delivering high bandwidth and low latency in a scalable fashion, (2) providing durability and persistence of data, and (3) enabling as much POSIX compatibility as possible to enable portability. Being complex systems, storage systems also fail in complex ways; while an outright outage may be straightforward to diagnose, the more complex failure modes, such as fail-slow scenarios [2], require commensurately more complex diagnosis procedures.
- **Complex architectures beget a complex set of essential tools.** Storage systems are composed of software, middleware, and hardware created by a variety of different technology providers who are the world's experts in the components they provide. As a result, they are also best qualified to provide the tools that report on the performance and well-being of those components.
- **Broad expertise is required to gain insight from these tools.** Drawing insight from a diversity of tools that operate on complex systems requires a combination of expert knowledge of the entire I/O subsystem and well-defined and composable analysis methods.

These three axioms speak to the need for I/O characterization frameworks that take a holistic approach to performance analysis and capture the full resolution of the available telemetric data on I/O subsystems while simultaneously providing simpler, semantically relevant access to those data. Such a framework would be built upon the following design criteria, motivated by the three axioms of I/O characterization frameworks:

1) **Use existing tools already in production.** A variety of tools already exist to characterize individual components of the I/O subsystem, and different HPC centers often already have many of these tools in production. Using existing tools not only leverages the component-specific expertise of the individuals who created each tool, but it reduces the burden of integrating the framework with HPC centers since it builds upon the tools with which facilities staff are already familiar.
2) **Leave data where it is.** Attempting to aggregate all component-level telemetric data in a centralized data warehouse often requires coercing the output data to fit a schema, and this process can be lossy. Furthermore, the overheads of maintaining a data warehouse suitable for aggregating all monitoring data can be high if one is not already deployed at an HPC facility. Thus, the framework should meet the tools where they are and work with data as it is natively generated. Organizing and querying the data can be achieved by indexing the different data types and data sources rather than replicating and normalizing them.
3) **Make data as accessible as possible.** The principal role of the framework is to provide semantically sensible and

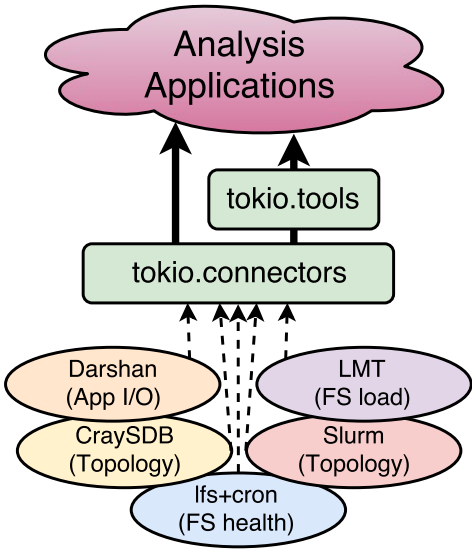


Fig. 1. TOKIO Architecture. `tokio.connectors` take input from component-level monitoring tools in their native output formats and expose that data to upstream analysis in standard data formats such as key-value pairs, pandas DataFrames, and NumPy arrays. `tokio.tools` provide indices, site-specific data placement information, and enhanced usability.

consistent access to the diversity of data generated by component-level tools. Users should be able to request a single logical quantity (such as bytes written to a storage server) and be given the data in a standard data format without having to understand the tool that collected that data. Furthermore, the framework should not require escalated privileges to be useful; access controls are better handled by the tools and data sources that the framework indexes.

In previous work, we presented the Total Knowledge of I/O (TOKIO) framework as a formalization of these requirements on a conceptual basis and demonstrated the new insights into I/O performance that such an approach enables [1]. In this work, we present *pytokio*, a reference implementation of the TOKIO framework implemented in Python 2, which is available as a freely downloadable library. We also present accompanying data analysis tools and services that the community can use to extract meaningful and holistic insights from the data that is already produced on Cray XC and ClusterStor platforms.

II. TOKIO Architecture & Implementation

To meet the design criteria outlined in Section I, the TOKIO framework is comprised of four layers as shown in Figure 1. Each layer is then composed of modules which are largely independent of each other to allow TOKIO to integrate with whatever selection of tools a given HPC center has running in production. That said, certain tools are *de facto* standards for I/O performance analysis, and those tools will be highlighted in the following architectural description as examples of how TOKIO enables holistic analysis.

A. Component-level monitoring tools

Following design criterion #1, TOKIO builds upon whatever monitoring and profiling tools are already in production at an HPC facility. Although the tools and infrastructure supporting them exist outside of the scope of TOKIO, we identify several broad categories of metrics that are relevant to understanding I/O performance.

1) Application behavior: The I/O patterns that a program issues can affect performance dramatically regardless of the underlying storage system. This behavior can be automatically profiled using tools like Darshan [3], which is a link-time library that transparently records concise, bounded statistics about an application's I/O behavior. It is commonly included with all compiled applications at CUG member sites including NERSC, ALCF, NCSA, and KAUST [1], [4]–[6]. Complementary information from client-side monitoring such as the collection of file system client metric collection enabled by RUR [7] also falls in this category.

In all cases, though, understanding application behavior can yield insights into client-side caching and intra-node or inter-application contention [8] that may not be quantifiable from other parts of the I/O subsystem. However, the sensitivity of application performance to jitter caused by continuous performance monitoring on compute nodes results in these application behavior data being scalar in nature; collecting these data over time can be prohibitively disruptive.

2) Storage system traffic: The servers that manage file data and metadata are inherently shared by all users of an HPC system, and as a result, are intuitively susceptible to resource contention from competing jobs. It follows that monitoring the traffic and load on each storage system server is valuable for identifying contention-related issues that may be beyond the visibility of individual users and their jobs.

Just as there are a variety of storage system implementations, there are a variety of implementation-specific tools for collecting these data. On Cray ClusterStor systems, Lustre Monitoring Tools (LMT) have historically collected this data [9], and Cray's newer Caribou and Seastream infrastructure is being positioned as the preferred alternative for the future [10]. File system-specific tools for Cray DataWarp systems do not yet exist, but NERSC has demonstrated that collectd and Elasticsearch can collect and store device-level load data to similar ends [11]. Unlike application behavior data, collecting storage system traffic at high frequency results in minimal perceptible jitter relative to the latency of typical I/O operations. As a result, storage system traffic is commonly recorded as time series data with an ideal sampling rate greater than $^1/_{60}$ Hz [12].

3) Storage system health: There are many circumstances in which a component of a storage system is known to be in an available but degraded state of performance as a result of a temporary failure or condition. Conditions such as storage device failovers (where a server may have to take on the load of a failed partner server) or RAID rebuilds (where device bandwidth is being consumed by reconstruction of data from

parity) can introduce significant, long-tail performance losses to files that are striped across those degraded devices [13].

As with storage system traffic data, storage system health data is often monitored using implementation-specific tools that understand architecture-specific failures that degrade performance. In the case of Lustre, the `lctl dl -t` command is sufficient to identify failed-over OSTs from any Lustre client, while `lfs df` provides information that implicates performance loss due to high file system fullness. Similarly, DataWarp can be monitored on a per-device basis using standard tools such as `smartctl` or vendor-specific tools such as the Intel SSD Data Center Tool (ISDCT) [14].

4) Job topology: The effects of locality on I/O performance within high-diameter networks have been well documented [15]–[17], and modern high-radix topologies continue to be susceptible to topology-induced performance variation [18]. Obtaining the topological mapping of a given job's compute nodes across a given network fabric requires several different tools that combine job↔node mappings from the system resource manager with the node↔coordinate mapping from the system component which tracks global system state.

In practice, both of these tool sets are highly system-specific; the job↔node mappings may be provided by a resource manager such as Slurm [19] or ALPS [20], and node↔topology mappings are provided by the Cray Service Database. Fortunately, the relationship between nodes and topological coordinates in the fabric is a relatively static mapping, and it can be aggressively cached so long as that cache is expired any time the fabric topology is altered.

5) Network traffic: The networks over which I/O transits, including both the high-speed network and the back-end storage network, are shared resources and therefore susceptible to contention from other workloads. Unlike the storage system traffic data, though, network traffic loads tend to be very complex since they are a function of loads at both compute and storage server endpoints as well as incidental traffic being routed over the same links.

On Cray systems, the Gemini Counter Performance Daemon [21], [22] or AriesNCL [23] can be used to collect the performance counters available on Aries routers. In practice, the complexity and scale of these network traffic data make them challenging to collect effectively. LDMS [24] has emerged as a scalable infrastructure for collecting these data system-wide, while PAPI has been demonstrated to collect these data on a per-job basis [25].

B. TOKIO connectors

The foundational layer of TOKIO are *connectors*, which are independent, modular components that provide an interface between the individual component-level tools described above and the higher-level TOKIO layers described later. Each connector interacts with the native interface of a component-level tool and provides data from that tool in the form of a tool-independent interface.

As a concrete example, consider the *LMT component-level tool* which exposes Lustre file system workload data through a MySQL database hosted on the ClusterStor management node [9]. The *LMT database connector* is responsible for establishing and destroying connections to this MySQL database as well as tracking stateful entities such as database cursors. It also encodes the schema of the LMT database tables, effectively abstracting the specific workings of the LMT database from the information that the LMT tool provides. In this sense, a user of the LMT database connector can use a more semantically meaningful interface (e.g., `lmtdb.get_mds_data()` to retrieve metadata server loads) without having to craft SQL queries or write any boilerplate MySQL code.

At the same time, the LMT database connector does *not* modify the data retrieved from the LMT MySQL database before returning it. As such, using the LMT database connector still requires an understanding of the underlying LMT tool and the significance of the data it returns. This design decision restricts the role of connectors to being convenient interfaces into existing tools that eliminate the need to write glue code between component-level tools and higher-level analysis functions.

All connectors also provide serialization and deserialization methods for the tools to which they connect. This allows the data from a component-level tool to be stored for offline analysis, shared among collaborators, or locally cached for rapid subsequent accesses. Continuing with the LMT connector example, the data retrieved from the LMT MySQL database may be serialized to formats such as SQLite. Conversely, the LMT connector is also able to load LMT data from these alternative formats for use via the same downstream connector interface (e.g., `lmtdb.get_mds_data()`). This dramatically simplifies some tasks such as publishing analysis data that originated from a restricted-access data source or testing new analysis code.

The pytokio implementation of TOKIO implements each connector as a Python class. Connectors which rely on stateful connections, such as those which load data from databases, generally wrap a variety of database interfaces. Connectors which operate statelessly, such as those that load and parse discrete log files, are generally derived from Python dictionaries and self-populate when initialized. Where appropriate, these connectors also have methods to return different representations of themselves; for example, many connectors provide a `to_dataframe()` method that returns the requested connector data as a pandas DataFrame.

The initial release of pytokio, version 0.9, includes the connectors listed in Table I. The first six connectors listed (CraySdb through Slurm) connect to component-level tools that are either pre-installed on Cray XC and ClusterStor systems or commonly deployed open-source tools. The latter three connectors (CollectdEs, NerscIsdct, and NerscJobsDb) rely on third-party infrastructure and/or use non-default schemata or encoding to represent data. That said, these three connectors contain the logic necessary to create more generic connectors for consuming output from sources like Elasticsearch (from CollectdEs) or smartctl (from NerscIsdct). Finally, the Hdf5 connector provides an interface into the TOKIO Time Series

TABLE I
TOKIO CONNECTORS AVAILABLE IN PYTOKIO 0.9

Connector	Parent Class	Component-Level Tool	Data Provided
CraySdb	dict	Cray Service Database	Node network topology
Darshan	dict	Darshan 3.0 or newer	Application-level I/O profiles
LfsOstMap	dict	Lustre `lctl dl -t`	Failover status of OSTs and OSSes
LfsOstFullness	dict	Lustre `lfs df`	Fullness of OSTs
LmtDb	N/A	MySQL with LMT schema	Lustre server-side traffic
Slurm	dict	Slurm	Job IDs, node lists, start/end times
Hdf5	h5py.File	Multiple TOKIO services	File system server-side time series
CollectdEs	N/A	Elasticsearch with collectd schema	Burst buffer server-side traffic
NerscIsdct	dict	ISDCT w/ NERSC directory structure	Burst buffer SSD SMART data
NerscJobsDb	N/A	MySQL with NERSC accounting schema	System-wide job workload

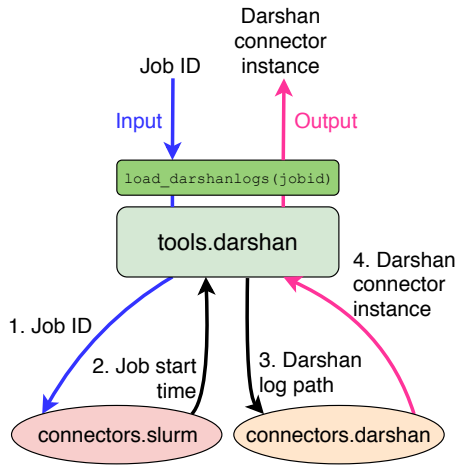


Fig. 2. Darshan tools interface for converting a Slurm Job ID into Python interfaces into one or more Darshan logs. User specifies a Slurm Job ID; the Darshan tool retrieves the date this job ran from Slurm, and then uses this date to find the location of any relevant Darshan logs in the site-wide Darshan log repository. The tool then connects to each log and returns a dictionary-like Darshan connector instance for each.

data format, a derivative of HDF5 that is used to store file system traffic data in a connector-agnostic format. This TOKIO Time Series format is described in more detail in Section III-C.

C. TOKIO tools

TOKIO *tools* are implemented on top of connectors as an optional set of interfaces that are semantically closer to how analysis applications may wish to access component-level data. To this end, the TOKIO tools interfaces typically serve two purposes: encapsulating site-specific information on how certain data sources are indexed or where they may be found, and providing higher-level abstractions atop one or more connectors to mask the complexities or nuances of the underlying data sources.

1) Encapsulating site-specific information: pytokio factors out all of its site-specific knowledge from connectors into a single site-specific configuration file. This configuration file is composed of arbitrary JSON-encoded key-value pairs which are loaded whenever pytokio is imported, and the specific meaning of any given key is defined by whichever tool accesses it. Thus, this site-specific configuration data does not prescribe any specific schema or semantic on site-specific information, and it does not contain any implicit assumptions about which connectors or tools are available on a given system.

To illustrate this more concretely, consider the case of the Darshan component-level tool [3]. When deployed system-wide, Darshan automatically saves users' output logs to a pre-defined, site-wide log repository. This repository is structured such that logs are indexed by year, month, and day, and Darshan encodes a variety of metadata (including the user name, executable name, and job id) in each log file's file name. The Darshan *tool* contains the logic required to find Darshan log files in this site-wide repository when given any or all of these metadata attributes. The top-level directory of the site-wide Darshan log repository (e.g., `/global/darshanlogs/`) is site-specific and therefore stored in the pytokio configuration file. However, the directory structure *within* that log repository is dictated by Darshan itself, so the mapping between dates and subdirectories is implemented within the Darshan tool. It is then the responsibility of the Darshan connector to provide an interface into an individual Darshan log file.

2) Providing higher-level abstractions atop connectors: The other role of TOKIO tools are to combine site-specific knowledge and multiple connectors to provide a simpler set of interfaces that are semantically closer to a question that an I/O user or administrator may actually ask. Continuing with the Darshan tool example from the previous section, such a question may be, "How many GB/sec did job #2468187 achieve?" Answering this question involves several steps:

1) Retrieving the start date for job id #2468187 from the system workload manager or a job accounting database
2) Looking in the Darshan repository for logs that match jobid=2468187 on that date
3) Running the "`darshan-parser --perf`" tool on the matching Darshan log and retrieve the estimated maximum I/O performance

pytokio provides connectors and tools to accomplish each one of these tasks:

1) The **Slurm connector** provides `get_job_startend()` which retrieves a job's start and end times when given a Slurm job id
2) The **Darshan tool** provides `find_darshanlogs()` which returns a list of matching Darshan logs when given a job id and the date on which that job ran
3) The **Darshan connector** provides `darshan_parser_perf()` which retrieves I/O performance data from a single Darshan log

Because this is such a routine process when analyzing application I/O performance, the Darshan tools interface implements this entire sequence in a single, higher-level function called `load_darshanlogs()`. This function, depicted in Figure 2, effectively links two connectors (Slurm and Darshan) and provides a single function to answer the question of "how well did job #2468187 perform?" This greatly simplifies the process of developing user-facing tools to analyze Darshan logs. Any analysis tool which uses application I/O performance and operates from job ids can replace hundreds of lines of boilerplate code with a single function call into the Darshan tool, and it alleviates users from having to understand the Darshan log repository directory structure to quickly find profiling data for their jobs.

3) Simplifying portability: TOKIO tools interfaces are also what facilitate portable, highly integrated analyses and services for I/O performance analysis. In the aforementioned examples, the Darshan tools interface assumes that Slurm is the system workload manager and the preferred way to get start and end times for a job id. However, there is also a more generic jobinfo tool interface which serves as a connector-agnostic interface that retrieves basic job metrics (start and end times, node lists, etc) using a site-configurable, prioritized list of connectors.

Consider the end-to-end example shown in Figure 3. In this case, an analysis application's purpose is to answer the question, "What was a job's I/O performance?" To accomplish this, the analysis takes a job id as its sole input and makes a single call into the pytokio Darshan tool's `load_darshanlogs(jobid)` function as previously described. The Darshan tool first uses the jobinfo tool to convert the job id (1) into a start/end time in a site-independent way. The jobinfo tool examines the site configuration and determines that the Slurm connector is the best way to convert the job id (2) into a job start/end time (3), which is passed back to the Darshan tool (4). The Darshan tool then uses the job start time to determine where the job's Darshan log is located in the site-specific repository, and uses this log path (5) to retrieve a connector interface into the log (6). The Darshan tool returns this connector interface to the analysis application (7), which extracts the relevant performance metric (8) and returns it to the end user.

Through this entire process, the analysis application's only interface into pytokio was a single call into the Darshan tools interface. Beyond this, pytokio was responsible for determining both the proper mechanism to convert a job id into a job start time and the location of Darshan logs on the system. Thus, this analysis application is entirely free of site-specific knowledge and can be run at any HPC center to obtain I/O performance telemetry when given a job id. The only requirement is that pytokio is installed at the HPC center, and it is correctly configured to reflect that center's site-specific configurations.

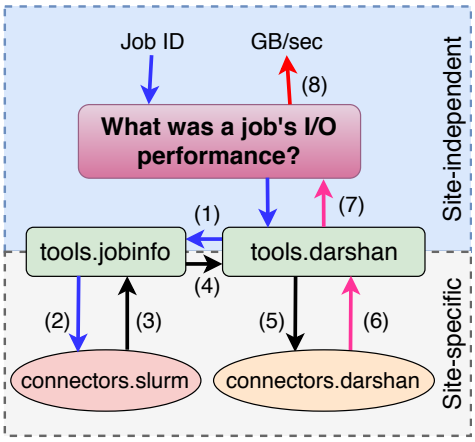


Fig. 3. TOKIO tools interfaces to enable portability. An analysis application answers the question, "What was a job's I/O performance," and it accepts the job's ID as its sole input. The TOKIO tools interface abstracts all of the site-specific information (such as how job ids are mapped to job start times and where Darshan logs are saved) from the higher-level analysis application. Thus, the analysis application can be run at any HPC center without modification provided that center has pytokio installed and correctly configured.

III. Analysis Applications and Services

TOKIO's connector and tool interfaces are simply mechanisms to access I/O telemetry from throughout an HPC center. As illustrated in Figure 3, a higher-level analysis application is required to actually connect pytokio's interfaces to meaningful insight to an end-user. To demonstrate how such an analysis application may be built on top of pytokio, pytokio includes a number of example applications and services that broadly fall into three categories: command-line interfaces into pytokio, statistical analysis tools, and data and analysis services.

A. Command-line interfaces to pytokio

Many connectors and tools present methods and functions that are valuable for users as-is. For example, being able to retrieve application-level I/O performance telemetry with a single job id (as was presented in Section II-C) is an intrinsically useful operation. To expose such useful functions directly to users without requiring that they write python, pytokio includes a set of command-line tools that simply convert command-line options into input arguments, pass these arguments to a single pytokio function, and then return the resulting output as ASCII to stdout.

Perhaps the most immediately valuable tools of this category are the command-line interfaces for each connector's serialization method, which allow specific component-level data to be quickly serialized into a generic and portable format. For example, the LMT connector allows the contents of the LMT MySQL database to be serialized to a local SQLite file. By serializing this data during a time period of interest, LMT

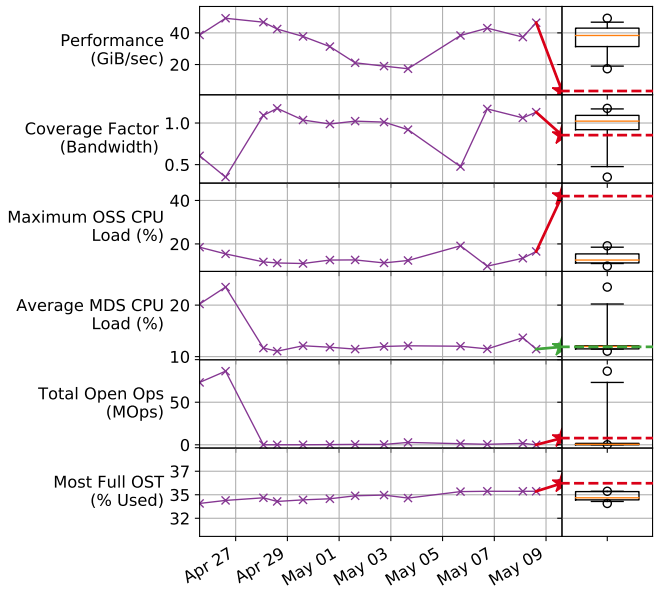


Fig. 4. UMAMI diagram of a simulated science campaign of BD-CATS [26] simulations performed on ALCF's Theta system. "Coverage Factor (Bandwidth)" is the fraction of global file system traffic originating from each BD-CATS job and the remaining metrics represent server-side Lustre loads.

data can be analyzed long after the 24-hour window in which LMT database retains data. Furthermore, because the data is serialized to a standard and portable format, the SQLite database file itself can be shared and analyzed on remote systems for purposes of collaboration or reproducibility.

B. Statistical analysis tools

The modularity of TOKIO's connectors and tools interfaces make it a powerful foundation upon which more sophisticated, integrated analyses can be performed. For example, pytokio arose from a proof-of-concept study that presented the concept of Unified Monitoring and Metrics Interface (UMAMI) diagrams, a visualization that contextualizes I/O performance with variation in other metrics across the I/O subsystem [1]. The tools required for a user to generate these diagrams is included in pytokio as an example of such a statistical analysis.

Figure 4 is an example of such an UMAMI diagram that shows a simulated science campaign where a large-scale particle physics analysis application, BD-CATS [26], was run once per day over the course of two weeks on ALCF's Theta system and its Lustre file system. The topmost panel ("Performance (GiB/sec)") indicates that the job which ran on May 11 showed significantly less I/O performance (under 5 GiB/sec) than previous days (40 GiB/sec). The poor I/O performance on this date coincided with an extraordinarily high CPU load on one or more Lustre OSSes (a high "Maximum OSS CPU Load"), indicating the drastic performance loss could be due to competing compute workloads on the Lustre storage servers. Interestingly, for this series of application executions, instances of increased server-side bandwidth contention (a low "Coverage Factor (Bandwidth)", e.g., on April 26 or May 6) did not coincide with marked decreases in performance, though this is often speculated to be a leading cause for I/O performance issues on HPC systems. This diagram includes data obtained through a number of pytokio connectors: Darshan (Performance), LMT (Maximum OSS CPU Load, Average MDS CPU Load, Total Open Ops) and Lustre health (Most Full OST).

To simplify the process of gathering data from all of these disparate sources of telemetry, pytokio includes the `summarize_job` command-line tool which gathers data about a given job using every available connector. When given either a list of job ids or a list of paths to Darshan log files (which themselves encode job ids), `summarize_job` infers the job start and end times and uses these data to retrieve the relevant Lustre system traffic and health data for the times during which each job ran. It also retrieves the list of nodes used by each job via the Slurm connector and calculates several metrics representing the job's placement on the dragonfly network. All of this data is then flattened into a set of key-value metrics for each job id, and the key-value pairs for all job ids are compiled into a table of comma-separated values which is returned to the user[1]. Each row of this CSV corresponds to a single job, and each column contains a single metric produced by summarizing the output of single connector.

This CSV output of `summarize_job` is then used to generate the UMAMI diagram itself. The UMAMI diagram shown in Figure 4 was generated using a Jupyter notebook, `tokio.analysis.umami.ipynb`, which included in the pytokio repository. This notebook does the following:

1) Loads the CSV file using `pandas.read_csv()`
2) Performs basic filtering to discard job records which do not correspond to the analysis of interest (e.g., if a single job id generated multiple Darshan logs, but only one corresponds to the full-scale simulation execution). Each job record corresponds to a row in the original `summarize_job` output CSV.
3) Builds a set of `UmamiMetric` objects, each essentially representing a vector of measurements of one metric over time. Each metric corresponds to a column in the original `summarize_job` output CSV.
4) Creates a single `Umami` object (essentially an ordered dictionary) and appends each `UmamiMetric` object
5) Generates the UMAMI diagram using `UmamiMetric.plot()`

The Jupyter notebooks included with pytokio are a convenient way to explore data and perform ad-hoc performance analysis with pytokio. These notebooks are examples of an effective design pattern for building analysis capabilities atop pytokio: a potentially useful analysis is first prototyped in notebook format, and once the analysis methods are sufficiently robust, they are converted into a standalone command-line tool that can be used with a greater degree of automation.

C. Data and analysis services

Many component-level monitoring tools are designed for system operators who perform real-time inspection of system

[1]This tool, as with many other pytokio components, uses pandas DataFrames internally to represent tabular data. As such, pytokio can easily output to other supported formats such as JSON.

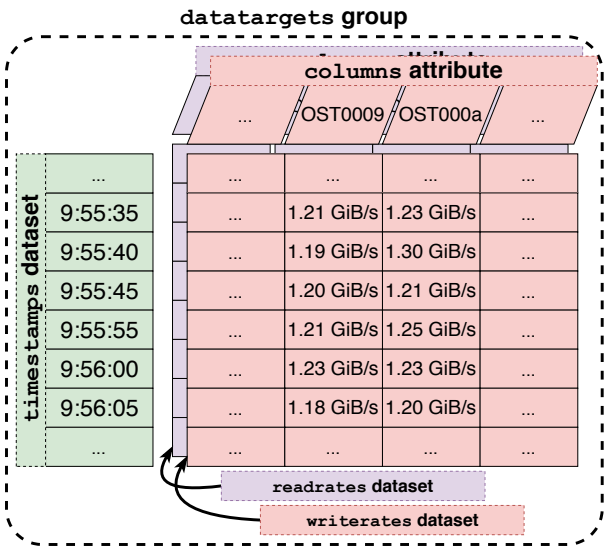


Fig. 5. Organization of the `datatargets` group, which encode traffic to and from file system data targets (Lustre OSTs, GPFS LUNs, or DataWarp NVMe drives), in a TOKIO Time Series file. This example reflects read-rate and write-rate data collected from LMT at its native 5-second sampling frequency between 9:55:35 AM and 9:56:05 AM across two OSTs (0x0009 and 0x000a).

performance and health. Postmortem performance debugging and long-term trend analysis with these tools can only be performed if such real-time, component-level monitoring tools are periodically polled and their outputs recorded so that an historic record can be consulted at a later date. To address this need, pytokio includes several infrastructure components that simplify the process of creating services that automatically poll and archive time-series data, and then serve such archived data through a diversity of simple user interfaces.

1) Archival service for real-time data sources: As described earlier, LMT is a standard component of the Cray ClusterStor software stack which retains high-resolution file system traffic in a MySQL database. In its most common configuration, LMT retains file system traffic measurements (e.g., bytes read and written) on a per-OST basis every five seconds, but purges this full-resolution data from the MySQL database every 24 hours to prevent the database from becoming untenably large and slow. To retain these data at full resolution indefinitely, pytokio includes the `archive_lmtdb` tool which serializes time series data from the LMT database into an indexed, portable file format called the TOKIO Time Series format.

Based on HDF5, the TOKIO Time Series format organizes time series data from different monitoring tools in their own HDF5 groups, and one such group is schematically depicted in Figure 5. This group (named `datatargets`) contains two two-dimensional datasets (`readrates` and `writerates`) which contain measurements sampled at a fixed frequency. Each row of these datasets corresponds to a single point in time in the time series (e.g., 9:55:35 AM), and each column corresponds to a single component being measured (e.g., OST #0009). To index these datasets in time, a single one-dimensional `timestamps` dataset is also included in each HDF5 group to describe the absolute times corresponding to each row in the other datasets. Because data encoded in the TOKIO Time Series format is stored at a fixed frequency for each group, indexing data in time can be done arithmetically using the first timestamp (to establish an absolute reference) and the sampling frequency (to calculate a relative offset). The columns of each dataset are labeled using an HDF5 dataset attribute which can also be indexed.

The `archive_lmtdb` tool that generates these TOKIO Time Series files acts as a bridge between the LMT database connector and the HDF5 connector. At a high level, it performs data conversion when given a time range and a target LMT database; this core function is supplemented by a thin in-memory caching layer for performance and the logic necessary to ensure the idempotency of updates to existing TOKIO Time Series files.

To automate the archival of LMT data into this TOKIO Time Series format, pytokio's ClusterStor companion repository contains a simple archival service that creates and maintains a library of TOKIO Time Series files that are organized into a date-indexed directory structure. It creates one TOKIO Time Series file per calendar day, and at a configurable frequency, checks to determine if the day's TOKIO Time Series file is more than an hour out of sync with the LMT database. If it is, the pytokio archival service uses `archive_lmtdb` to retrieve an hour's worth of new data, update the corresponding day's TOKIO Time Series file, and if necessary, roll over to a new day's file. This service is resilient to falling out of sync with the LMT database due to interruptions in service and has minimal dependencies, allowing it to be easily deployed on any Cray ClusterStor platform. To date, pytokio and the LMT archival service has been running in production on the Edison XC-30 and Cori XC-40 systems at NERSC and the Theta XC-40 system at the Argonne Leadership Computing Facility. At NERSC, this archival service is deployed full-time using cron jobs, while at the ALCF it is deployed as part of a broader Jenkins-based continuous integration framework.

The pytokio archival service is sufficiently simple and modular to be able to poll and archive the data from any real-time monitoring tool. For example, NERSC also runs a collectd archival service to retain file system traffic data from the DataWarp burst buffer servers on Cori. These burst buffer data are retrieved from NERSC's Elasticsearch-based Data Collect [11] and encoded in TOKIO Time Series files that are schematically identical to those generated by `archive_lmtdb`. Similarly, Lustre failover and fullness data is archived every five minutes using the Lustre LfsOstMap and LfsOstFullness connectors described in Table I. In all of these cases, the date-based indexing performed by the pytokio archival service is used to enable the rapid lookup of data from these sources for a given point in time, and in all cases, the archival service runs from an unprivileged account.

2) Data services for archived data: The pytokio archival service provides a mechanism by which high time resolution data can be stored on a file system yet still be quickly accessed via date-based file indexing. To make these data

more accessible to users, pytokio also provides an HDF5 tools interface which serves a very similar function as the Darshan tools interface described in Section II-C. Given a range of time and a file system name, the `tools.hdf5` interface locates the correct TOKIO Time Series file(s) containing the date ranges requested, loads the relevant datasets, and stitches the data together into pandas DataFrames indexed by time.

While this functionality is exposed via the Python-based `tools.hdf5` interface, it is very simple to expose these capabilities through a REST API as well. To demonstrate this, pytokio includes a companion repository that demonstrates a Flask-based application wrapper which exposes the aforementioned `tools.hdf5` interface as a REST endpoint. Despite its simplicity, this is an extremely powerful way to connect users with I/O performance data; for example, JavaScript-based web dashboards can query such a pytokio REST service for JSON-encoded data, then display it as an interactive performance plot using Highcharts or D3. Similarly, the UMAMI diagram previously described (Figure 4) could be statically generated and served on-demand to a web-based frontend to enable quick diagnoses of poorly performing applications.

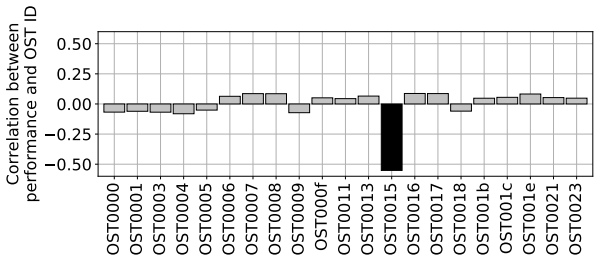
IV. Case Studies

The pytokio package and its accompanying services described in Sections II and III provide the interfaces and infrastructure necessary to explore I/O performance in a holistic manner. To demonstrate this, we present several case studies where pytokio has been applied to solve specific operational problems or gain new insight into storage system behavior. In all cases, file system traffic data archived using the pytokio archival service is combined with data from other sources such as Darshan to identify and confirm behavior that would otherwise be ambiguous from a single component-level monitoring tool.

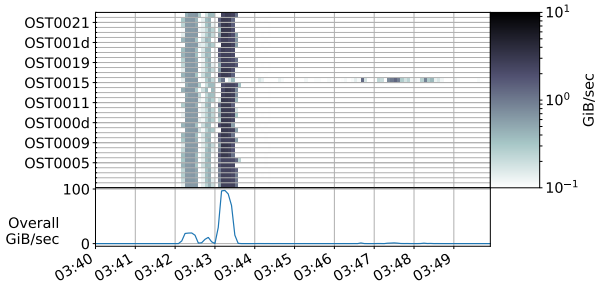
A. Performance analysis of individual jobs

Answering the question of why I/O was slow for a user's job was one of the principal motivators behind developing pytokio, and the UMAMI approach described in Section III-B goes a long way towards answering this question. In the absence of a series of jobs with similar I/O patterns from which an UMAMI diagram can be assembled, though, the `summarize_job` tool can still provide useful metrics such as the degree to which the user's job experienced I/O contention with others (its coverage factor [1]).

However, there are some causes of poor I/O performance that are not well resolved by spatially reduced data as is provided by `summarize_job`. For example, a single slow Lustre OST can dramatically impact overall I/O performance [13], but detecting this condition requires examining the performance of each OST individually. Because the specific malady of a straggling OST is sufficiently common in highly parallel Lustre systems, pytokio includes a `darshan_bad_ost` analysis tool that attempts to statistically identify slow OSTs using the per-file performance and OST mappings contained in Darshan logs.



(a) Correlation between I/O performance and Lustre OST



(b) Per-OST performance over time

Fig. 6. (a) Correlation between per-file application I/O performance and the OSTs to which each file was mapped (from Darshan); shading indicates statistical significance, with darker bars being more significant. (b) Per-OST I/O performance measured over the same time that the job in (a) was running (from LMT). In both (a) and (b), OST0015 is identified as showing abnormally poor performance.

This tool first estimates per-file I/O bandwidths by dividing the total bytes read/written to each file by the time the application spent performing I/O to that file. It then uses data from Darshan's Lustre module [27] to map these performance estimates to the OSTs over which each file was striped. With the list of OSTs and performance measurements corresponding to each OST, the Pearson correlation coefficient is then calculated between performance and each individual OST.

Figure 6a shows the result this correlation analysis for a slow-running job that performed file-per-process I/O to files with a stripe width of 1 on NERSC Edison's "scratch3" Lustre file system. For almost all OSTs, the correlation between performance and OST ID wavers around zero, indicating minimal correlation. However, OST0015 stands out in stark contrast; it correlates strongly and negatively with performance. This inverse relationship between file I/O performance and files existing on this OST suggests that this OST is in an unhealthy state and is not delivering the same level of performance as its peers.

To confirm that OST0015 is indeed behaving abnormally relative to its peers, spatially resolved bandwidth measurements from the LMT connector can be retrieved using the pytokio LMT tool and rendered using a heatmap visualization notebook provided with pytokio. The result of this temporally and spatially resolved OST performance is shown in Figure 6b. Whereas almost all OSTs during this job's write phase were able to complete their I/O between 3:43 AM and 3:44 AM, OST0015 shows a long tail of performance, with relatively little I/O activity between 3:43 AM and 3:44 AM but I/O activity still occurring as late as 3:48 AM. Using both statistical

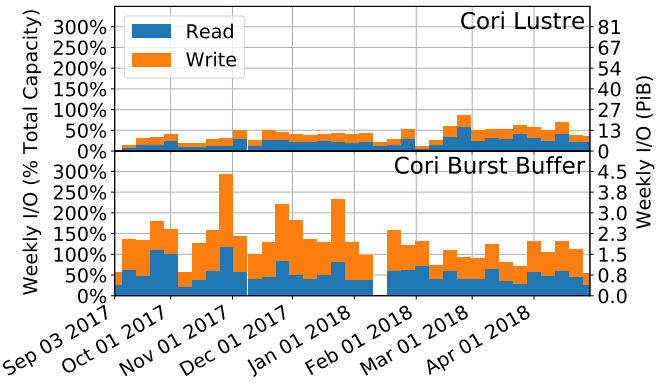


Fig. 7. Total I/O traffic read and written to Cori's Lustre file system (top, obtained from LMT) and DataWarp burst buffer (bottom, obtained from SMART data) per week. Left axis expresses I/O volumes normalized to the total capacity of each storage system; right axis shows the absolute I/O volumes in PiB (2^{50} bytes). Overall read/write ratios are 0.568 (Lustre) and 0.429 (Burst Buffer).

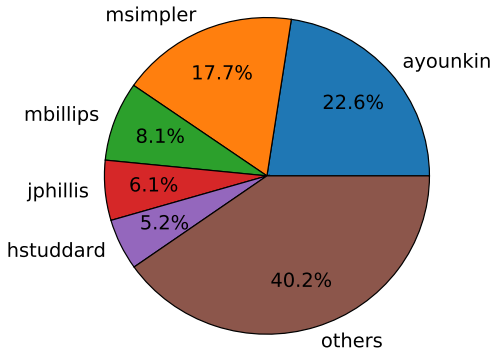


Fig. 8. Top five I/O users on the Cori Lustre file system during April 2018 as measured by Darshan. Data only reflects I/O performed by MPI applications that produced valid Darshan logs. User names shown are pseudonyms.

analysis of Darshan data and a visualization of high-resolution LMT data, the poor performance of this job can be confidently attributed to a poorly performing OST.

While this method is a useful user-facing tool for performance analysis, it can also be a valuable tool for systems engineers. A number of HPC facilities perform automated, daily I/O benchmarks to establish baseline levels of performance variation[1], [28]. Automatically inspecting the Darshan logs from these performance probes using `darshan_bad_ost` enables the detection of more complex, fail-slow scenarios caused by individual component degradations [2]. Automatically running `darshan_bad_ost` with appropriate correlation and significance thresholds allows HPC operators to be alerted whenever a Darshan log is generated that implicates a subset of OSTs as causing overall performance degradation.

B. User engagement for burst buffers

DataWarp-based burst buffers enable users to dynamically provision high-performance flash storage in the form of scratch instances that provide private, ephemeral parallel file systems [29]. Although this enables much higher, more reliable performance than a globally shared Lustre file system, users must consciously incorporate DataWarp into their application workflows to realize these benefits. The "opt-in" nature of DataWarp, NERSC's steady stream of new users, and a general lack of awareness of I/O issues results in a number of NERSC users continue to rely exclusively on Cori's Lustre file system for their I/O-intensive workloads despite the potential benefits of Cori's burst buffer.

To determine how well balanced the utilization of the Lustre file system and DataWarp burst buffer are, NERSC uses a pytokio-based service that provides automated reporting on the total I/O traffic reported by both storage systems. Figure 7 shows the amount of bytes read from and written to both Cori's Lustre file system and burst buffer on a weekly basis. The burst buffer sees traffic equivalent to 120% of its total capacity moved every week on average, while Cori's Lustre file system averages 39%. Because Cori's Lustre is $> 17\times$ more capacious than its burst buffer, though, these I/O volumes reflect a significant disparity–the burst buffer sees 1.85 PiB/week of traffic, while the Lustre file system sees 10.6 PiB/week. Furthermore, Cori's Lustre capacity fills at a rate of $\approx 10\%$ of its weekly write volume, indicating that a significant amount of this Lustre traffic is for temporary data that is not retained.

These data confirm that, despite the availability of Cori's burst buffer, Cori's Lustre file system still experiences a significant amount of scratch-like I/O traffic. In addition, the write volumes for the burst buffer in Figure 7 reflect an average of 0.117 drive writes per week, while the SSDs in Cori's burst buffer are rated for 70 drive writes per week. Thus, we can conclude that Cori's Lustre file system still has a significant amount of scratch-like I/O it can shed to the burst buffer, and the burst buffer has more than enough endurance to sustain such an increase in its workload.

To identify workloads that may be good candidates for migration to Cori's burst buffer, NERSC continuously monitors the application I/O load data targeting Lustre. A pytokio-based service scans and indexes the Darshan logs generated by user jobs on Cori on a daily basis at NERSC to simplify the process of identifying jobs that perform significant I/O to a specific file system. pytokio also includes the `darshan_scoreboard` command-line tool that enables simple querying of these Darshan log indices to determine which applications and users are generating the highest I/O traffic. Combining these tools results in a daily, weekly, or monthly "scoreboard" that identifies the individuals producing the most significant I/O to each storage system. An example of such a scoreboard is shown in Figure 8, which reveals that almost 60% of the I/O traffic captured by Darshan is generated by five users.

Simultaneously, NERSC continuously monitors the file system traffic data targeting Cori's burst buffer to monitor its utilization and wear rate. With these two pytokio-derived services, NERSC staff receive regular automated reports that indicate (1) which users are responsible for the largest fraction of I/O traffic to Lustre, and (2) when the burst buffer is not being heavily utilized. With this information, staff are able to engage with specific users about migrating their workload

to use the burst buffer, effecting significant impact on both improving performance for major I/O workloads and reducing the load on the Lustre file system for other users.

C. Identifying major factors affecting performance

Daily I/O benchmarks can be used to develop a quantitative understanding of how different components of the I/O subsystem affect I/O performance. Applying `summarize_job` to collect all available telemetry associated with daily benchmarks provides a series of performance snapshots *and* the factors that contributed to that performance. By applying simple correlation analysis–calculating the Pearson correlation between I/O performance (measured by Darshan) and every other measured metric–pytokio can be used to shed light on how sensitive I/O performance is to changes in different parts of the I/O subsystem.

Figure 9 shows the results of such a correlation analysis over two file-per-process workloads. The overall I/O bandwidth, as estimated by Darshan, was compared to the nine component-level measurements listed in the table. Although most of correlations between each metric and the four workloads were found to be statistically insignificant (p-value $> 10^{-5}$ for $\approx$ 315 observations each), these results identify the following notable relationships.

All four workloads correlate positively with the bandwidth coverage factor. That is, performance tends to be higher when the file system is not providing bandwidth to multiple workloads simultaneously. While this finding is intuitive, the fact that the correlation coefficients are all well short of 1.0 indicate that bandwidth contention is far from the only source of performance loss.

Fast write workloads correlate with high coverage factors for `open(2)` operations. Assuming that the coverage factor for `open(2)` operations is inversely proportional to the metadata load of the file system, this is also intuitive; in both write workloads, files must be created before they can be written, whereas read workloads simply have to open existing files. It follows that write workloads, which are more metadata-intensive, are more sensitive to competing metadata-intensive workloads.

Write performance correlates positively with average OSS CPU load, with smaller-transfer sizes (IOR) correlating more strongly than larger (HACC). This is an important example of correlation not implying causation because high OSS CPU load is actually a *result* of high write performance in this case; higher OSS CPU loads do not cause better performance. The reason for these relationships is likely influenced by ClusterStor's use of GridRAID, which uses the CPU to calculate parity on writes but does not verify parity on reads. Furthermore, calculating GridRAID parity on HACC's large writes may be more efficiently pipelined than IOR's smaller writes, resulting in HACC performance correlating with high CPU load less strongly than IOR.

Write performance correlates negatively with failed-over OSTs to a much greater degree than read performance. This is likely related to GridRAID as well, because an OSS that is hosting a failed-over partner's OST must calculate twice as much parity on writes. By comparison, read performance is impacted much less significantly because it is not bound by the rate at which the OSS CPUs can calculate parity. If the OSS CPUs were more capable and parity calculations were not the performance-limiting factor on writes during failover, the correlation between write performance and failover state would have been likely to more closely resemble that of read performance.

Read performance shows some sensitivity to job topology while write performance does not. Although the low-diameter dragonfly network on Cray XC systems is designed to make performance independent of topology, Lustre Fine Grained Routing [17] can limit path diversity between compute nodes and LNET gateways. In the case of reads, this limited path diversity can lead to network incasts that result in network congestion near compute nodes and overall performance degradation. In the case of writes, this is not true; compute nodes broadcast their write data to fine-grained routing groups that are topologically scattered, avoiding incasts. Although the exact cause for the positive correlation between read performance and job spread is not clear, the asymmetry between read and write paths and the fewer number of global links surrounding closely packed jobs are likely to contribute to this correlation.

While the correlation between high read performance and large job spread is a novel finding, the other correlations are largely intuitive. That said, this analysis demonstrates how pytokio enables the quantiative, holistic analysis of automated I/O benchmark data and reveals more insight into I/O performance overall than any single component is able to provide. Figure 9 clearly shows that many factors affect I/O performance, but no single metric is a direct indicator. Different I/O patterns and read or write behavior contribute to different sensitivities between performance and the various components of the I/O subsystem.

V. Conclusion

The Total Knowledge of I/O (TOKIO) framework and its reference implementation, pytokio, provide a simple, modular approach to the holistic analysis of I/O performance. *Connector* interfaces retrieve data from the best-in-class I/O monitoring tools already installed on Cray XC and ClusterStor systems. In addition, site-independent abstractions in TOKIO's *tools* interfaces enable the creation of portable analysis tools that can be applied by users and administrators to understand performance at many levels of the I/O subsystem.

pytokio's archival data service extends these capabilities by allowing real-time, operations-focused diagnostic tools such as LMT to serve as sources of long-term, high-resolution time series data. Retaining these time series data in the portable TOKIO Time Series file format enables retrospective performance analyses that uncover a variety of new insights about storage systems. Several analyses have been illustrated with example tools built on pytokio: `darshan_bad_ost` detects straggling Lustre OSTs based on Darshan and LMT data, `darshan_scoreboard` identifies specific users and applications

	IOR Write	HACC Write	IOR Read	HACC Read
Coverage Factor (Bandwidth)	**+0.3954**	**+0.3944**	**+0.4881**	**+0.3987**
Coverage Factor (opens)	**+0.3006**	**+0.3551**	+0.2034	+0.1503
Average MDS CPU Load	-0.2241	-0.2081	+0.0408	+0.0369
Average OSS CPU Load	**+0.5131**	**+0.3266**	+0.1999	+0.0659
Peak MDS CPU Load	-0.2226	-0.2438	-0.0249	+0.0068
Peak OSS CPU Load	+0.1091	-0.0124	-0.0700	-0.0995
OST Fullness	-0.1834	-0.1553	+0.1211	+0.0697
Number of failed-over OSTs	**-0.4304**	**-0.4209**	-0.1064	-0.0677
Average Job Radius	-0.0140	+0.0301	**+0.3510**	**+0.4061**

Fig. 9. Pearson correlation between application I/O performance and other metrics collected by pytokio on NERSC's Cori system. Each value is shaded according to the magnitude of the positive or negative correlation, and values printed in bold are statistically significant (p-value $< 10^{-5}$) whereas other values are not. The IOR benchmarks used 4,096 processes to read and write 16 TiB of data using 4 MiB transfers. The HACC benchmarks used 4,096 processes to read and write 8 TiB of data using $\approx$ 128 MiB transfers. Data reflects daily benchmark results obtained between February 14, 2017 and February 15, 2018.

that are good candidates for migration to burst buffers, and more complex analysis implemented in Jupyter notebooks demonstrate that components of the Cray XC and ClusterStor infrastructure correlate with poor I/O performance.

Because pytokio is BSD-licensed, new connectors to site-specific tools can be developed to suit the needs of different centers as well. Full pytokio source code, complete with a comprehensive suite of tests, documentation, and example analyses are all included in the core package repository. Furthermore, the pytokio archival data service for Cray XC and ClusterStor are also freely available and specifically designed for easy deployment on any Cray systems.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contracts DE-AC02-05CH11231 and DE-AC02-06CH11357 (Project: A Framework for Holistic I/O Workload Characterization, Program manager: Dr. Lucy Nowell). This research used resources and data generated from resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 and the Argonne Leadership Computing Facility, a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

[1] G. K. Lockwood, W. Yoo, S. Byna, N. J. Wright, S. Snyder, K. Harms, Z. Nault, and P. Carns, "UMAMI: A Recipe for Generating Meaningful Metrics through Holistic I/O Performance Analysis," in *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems - PDSW-DISCS '17*. New York, New York, USA: ACM Press, 2017, pp. 55–60. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3149393.3149395

[2] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li, "Fail-slow at scale: Evidence of hardware performance faults in large production systems," pp. 1–14, 2018. [Online]. Available: https://www.usenix.org/conference/fast18/presentation/gunawi

[3] 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*. IEEE, 2009, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5289150

[4] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao, "A Multiplatform Study of I/O Behavior on Petascale Supercomputers," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15, 2015, pp. 33–44. [Online]. Available: http://doi.acm.org/10.1145/2749246.2749269

[5] B. Hadri, S. Kortas, S. Feki, R. Khurram, and G. Newby, "Overview of the KAUST's Cray X40 System - Shaheen II," in *Proceedings of the 2015 Cray User Group*, 2015.

[6] J. P. White, R. Brunner, A. Kot, G. Bauer, B. Bode, J. Enos, W. Kramer, M. Innus, M. D. Jones, R. L. DeLeon, N. Simakov, J. T. Palmer, S. M. Gallo, T. R. Furlani, and M. Showerman, "Challenges of Workload Analysis on Large HPC Systems," in *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact - PEARC17*, vol. Part F1287. New York, New York, USA: ACM Press, 2017, pp. 1–8. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3093338.3093348

[7] T. Butler, "Using Resource Utilization Reporting to Collect DVS Usage Statistics," in *Proceedings of the 2014 Cray User Group*, 2014.

[8] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing Variability in the IO Performance of Petascale Storage Systems," in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, no. November. IEEE, nov 2010, pp. 1–12. [Online]. Available: http://ieeexplore.ieee.org/document/5644883/

[9] J. Keopp and H. Longley, "Cray Lustre File System Monitoring," in *Proceedings of the 2014 Cray User Group*, 2014.

[10] C. Flaskerud, "Project Caribou Streaming Telemetry for Sonexion," *Proceedings of the 2017 Cray User Group*, 2017.

[11] C. Whitney, T. Davis, and E. Bautista, "NERSC Center-wide Data Collect," in *Proceedings of the 2016 Cray User Group*, 2016.

[12] S. Madireddy, P. Balaprakash, P. Carns, R. Latham, R. Ross, S. Snyder, and S. M. Wild, "Analysis and Correlation of Application I/O Performance and System-Wide I/O Activity," in *2017 International Conference on Networking, Architecture, and Storage (NAS)*. IEEE, aug 2017, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/document/8026844/

[13] S. Byna, A. Uselton, D. Knaak, and Y. H. He, "Lessons Learned from a Hero I/O Run on Hopper," in *Proceedings of the 2013 Cray User Group*, Napa, CA, 2013.

[14] "Intel SSD Data Center Tool," 2017. [Online]. Available: https://www.intel.com/content/www/us/en/support/articles/000006289

[15] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka, "Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*. New York,

New York, USA: ACM Press, 2011, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2063384.2063409

[16] H. Bui, H. Finkel, V. Vishwanath, S. Habib, K. Heitmann, J. Leigh, M. Papka, and K. Harms, "Scalable Parallel I/O on a Blue Gene/Q Supercomputer Using Compression, Topology-Aware Data Aggregation, and Subfiling," in *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, feb 2014, pp. 107–111. [Online]. Available: http://ieeexplore.ieee.org/document/6787260/

[17] D. A. Dillow, G. M. Shipman, S. Oral, Z. Zhang, and Y. Kim, "Enhancing I/O throughput via efficient routing and placement for large-scale parallel file systems," in *30th IEEE International Performance Computing and Communications Conference*. IEEE, nov 2011, pp. 1–9. [Online]. Available: http://ieeexplore.ieee.org/document/6108062/

[18] M. Mubarak, P. Carns, J. Jenkins, J. K. Li, N. Jain, S. Snyder, R. Ross, C. D. Carothers, A. Bhatele, and K.-L. Ma, "Quantifying I/O and Communication Traffic Interference on Dragonfly Networks Equipped with Burst Buffers," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, vol. 2017-Septe. IEEE, sep 2017, pp. 204–215. [Online]. Available: http://ieeexplore.ieee.org/document/8048932/

[19] D. M. Jacobsen, J. F. Botts, and Y. H. He, "SLURM. Our Way." in *Proceedings of the 2016 Cray User Group*, 2016.

[20] M. Karo, R. Lagerstrom, M. Kohnke, and C. Albing, "The Application Level Placement Scheduler," in *Proceedings of the 2006 Cray User Group*, 2006.

[21] K. Pedretti, C. Vaughan, R. Barrett, K. Devine, and K. S. Hemmert, "Using the Cray Gemini Performance Counters," in *Proceedings of the 2013 Cray User Group*, Napa, CA, 2013. [Online]. Available: http://www.osti.gov/scitech/biblio/1063364-using-cray-gemini-performance-counters

[22] J. Brandt, E. Froese, A. Gentile, L. Kaplan, B. Allan, and E. Walsh, "Network Performance Counter Monitoring and Analysis on the Cray XC Platform," in *Proceedings of the 2016 Cray User Group*, 2016.

[23] "Aries Network Performance Counters Monitoring Library," United States, 2014. [Online]. Available: http://www.osti.gov/scitech/servlets/purl/1232554

[24] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, "The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, nov 2014, pp. 154–165. [Online]. Available: http://ieeexplore.ieee.org/document/7013000/

[25] T. Groves, Y. Gu, and N. J. Wright, "Understanding Performance Variability on the Aries Dragonfly Network," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, vol. 2017-Septe. IEEE, sep 2017, pp. 809–813. [Online]. Available: http://ieeexplore.ieee.org/document/8049022/

[26] 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 *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*. IEEE, 2015, pp. 1–12.

[27] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright, "Modular hpc I/O characterization with darshan," in *Proceedings of the 5th Workshop on Extreme-Scale Programming Tools*. IEEE Press, 2016, pp. 9–17.

[28] N. A. Simakov, J. P. White, R. L. DeLeon, A. Ghadersohi, T. R. Furlani, M. D. Jones, S. M. Gallo, and A. K. Patra, "Application kernels: HPC resources performance monitoring and variance analysis," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 5238–5260, dec 2015. [Online]. Available: http://doi.wiley.com/10.1002/cpe.3564

[29] D. Henseler, B. Landsteiner, D. Petesch, C. Wright, and N. J. Wright, "Architecture and Design of Cray DataWarp," in *Proceedings of the 2016 Cray User Group*, London, 2016. [Online]. Available: https://cug.org/proceedings/cug2016_proceedings/includes/files/pap105.pdf