Democratizing Parallel Filesystem Monitoring

Richard Todd Evans
Texas Advanced Computing Center, University of Texas at Austin
Austin, USA
rtevans@tacc.utexas.edu

***Abstract*—Parallel filesystems (PFSs) are one of the most critical high-availability components of High Performance Computing (HPC) systems. Most HPC workloads are dependent on the availability of a POSIX compliant parallel filesystem that provides a globally consistent view of data to all compute nodes of a HPC system. Because of this central role, failure or performance degradation events in the PFS can impact every user of a HPC resource. There is typically insufficient information available to users and even many HPC staff to identify the causes of these PFS events, impeding the implementation of timely and targeted remedies to PFS issues. The relevant information is distributed across PFS servers; however, access to these servers is highly restricted due to the sensitive role they play in the operations of a HPC system. Additionally, the information is challenging to aggregate and interpret, relegating diagnosis and treatment of PFS issues to a select few experts with privileged system access.**

To democratize this information, we are developing an open-source and user-facing Parallel FileSystem TRacing and Analysis SErvice (PFSTRASE) that analyzes the requisite data to establish causal relationships between PFS activity and events detrimental to stability and performance. We are implementing the service for the open-source Lustre filesystem, which is the most commonly used PFS at large-scale HPC sites. Server loads for specific PFS I/O operations (IOPs) will be measured and aggregated by the service to automatically estimate an effective load generated by every client, job, and user. The infrastructure provides a real-time, user accessible text-based interface and a publicly accessible web interface displaying both real-time and historical data.

***Index Terms*—file system, monitoring, storage, Lustre**

I. Introduction

We are implementing, validating, and deploying in production High Performance Computing (HPC) systems a Parallel FileSystem Tracing and Analysis SErvice (PFSTRASE) that will enhance the reliability and performance of parallel filesystems (PFSs). As the principle means by which HPC systems serve data and present globally coherent storage to system services, applications, and users, the PFS is the most critical high availability component in many HPC systems. PFS performance and failure events can be unintentionally triggered in a multitude of ways, often in the form of overloading PFS servers with IO requests, and can be frequent occurrences in HPC environments with diverse workloads. The prevalence of such events can also be expected to grow as data-intensive workloads, such as those found in many Machine Learning-based applications, consume a greater proportion of cycles on HPC systems. While performance degradation and failure events within the PFS can negatively impact every user of a HPC system, there are no tools that precisely and accurately portray the current and past condition of the PFS and make this data widely accessible. PFSTRASE fills this gap.

This work was supported by the US National Science Foundation [OAC-1835135].

Much of the reason events are difficult to understand and resolve is that the PFS is a shared resource with loads induced from multiple contemporaneous but not necessarily related IO activities. These activities can cause contention for PFS resources with no straight-forward way to quantify which activities have the greatest impact. The information required to understand and resolve PFS events is typically available but distributed throughout the PFS in the form of pseudo-files hosted locally on PFS servers. The extraction, aggregation, and interpretation of this data requires privileged access to the PFS servers along with significant time and expertise.

With this data, it is possible to identify a PFS event before it has widespread impact on users and mitigate its effects, for example by cancelling an problematic workload (e.g. a job) or subset of workloads executing on a system before they overwhelm PFS resources. That workload may then be inspected and modified to reduce the load it generates on the PFS. Reducing load can be achieved in many ways and varies by workload, but usually involves reducing the rate of IOPs or PFS synchronizations it generates. Negative PFS events can also be avoided by identifying patterns in multiple distinct, non-problematic workloads, that when run simultaneously, may induce pathological loads. Expert administrators may then restrict the sets of problematic workloads that can run simultaneously or ensure they are using distinct filesystem resources. After an event has started impacting users, the signal identifying the source may be lost because the compromised PFS components are no longer servicing job requests in a typical fashion. Historical PFS data is critical in these situations to diagnose the cause.

PFSTRASE provides aggregated PFS data in a location accessible to unprivileged users and staff, along with an estimate of the effective load (`load_eff`) induced by each client, job, and user on the filesystem. Both real-time and historical data are available. In this way the service democratizes PFS monitoring and health maintenance. All users and staff can readily see in real-time what load a workload is generating and it's contribution to the total load on the PFS. Along with post-mortem diagnostics of events, the historical data facilitates the classification of applications according to IO pattern and development of best practices for how the PFS can be configured to support them.

We choose to implement PFSTRASE for the Lustre filesystem. Lustre is an open-source, community supported, scalable PFS well-suited as an HPC storage solution, with clusters of servers acting in parallel to respond to compute node clients' requests for data [1]. As the PFS and compute components of a cluster grows, additional servers can be added to handle new storage or service requests, enabling scalable performance. Lustre is a transparent storage solution for users. The same commands and system calls used to manipulate data on local storage systems are used on Lustre filesystems from the perspective of applications and the user. These combined characteristics of scalable performance and ease of use make Lustre prevalent PFS in HPC. This open-source PFS was used in 70% of the top 100 fastest supercomputing sites as of the writing of this paper [2].

At TACC we've been running PFSTRASE on the Lustre filesystem of the NSF-funded Wrangler cluster for several months [3]. The Wrangler filesystem has 31 Lustre servers serving 99-103 clients each. We've designed PFSTRASE to have a minimal overhead and footprint on the PFS, requiring a constant 0.5% of a single core and 1MB of memory. This setup currently provides real-time I/O data accessible by any Wrangler user.

II. Prior Work

Lustre provides access to much of the data necessary to track and understand IO activity and resource usage in the Linux `procfs` filesystem. `procfs` is a pseudo-filesystem in the Linux root ('/') directory that permits users to query the current state of certain kernel data structures. A series of standardized kernel routines are run every time a pseudo-file in `procfs` is open and read that display this up-to-date kernel data as the pseudo-file's text contents. In the case of Lustre this data consists of monotonically incrementing counters with additional statistics. A typical record in a Lustre maintained pseudo-file contains the following fields for a given IOP: type, count, min value, max value, sum of values, sum of squared values. It is possible to compute the average and standard deviation of the rate for each IOP over specific time ranges from these quantities. Lustre provides related but distinct information in this `procfs` format on the filesystem clients and servers. Current Lustre monitoring tools, including PFSTRASE, collect all of their data from these files, with each focusing on either client-side or server-side collection.

A particularly useful, built-in feature for extracting this data is `Jobstats`, available in Lustre release 2.3+ [4]. This feature tags `procfs` Lustre IOP data with either the process name and user id or the value of an administrator-specified environmental variable (such as JOBID). This feature thus enables the association of IOPs the server is handling to sources of IO activity. The data must still be exported from the server to be accessible, at which point it could be grouped by clients, jobs, or users with the appropriate processing infrastructure.

There are a number of third-party tools that collect and present the data in `procfs` in various ways:

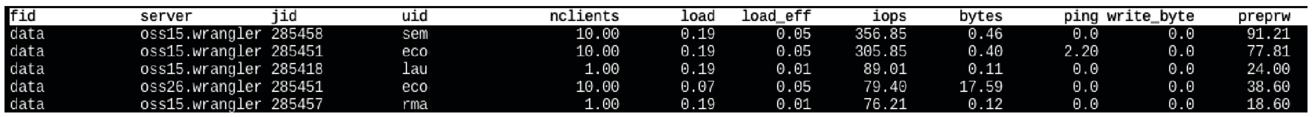
1) **The Lustre Monitoring Tool**, or LMT, monitors server activity [5]. It does not provide information about clients, or have a mechanism for associating activity data with processes or job ids. It stores data in a MySQL database which can be displayed with its `lstat`, `ltop`, or `lwatch` utilities. `lstat` and `ltop` perform functions similar to the Linux utilities `stat` and `top`, displaying data such as PFS disk space and inode capacities and instantaneous server loads, bandwidths, and memory usage. The data can be stored in a MySQL database and available for past and current times. `lwatch` is a GUI interface to this data that is no longer supported.
2) **lltop** is a tool which can take a snapshot of filesystem activity over a configurable time interval [6]. It reports read/write bandwidths and total IOPS aggregated over the whole system and can group the data by job if the source code has been modified and configured to use a particular system's job scheduler. Thus it does provide information connecting overall PFS activity to jobs and users. It also records PFS server loads. It does not compute a `load_eff` and does not break IOPs down by type, making IO patterns impossible to recognize. It also relies on direct `curl` queries to daemons running on the PFS servers and thus is not suitable for unprivileged usage - if large numbers of queries were made simultaneously the performance of the servers could be compromised.
3) **xltop** is a real-time version of `lltop` [7]. It's `top`-like interface heavily influences PFSTRASE's `ncurses`-based text interface.
4) **TACC Stats** collects a wide variety of performance data including client-side Lustre statistics [8]. It associates data with jobids and processes and has a searchable web interface that automatically flags jobs that may require further attention, including a list of recent jobs that have incurred the highest IOP rate. It does not collect PFS server load or storage target data and cannot correlate IOPs to load.
5) **OVIS** project collects a wide variety of performance data including client-side PFS statistics similar to TACC Stats [9], [10]. It is additionally able to combine these statistics with log data in some cases. In June 2019, the ability to collect PFS server data was added. It does not have an automatic mechanism to correlate IOPs to load.

None of these existing tools are capable of quantitatively connecting user and application activities to server loads in a widely accessible user interface.

III. PFSTRASE Design

A. Data Collection and Aggregation

Because PFS servers are so mission critical to HPC systems operations and often have software environments that can be



```
fid          server          jid        uid              nclients      load  load_eff        iops       bytes      ping write_byte      preprw
data         oss15.wrangler 285458      sem                 10.00      0.19      0.05      356.85        0.46       0.0        0.0       91.21
data         oss15.wrangler 285451      eco                 10.00      0.19      0.05      305.85        0.40      2.20        0.0       77.81
data         oss15.wrangler 285418      lau                  1.00      0.19      0.01       89.01        0.11       0.0        0.0       24.00
data         oss26.wrangler 285451      eco                 10.00      0.07      0.05       79.40       17.59       0.0        0.0       38.60
data         oss15.wrangler 285457      rma                  1.00      0.19      0.01       76.21        0.12       0.0        0.0       18.60
```

Fig. 1. The `pfstop` text-interface grouped by job is shown. The filesystem (`fid`), PFS server (`server`), job id (`jid`) and user (`uid`) are shown with the `nclients`, `load`, `load_eff`, total IOPS (`iops`), total MB/s (`bytes`) and other IOP type rates aggregated over all clients associated with that job. The interface can be grouped by filesystem, server, user, jobid, or client dynamically by single keystrokes. The data can also be sorted by `load_eff`, `iops`, or `bytes` by single keystrokes. The interface updates at a configurable interval. The user names have been truncated in this figure to maintain anonymity.

challenging to modify, we've designed the PFSTRASE collection mechanism to be as low overhead and minimally invasive as possible. The collection is performed by either a `init.d` or `systemd` governed daemon running on each server. The daemon is written in C with only two package dependencies external to standard `glibc`-wrapped Linux system calls - `libev` for event synchronization and `json-c` for data organization. It is vital for wide adoption of PFSTRASE that dependency requirements are minimized because PFS servers often run their own vendor-customized Linux OS distributions that have limited software support.

The CPU overhead of the daemon on each PFS server is impacted by the processor architecture, number of clients, and the frequency of sampling. We have measured an average overhead of 0.5% on a single Intel Xeon E5-2680 v3 (Haswell) core at a 5s sampling frequency on a Lustre server with 101 clients. Comparable CPU overhead could be maintained on larger systems by reducing the sampling frequency or newer CPU architectures and/or additional cores. The memory footprint of the daemon is determined by the number of clients mounted on the server and is found to be 1040KB for 100 clients, or about 10KB per client. Thus the memory footprint will be small even for very large systems.

Existing monitoring tools that collect and aggregate data from the PFS servers require direct access to the servers or actively poll the servers. These mechanisms violate the isolation of the servers and can introduce unpredictable load on them based on how many requests are in flight and at what time resolution. Thus such tools are unsuitable for unrestricted usage by users and staff. The PFSTRASE collection daemon sends collected data off-node to a `pfstrase_server` daemon running in a publicly accessible space such as a login node at a constant rate. In contrast to other server collection methods, the collection daemon need never be accessed externally to the PFS server and thus generates a constant and predictable overhead regardless of demand for the data it provides.

The `pfstrase_server` daemon aggregates the data and publicly exposes it in a read-only, memory-mapped file while concurrently sending the data to a database backend. The data it exposes is tagged by filesystem, server name, client name, job id, and user. Much of the data sent from each PFS server is in a per-client format but labeled by what is known internally to Lustre as a *nid* rather than a hostname. This is because the clients' hostnames are unknown to the PFS servers, and thus a map of client nids to client hostnames must be maintained within `pfstrase_server`. The same mapping allows the data to be unambiguously tagged by client, job, and user on a single-tenant system (the development of efficient means for handling shared nodes is still an open topic of research). nid-to-client, client-to-job, and job-to-user mapping data is conveyed to `pfstrase_server` through a TCP socket that accepts json formatted text. The socket-based approach to updating the mapping enables updates to be made in a multitude of ways at whatever frequency is determined to be appropriate. We recommend updating the job mapping every time a job starts and stops in order to have the most up-to-date mapping.

The `pfstrase_server` daemon has very few dependencies, although the publicly accessible node it runs on is typically far more flexible regarding the software that can be installed and supported on it compared to the PFS servers. The server is also written in C in order to maintain low overhead on the public node. For Wrangler's filesystem, comprised of 31 filesystem servers with 99-103 clients, `pfstrase_server` requires 15% of a core on average and roughly 170 MB of memory. This requirement is independent of the number of queries made to the server because the data is exposed in a read-only, memory-mapped file and database backend.

B. Real-time Text Interface

The data in the shared memory-mapped file can be accessed and viewed by any user using `pfstop`, a `top`-like text interface heavily inspired by `xltop`. Unlike `xltop`, `pfstop` does not query the PFS servers directly and can be safely run by any user. It can display IOPS and effective loads (`load_eff`) per server grouped by clients, users, jobs, or servers dynamically. The default view displayed by `pfstop` is grouped by client and thus tagged by client name (`client`), job id (`jid`), and user (`uid`). In this view the total server load (`load`), load per group by which the data is aggregated (`load_eff` per client in this case), IOPS per client (`iops`) and total Megabytes/s per client (`bytes`) are also shown. A detailed view of the data is also available with a key-stroke that breaks down IOPS by type. Views can then be grouped by `jid`, `uid`, and `server`. The data are dynamically sortable by `load_eff`, `iops`, or `bytes` with a keystroke.

A captured image of the `pfstop` detailed view is shown in Fig. 1. The PFS server data in this figure are grouped by `jid`. The tags showing `fid`, `server`, `jid` and `uid` can be seen, while the data from the clients participating in each job have been aggregated. The `load`, `load_eff`, `iops`, `bytes`, and IOPS by type (`ping`, `write_byte`, `preprw`) per filesystem, server, job and user are shown. While all IOPS

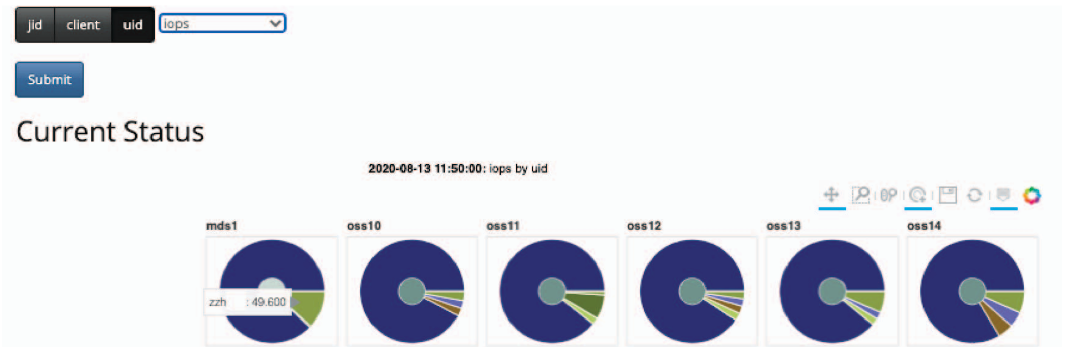


Fig. 2. The PFSTRASE web interface with charts displaying the `iops` for each server grouped by `uid` at an instant in time is shown. Different colors and widths in the annular charts indicate each user and the fraction of total `iops` than can be attributed to each user. The colors in the center of the annular charts indicate the total `iops` on the server. Data can be grouped by `jid`, `uid`, or `client` using the buttons in the upper left of the webpage. IOP type to display can be chosen from the dropdown menu to the right of those buttons. Data is currently available for any time PFSTRASE was running.

types are displayed in the text interface's detailed view, some are omitted in this figure for readability.

`load` and `load_eff` are currently expressed as the number of cores being used at any given time, similar to what is shown in a command such as `uptime`. This may be expressed as a percent CPU utilization in the future based on user feedback. Additional information collected and aggregated by the PFSTRASE server, but not currently displayed in `pfstop`, includes memory usage per server, 5, 10, and 15 minute load averages, and PFS disk usage information including inode and disk space usage. The incorporation of these quantities and metrics derived from them into the `pfstop` interface is ongoing work. There are also other statistics we are not currently collecting but may in the future such as file locks per second or statistics concerning data transfer sizes of individual read and write operations.

The most useful definition of `load_eff` is also still under investigation. Currently it is computed as an equally-weighted, linear combination of all IOPS normalized by the total number of IOPS. We believe this approach to be simplistic, as certain IOPS may contribute more to the load than others due to a variety of factors including variation in CPU resources required to execute an IOP and disc synchronization requirements. While we don't expect to achieve perfect correlation of IOPS to load we expect some linear combination of IOPS to be meaningful, self-consistent, and provide actionable insight.

We will also be implementing a mechanism to anonymize the data presented to unprivileged users in `pfstop` for HPC sites where this capability is desired or necessary. This mechanism will show a user their own PFS activity while obscuring the labeling of other users' activities. Determining the privilege-level of a user can be readily determined, for example, using Linux group memberships.

C. Real-time and Historical Web and Database Interface

`pfstop` is only capable of displaying real-time data. A database backend and website is under development to enable the exploration of historical trends and enable off-line analyses such as IO pattern characterization of applications. The `pfstrase_server` sends the same data available to `pfstop` from the public host (e.g. login node) to a local or remote database at a configurable interval. We have chosen to use the TimeScaleDB extension of PostgresSQL for the database backend [11], [12]. This database has the advantage that it is highly optimized to work with time-indexed data while also presenting a conventional SQL interface. Database calls are made using the `pyscopg2` Python Package to PostgreSQL, while the Django web application framework serves that data to a website [13], [14]. Interactive plots are dynamically generated using the Python Bokeh library [15].

An instance of the web interface showing `iops` on a subset of filesystem servers at an instant in time is shown in Fig. 2. The annular chart for each server is broken down by `uid`, where the color indicates a particular `uid`, and the color's width indicates the fraction of `iops` that `uid` is generating. The colors in the center of the charts indicate the total `iops` for each server. `uids` and `iops` can be seen by hovering over the corresponding sections of the charts. The data can be grouped by `jid`, `client`, or `uid` by clicking the corresponding button in the upper left of the webpage. The IOP type to display in the charts can be chosen from the drop down menu to the right of those buttons. Data for any time can be displayed. A heat map displaying the total for each IOP type on each server in time is also currently available but not shown here. The colors in the heatmap correspond to the colors in the center of each chart.

In future work, we will include memory, inode, and disc usage per server in this web site. We are also exploring other visualizations and analyses to help identify problematic IO patterns. This website will be made available to all users with data anonymized in a configurable manner.

IV. Summary

We have introduced our PFSTRASE project, the current status of its development, and the next steps of the project. A prototype of the infrastructure has been implemented, deployed, and is successfully running on one of TACC's production filesystems.

References

[1] OpenSFS and EOFS, "Lustre Wiki," wiki.lustre.org, accessed: July 2020.

[2] R. Henwood, "Why Use Lustre," https://wiki.whamcloud.com/display/PUB/Why+Use+Lustre, accessed: July 2020.

[3] Texas Advanced Computing Center, The University of Texas at Austin, "Wrangler Home Page," https://www.tacc.utexas.edu/systems/wrangler, accessed: July 2020.

[4] Lustre, "Lustre Software Release 2.x Operations Manual," http://doc.lustre.org/lustre_manual.xhtml, accessed: July 2020.

[5] LMT, "LMT Home Page," https://github.com/chaos/lmt/wiki [http://code.google.com/p/lmt/], accessed: July 2020.

[6] J. Hammond, "lltop," https://github.com/jhammond/lltop, accessed: July 2020.

[7] J. Hammond, "xltop," https://github.com/jhammond/xltop, accessed: July 2020.

[8] R. Evans and J. Hammond, "TACC Stats," https://github.com/TACC/tacc_stats, accessed: July 2020.

[9] J. M. Brandt, B. J. Debusschere, A. C. Gentile, J. R. Mayo, P. P. Pébay, D. Thompson, and M. H. Wong, "Ovis-2: A robust distributed architecture for scalable ras," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–8.

[10] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. IEEE Press, 2014, pp. 154–165. [Online]. Available: https://doi-org.ezproxy.lib.utexas.edu/10.1109/SC.2014.18

[11] Timescale, "Timescale Home Page," https://www.timescale.com, accessed: July 2020.

[12] The PostgreSQL Global Development Group, "PostgreSQL Home Page," https://www.postgresql.org, accessed: July 2020.

[13] F. D. Gregorio and D. Varrazzo, "Pyscopg - PostgreSQL database adapter for Python," https://www.psycopg.org/docs, accessed: July 2020.

[14] Django, "Django Home Page," https://www.djangoproject.com, accessed: July 2020.

[15] NumFOCUS, "Bokeh Home Page," https://docs.bokeh.org/en/latest/index.html, accessed: July 2020.