Real-Time I/O-Monitoring of HPC Applications with SIOX, Elasticsearch, Grafana and FUSE

Eugen Betke, Julian Kunkel

Deutsches Klimarechenzentrum, Hamburg 20146

Abstract. The starting point for our work was a demand for an overview of application's I/O behavior, that provides information about the usage of our HPC "Mistral". We suspect that some applications are running using inefficient I/O patterns, and probably, are wasting a significant amount of machine hours. To tackle the problem, we focus on detection of poor I/O performance, identification of these applications, and description of I/O behavior.

Instead of gathering I/O statistics from global system variables, like many other monitoring tools do, in our approach statistics come directly from I/O interfaces POSIX, MPI, HDF5 and NetCDF. For interception of I/O calls we use an instrumentation library that is dynamically linked with `LD_PRELOAD` at program startup.

The HPC on-line monitoring framework is built on top of open source software: Grafana, SIOX, Elasticsearch and FUSE. This framework collects I/O statistics from applications and mount points. The latter is used for non-intrusive monitoring of virtual memory allocated with mmap(), i.e., no code adaption is necessary. The framework is evaluated showing its effectiveness and critically discussed.

1 Introduction

The moderate progress of network and storage technologies, and comparatively fast increase of computational power over the last decades had a negative impact on the balance of many current HPC systems. Especially, increasing number of cores per node facilitates higher data processing rates that often exceed the capabilities of network or storage. In data-intensive research fields, like climate science, where data volumes are large and steadily increasing, I/O became an annoying bottleneck. Nowadays, the imbalance between computational power, network bandwidth and storage performance makes us re-think the usage of I/O resources. Researchers in the I/O field propose various directions for new HPC architectures (e.g. burst buffer), hardware solutions (e.g. SSDs), and non-intrusive software solutions (e.g. compression), that solve partially the problem. But in many cases, poor I/O performance is a result of inefficient I/O access patterns of applications. These applications could probably be fixed, but the difficulty is to detect these applications and to describe to what extend they are affected by the problem. An insight of how application uses the underlying I/O interface could be of great help.

The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-67630-2_15

In data-intensive science, MPI-IO [11] is one of the most frequently used high I/O level interfaces. It was designed as a general purpose I/O interface, to facilitates parallel low level access to files. Most implementations contain a number of optimizations like Two-Phase I/O, Collective I/O, and Data Sieving, which purpose is to create from several I/O access, large and contiguous accesses, or other techniques like Non-Blocking I/O, which handle data asynchronously. HDF5 and NetCDF are high level, portable file formats, data models and libraries specialized to store large datasets. They also provide a set of tools for exploration and manipulation of data. The data size is not limited by the specifications (but limited by current implementations to 32 EiB). They run on a wide range of computational platforms, from laptops to large scale HPCs. Although, POSIX wasn't designed with parallel file access in mind and has some limitations when accessing shared file regions by multiple processes, it still remains one of the most important interfaces, especially because most of the back-ends of the high-level libraries use it to write data to storage.

Our long-term goals are the detection of poor performance and identification of problematic HPC applications. This work is an important step in this direction. Here, we present a user-friendly way for on-line visualization and description I/O of behavior of HPC applications. For that purpose, we build a monitoring framework on top of open source software: SIOX, Elasticsearch, Grafana, and FUSE. One of its features (and also our main contribution) is the novel approach for a non-intrusive instrumentation of virtual memory allocated by `mmap()` operation.

This paper has the following structure. Section 2 presents related work. Section 3 introduces the framework components. In Section 4 we show the design of our framework. In Section 5 we describe our experiments and evaluate the results in Section 6.

2 Related Work

In this section we introduce three monitoring tools: Darshan, Vampir, and SIOX. Unfortunately, it doesn't contain any related work about monitoring of `mmap()`, for the simple reason: even after a careful research, we didn't found any serios publication. This makes us think, our approach is a novelty.

Darshan Darshan [1,6] is an analysis tool for characterization of I/O behavior of HPC systems. It was developed to capture accurate pictures of application behavior and properties, e.g., I/O access pattern on a file. For instrumentation Darshan uses a number of different wrappers. They intercept I/O operations of all files used by the application and produce output for each file. Instead of storing all the data in a trace file, like conventional tools do, Darshan creates statistics, that are reduced, compressed, and represented in a compact form. After analysis, the data is written to a log file. The data in these files describes the behavior of the entire application. This approach has a negligible overhead and requires a limited amount of memory.

For analysis of log file Darshan provides a number of command line tools. One of them is “darshan-job-summary”. As the name indicates, it creates a summary of a log file. The Darshan instrumentation support different I/O interfaces. They have full support for the POSIX and MPI-IO interfaces. HDF5 and PNetCDF are supported partially. Darshan can be utilized in a broad spectrum of tasks, beginning with optimization of applications and ending with analysis of I/O behavior of entire HPC systems. The lightweight and efficient design of Darshan makes it possible to use it for load characterization on large systems, even on productive systems.

Darshan extended tracing (DXT) allows a more detailed profiling of I/O software stack. It contains two main components, the logging and the analysis tool. The former creates trace files while application runs and the latter can be used for the offline analysis and visualization of the data. The features work without any modification or recompilation of applications, provide a number of usefull statistics and work with a negligible overhead.

Vampir Vampir [3,9] is a graphical tool for performance analysis of parallel systems. It supports off-line analysis of parallel software (MPI, OpenMP, multi-threaded) and hardware accelerated (CUDA and OpenCL) applications. The analysis engine allows a scalable and efficient processing of large amounts of data. Vampir uses the infrastructure of Score-P [2] for instrumenting of applications. Score-P stores events in a file, that can be analysed by Vampir and converted to different views, e.g., events can be presented on a time-axis, or compressed to different statistics. Some views have elaborate filters and zoom functions, that can provide an overview, but can also show details. Effective usage of Vampir requires a deep understanding of parallel programming. Although, the program makes it possible to capture and to analyse sequences of POSIX I/O operations, it gives little or no information about the origin, or evaluation of I/O. The field of application of Vampir is restricted through the missing support of on-line analysis.

SIOX SIOX [10] is a highly modular instrumentation, analysis and profiling framework. It contains an instrumentation tool *“siox-inst”*, a trace reader *“siox-trace-reader”*, and a set of plug-ins and wrappers.

Currently, there are wrappers for MPI, POSIX, NetCDF and HDF5 interfaces. They contain re-implementations of the original I/O functions. Inside a reimplemented function is a call to the original function or syscall, and instrumentation code, that generates an activity after each execution. Activities in SIOX are structures that contain various information about the calls. The wrappers can be dynamically linked to an application by using the LD_PRELOAD feature.

Extreme modular design is one of the key features of SIOX. The tools *siox-inst* and *siox-trace-reader* can be considered as pure plug-in infrastructures. In other words, there is no functionality inside until some plug-ins and wrappers are loaded. Usage of different sets of plug-ins and wrappers may result in “new” tools, that fits exactly the problem. There is no restriction on the number of wrappers and plug-ins can be loaded simultaneously, so that the functionality of SIOX can be easily extended, e.g., to perform complex tasks.

Other two important features of SIOX are the support of on-line and off-line analysis. On-line analysis can be done by *siox-inst*, by collecting activities from the wrappers and forwarding them to the registered plug-ins. Off-line analysis is based on both tools. In the first step *siox-inst* stores the activities in a file, by using the *activity-writer-plugin*. In the second step *siox-trace-reader* reads the activities from the file and forwards them to the loaded plug-ins. (The second step is the actual off-line analysis.)

Most of the SIOX plug-ins are using plug-in interfaces that are supported by *siox-inst* and *siox-trace-reader*, and consequentially these plug-ins can be used by both tools.

3 Components

This section contains a short description of components used in our online monitoring framework.

3.1 Elasticsearch

Elasticsearch [7] is a distributed, scalable, real-time search and analytics engine, published under the Apache 2 license. It is built on top of the Apache Lucene full-text search-engine library. The complexity of the library is hidden behind a RESTful API. The indexing of all fields allow very fast lookups, and makes it real-time capable. The library can be used on a broad range of devices. It is suitable for a single machine as well as for large-scaled super computers.

3.2 Grafana

Grafana [5] is a feature-rich, interactive visualization and dashboard software. For visualization, it provides different widgets, e.g., time series, tables, text fields for single metrics. It also supports a many data sources, e.g., Graphite, Elasticsearch, InfluxDB, OpenTSDB.

Especially remarkable is the wide range of available features. Quick range selection makes the navigation inside a time series precise and easy. It has zoom and auto refreshing functions, and a set of predefined, often used ranges. In most cases, a few mouse clicks are sufficient to visualize required range of data. Templating is one of the most powerful features of Grafana. Templates define arrays, which are dynamically filled with values, depending on the current data or state of Grafana. These array can be used on different places, e.g., in metric queries, panel titles, automatic dashboard generation. The latter means, that it is possible to generate for each value in the array a graph or other widget, e.g., suppose an array holds a list of node names, and performance graph was defined, then this graph can be created for each node name automatically. When

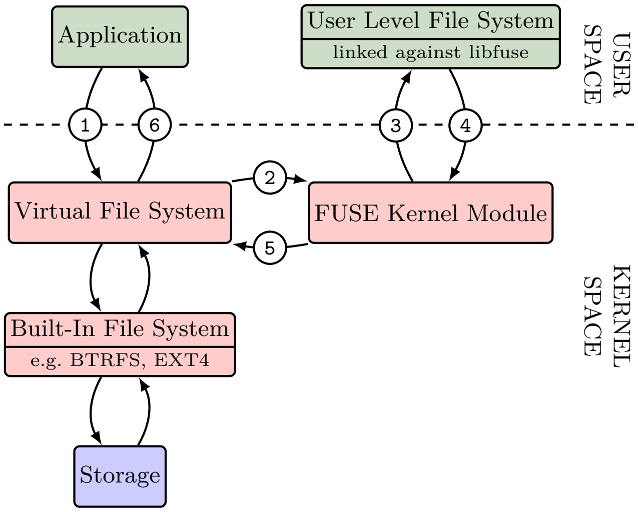


Fig. 1: FUSE I/O path.

a new node name appears in the array, the corresponding graph is automatically generated. Grafana support annotations. This feature is useful, when some event should be shown in the graph.

Grafana dashboards can be easily shared via URL. The URL is automatically updated on dashboard changes.

3.3 IOFS: a FUSE-based file system

FUSE (Filesystem in Userspace) [8] is a kernel interface for file system drivers, which can be run in non-privileged mode. The FUSE project provides an implementation of this interface. It consists of two key components, the *fuse* kernel module and *libfuse* library. The latter can be linked against a program to establish a connection to the *fuse* kernel module.

Virtual file system (VFS) is an abstraction that hides real file systems. Applications see VFS only, and communicate with file systems only over VFS. Figure 1 shows how I/O requests to a FUSE file system are processed. VFS and FUSE modules act like switches. At VFS arriving I/O requests, which are addressed to a FUSE file system, are routed to the FUSE kernel module and then to the destination. The replies take the reverse route. How user level file system stores and retrieves the data, is left to the implementation.

IOFS is a user level file system that implements the FUSE interface. It was developed to be used as an auxiliary tool for instrumentation of mount points. IOFS mounts a folder from an existing file system on some mount point. It runs completelly in user space and behaves like an ordinary application when started in foreground, i.e., SIOX wrappers can be dynamically linked using `LD_PRELOAD`. One important feature of IOFS is that it has neither caches nor buffers, i.e., all

I/O request are forwarded to VFS without delay. Furthermore, the implementation doesn't call mmap() function. All this makes it to a perfect candidate for instrumentation with SIOX.

3.4 SIOX + On-line Monitoring Plug-in

The SIOX-On-line-Monitoring plug-in captures data from SIOX activities, SLURM and system environment variables, and uses system clock for time stamp. The system clock is supposed to be synchronized. For performance reasons we don't collect all the data. Instead, in a defined time interval only relevant values are aggregated to statistics, and are sent to Elasticsearch in JSON format using the REST-API. This approach ensures a low data transfer rate and makes it independent from access pattern of applications. The data transfer rate increases only with number of files used in the application.

Statistics A data point or statistics (Table 1) consist of metrics, tags, and a time stamp. The distinction is based on usage of the data in Grafana.

The current set of metrics consists of number of bytes (*_bytes), duration (*_duration), number of calls (*_calls), and number of bytes per call (*_bytes_per_call) for read and write operations. Number of bytes and duration are obtained directly from SIOX activities. Number of calls is a counter of occurred activities in a time interval. Derived metrics are calculated from more than two metrics. They must be created inside the plug-in, because Elasticsearch doesn't support arithmetic operations on data, and Grafana is limited to scaling with a constant value, e.g., write_bytes_per_call is derived from basic metrics.

Tags provide additional information to the metrics. The tags username, hostname, procid, jobid are obtained directly from SLURM environment variables. hostname is provided by the system. filename and access (access type: write, read, ...) are provided by SIOX activities. layer is a user defined tag and can take any value, e.g., we use different values for monitoring applications and mount points.

timestamp is playing a special role in data series. Currently, milliseconds are the highest possible resolution supported by Elasticsearch.

Categories of operations Some I/O interfaces contain different functions that do similar operations, e.g., POSIX offers writev(), write(), pwrite(), pwrite64(), puts(), and other functions, which can do a write operation. For our purposes it's not necessary to know function names, but operation names is fully sufficient. At the moment our prototype supports write and read operations. Further operations can be added with a minimal effort.

Visualization For visualization of I/O behavior we use several Grafana dashboards. Generally, metrics are used on the y-axis and time stamp on the x-axis.

Name	Type	Value
write_duration	metric (basic)	time spent for writing
write_bytes	metric (basic)	bytes written
write_calls	metric (basic)	number of I/O operations
write_bytes_per_call	metric (derived)	write_bytes, write_calls
read_duration	metric (basic)	time spent for reading
read_bytes	metric (basic)	bytes read
read_calls	metric (basic)	number of I/O operations
read_bytes_per_call	metric (derived)	read_bytes, read_calls
filename	tag	filename
access	tag	access type (write, read, . . .)
username	tag	SLURM_USER
hostname	tag	HOSTNAME
procid	tag	SLURM_PROCID
jobid	tag	SLURM_JOBID
layer	tag	user defined
timestamp	date	system clock

Table 1: Statistics

The tags are used for filtering of data, e.g., we can choose a filename to show I/O behavior of a specific file. Several tags can be used simultaneously.

4 Monitoring Framework Design

On a properly configured system monitoring is enabled by starting an application with a SIOX wrapper. Virtually, one can think of SIOX as a function that takes an executable as argument. For this we use the notation: `SIOX(<exec>)`.

4.1 On-line monitoring of applications

`SIOX(Application)` in Figure 2 represents the instrumentation of an application. SIOX creates activities from I/O calls and builds an activity stream to the `Online-Monitoring-Plugin`. The plug-in aggregates the activities to statistics and sends them to Elasticsearch. Grafana uses data from Eleasticsearch for visualization.

Monitoring of `Virtual Memory` is not possible in this approach, because this component runs in kernel space, but it can produce application related I/O, e.g., when the application maps a part of a file to virtual memory by using the `mmap()` function and then accesses the content of the file through the memory.

4.2 On-line monitoring of mount points

The basic idea of this approach is to move I/O request produced by virtual memory from kernel space to user space. This can be easily achieved with a FUSE-based file system. In the first step IOFS mounts a folder, that contains required files, to some mount point. In the second step, we make sure, our application works on this directory. When the application applies the `mmap()` function to some file on this mount point, all I/O requests from virtual memory to this file will be forwarded to IOFS.

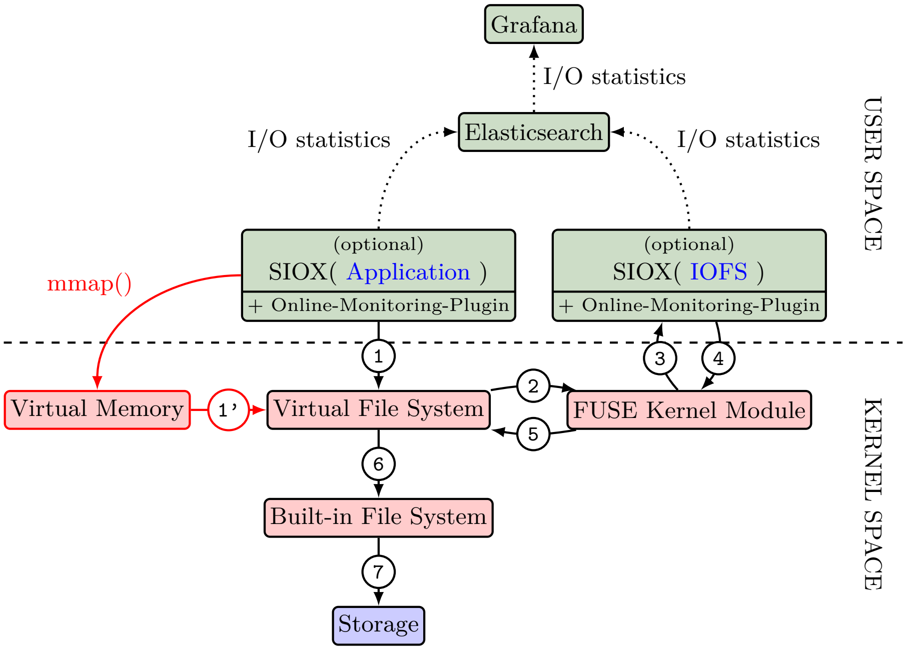


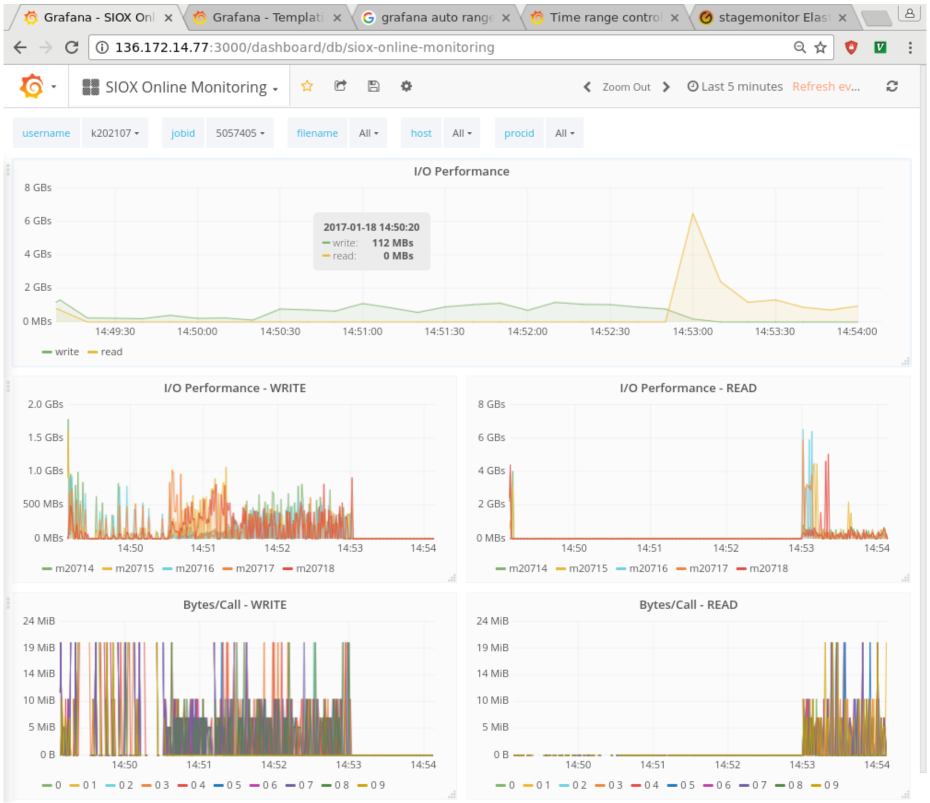
Fig. 2: Extended on-line monitoring

The monitoring works in the same way as `SIOX(Application)`, but this time we use `SIOX(IOFS)`.

Now, the monitoring is closer to the system than to the application. It provides information about real communication that takes place on a specific mount point. That means, in this way we can observe some thing that happens on system level, e.g., optimizations that are done by the operation system; changed access granularities or burst writes.

A nice side effect of this approach is the indirect instrumentation of POSIX `mmap` operations. Remember, that the direct instrumentation was a problem, because memory allocated by mmap is accessed directly without a syscall, and therefore, couldn't be instrumented by SIOX. In IOFS such accesses are transformed to common read/write POSIX operations, which in turn are supported by SIOX.

On-line monitoring of applications using this approach is possible only to a limited extent. Firstly, the I/O requests on this mount point cannot be tracked back to the application. There is an information loss. We must made an implicit assumption, that we know which application works with the data and that all I/O requests belong to the same application. Secondly, on each node we can create only one mount point with the same name. This can be a disadvantage for multi-threaded applications, because there is no way to track I/O requests back the the threads. This information will also be lost, and there is no easy



136.172.14.77:3000/dashboard/db/siox-online-monitoring

username k202107 jobid 5057405 filename All host All procid All

2017-01-18 14:50:20
write: 112 MBs
read: 0 MBs

Fig. 3: Screenshot of the on-line monitoring dashboard

way to solve this issue. Thirdly, not all I/O operations are directed to the mount point. Typically, there is a number of files that are accessed outside the mount point. This information will also not be registered.

5 Experiments

In the first experiment we measure how many metrics we can send to Elasticsearch. For that purpose Elasticsearch was installed on a system equipped with Intel i7-6700 CPU (Skylake) with 4 cores @ 3.40GHz and 16 GB DDR3 RAM. The metrics were generated on Mistral by 10 nodes and 20 processes per nodes and sent over 1 GiB ethernet to Elasticsearch in JSON containers each containing 100 metrics.

In the second experiment, the measurement of overhead, we run a series of experiments on system equipped with Intel Core i5-660 (Clarkdale), 4M Cache, 3.33 GHz, 12 GB DDR3 RAM, 2 TB HDD (test disk), 1 GB/s network, 500 GB HDD (OS disk). The experiments were conducted with IOR and IOZone benchmarks. IOR was used to produce independent streams of POSIX operation calls and IOZone was started in mmap-mode. We varied the number of processes (NP) and request size and run the experiments several times for all four configurations.

The mean values of I/O performance of benchmarks without monitoring (NMON) were used as reference values. The same benchmarks were run with monitoring of application (APPIO), mount point (IOFS), and with both (BOTH). The experiments were repeated 10 times and the results are shown in Figures 4 and 5.

6 Evaluation

The primary goal of the framework is to provide enough information to identify inefficient applications. Additionally, from the user perspective, the framework must be convenient to use and from the perspective of HPC systems, it must be scalable and perform well with low overhead. In this section we investigate both aspects.

6.1 Performance

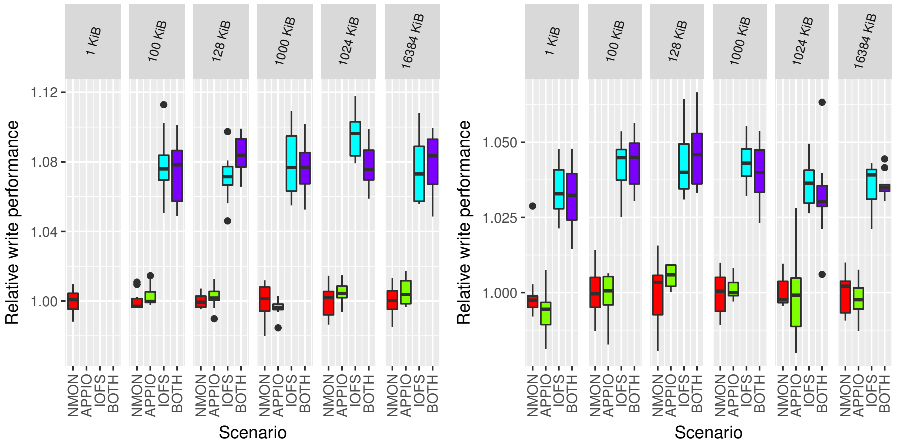
In our test environment, Elasticsearch processes about 750,000 metrics per second, while the aggregated transfer rate stays below 10 MiB/s. Since our current plug-in implementation uses 16 metrics, this is sufficient to capture I/O statistics from about 46000 processes, simultaneously. The limiting factor is the CPU utilization induced by Elasticsearch, but this bottleneck can be relaxed by scaling up/out Elasticsearch.

6.2 Overhead

The Figures 4 and 5 show relative overhead of monitoring (APPIO, IOFS, BOTH). To enhance comparability, it also contains benchmark results of test runs without monitoring (NMON). In these figures we can observe a negligible overhead for file I/O. For mmap I/O there is also a negligible overhead, but only for read operations. For write operations, the overhead is around 8% for file I/O and 3% for mmap I/O. In our case this was mostly the case. The outliers in Figure 4a can be explained by a large number of function calls. For the outliers in Figure 5b we have no explanation at the moment.

6.3 User experience

We paid particular attention to user experience, because we are convinced, that software which is difficult to use or that doesn't work properly finds little or no acceptance by users. Although, the most parts of the framework meet our expectations, after a closer look we found some limitations. The points below refer to Grafana 4.2.0.



(a) IOR

(b) IOZone

Outliers for 1 KiB

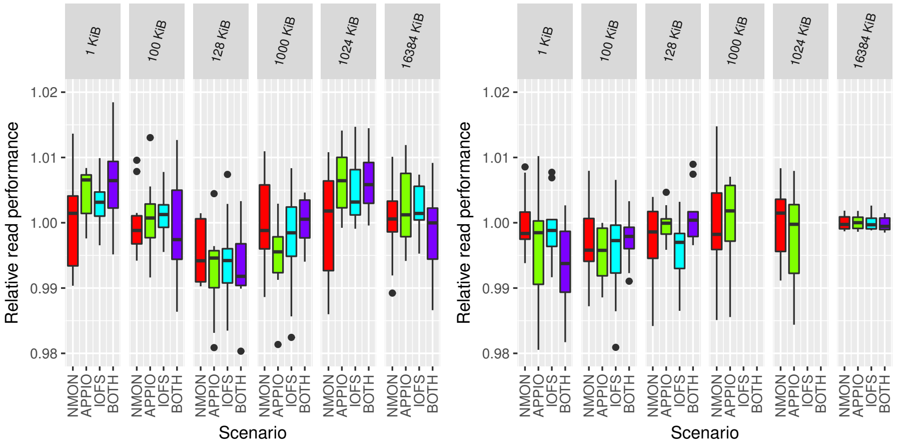
Case	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1	1.125	1.133	1.139	1.137	1.142	1.147
2	3.506	3.537	3.580	3.590	3.652	3.662
3	4.738	4.888	5.120	5.078	5.257	5.384

Fig. 4: Write overhead. (NMON: no monitoring; APPIO: file I/O; IOFS: mmap I/O; BOTH: file and mmap I/O)

Firstly, the update of information inside the drop-down lists is not sophisticated. Grafana provides two options: *update on dashboard load* and *update on time range change*. Under some conditions the drop-down list are not updated when new values are available in the database. Depending on the configuration, there are two workarounds to get the jobid appear. It can be done by leaving and entering the dashboard or by changing the time range. Both options are non-intuitive for users. In general, if entries doesn't appear in the drop-down lists, they can be entered manually, but it is also inconvenient, especially when several template values must be updated. A solution could be a third option (which is not implemented), that updates the information in the drop-down list automatically on each mouse-click.

Secondly, the zoom function doesn't provide an auto range function which shows all data for current template values or allows jumping to the beginning of the data.

Thirdly, neither Grafana nor Elasticsearch provide possibilities to compute new metrics from existing ones. This could be a problem for advanced users who need derived metrics. At the moment, derived metrics must be computed by SIOX and sent to Grafana, which means additional network overhead and more storage space consumption.



(a) IOR

(b) IOZone

Outliers for 1000 KiB

Case	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
2	1.224	1.241	1.261	1.257	1.271	1.288
3	1.250	1.257	1.260	1.265	1.267	1.300

Outliers for 1024 KiB

Case	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
2	1.260	1.282	1.288	1.293	1.308	1.342
3	1.270	1.275	1.288	1.287	1.298	1.304

Fig. 5: Read overhead. (NMON: no monitoring; APPIO: file I/O; IOFS: mmap I/O; BOTH: file and mmap I/O)

7 Summary

The paper proposes an on-line monitoring framework for HPC systems, which can help to detect and to describe the I/O behavior of parallel applications. It is built on top of open source software: the instrumentation framework “SIOX”, database “Elasticsearch”, visualization tool Grafana and a FUSE-based file system “IOFS”.

SIOX is able to intercept the I/O requests from applications, and mount point, when used with IOFS. The latter method can be used as a novel approach for indirect interception of mmap I/O.

The performance of Elasticsearch on an office computer is sufficient to gather 750000 metrics per second. Since Elasticsearch is a distributed database this value can be easily increased. The preliminary experiments on an office computer showed that the overhead for file I/O is negligible in most cases. For mmap I/O the overhead is around 8% for file I/O and 3% for mmap I/O. We intend to run extended experiments on Mistral [4] as soon as the FUSE module is available, paying particular attention to the outliers.

References

1. Darshan hpc i/o characterization tool. `http://www.mcs.anl.gov/research/projects/darshan/`, 2015.
2. Score-p. `http://www.vi-hps.org/projects/score-p/`, 2015.
3. Vampir. `http://www.paratools.com/Vampir`, 2015.
4. Mistral. `https://www.dkrz.de/Nutzerportal-en/doku/mistral`, 2016.
5. Beautiful metric & analytic dashboards. `http://grafana.org/`, 2017.
6. Philip Carns. Darshan. In *High Performance Parallel I/O*, Computational Science Series, pages 309–315. Chapman & Hall/CRC, 2015.
7. Clinton Gormley and Zachary Tong. *Elasticsearch: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2015.
8. Brijender Kahanwal. File system design approaches. *CoRR*, abs/1403.5976, 2014.
9. Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. *Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir*, pages 79–91. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
10. Julian Kunkel, Michaela Zimmer, Nathanael Hübbe, Alvaro Aguilera, Holger Mickler, Xuan Wang, Andriy Chut, Thomas Bönisch, Jakob Lüttgau, Roman Michel, and Johann Weging. The siox architecture – coupling automatic monitoring and optimization of parallel i/o. 2014.
11. Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing mpi-io portably and with high performance. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, IOPADS '99, pages 23–32, New York, NY, USA, 1999. ACM.