University of Reading

Footprinting parallel I/O – machine learning to classify application's I/O behavior

Conference or Workshop Item

Accepted Version

It is advisable to refer to the publisher's version if you intend to cite from the work. See Guidance on citing.

Footprinting Parallel I/O – Machine Learning to Classify Applications I/O Behavior

Eugen Betke[1] and Julian Kunkel[2]

[1] DKRZ – `betke@dkrz.de`
[2] University of Reading–`j.m.kunkel@reading.ac.uk`

Abstract. It is not uncommon to run tens of thousands of parallel jobs on large HPC systems. The amount of data collected by monitoring systems on such systems is immense. Checking each job individually by hand, e.g., for identification of high workloads or detection of anomalies, is infeasible. Therefore, we are looking for an automated approach.

Many automated approaches are looking at job statistics over the entire job run time. Information about different activities during the job execution is lost. In our work, we partition the collected monitoring data for each job into a sequence of smaller windows for which we analyze the I/O behavior. Then, we convert the sequence to a footprint vector, where each element shows how often this behavior occurs. After that, the footprint dataset is classified to identify applications with similar I/O behavior. The classes are interpretated by a human which is the only non-automatic step in the workflow.

The contribution of this paper is a data reduction technique for monitoring data and an automated job classification method.

1 Introduction

Modern HPC systems involve a complex interaction between many hardware and software components. To get information about the system health and utilization powerful monitoring systems are deployed and constantly improved. Basically, monitoring systems share the same principle: (1) Collectors capture utilization metrics from HPC components in a fixed interval. (2) Analysis tools access data for visualization and statistic computation.

They may utilize various visualization and statistical tools that process monitoring data, and aggregate it to an appropriate and understandable representation. These systems allow a detailed look at the current system state. Advanced features allow archiving of job data, set alerts and expose the data to users.

Daily, tens of thousands of jobs can be executed on modern HPC systems, producing an immense amount of monitoring data. These data contain a lot of useful and interesting information but it is a challenging task to evaluate all these data with human power only. Therefore, these systems require new tools for automatic evaluation.

In respect to I/O, storage is known to be not very well utilized on many systems. As data is crucial for operation and storage an expensive part of HPC

systems, data centers are interested in its efficient usage. The good news is that many workflows have often hidden optimization potential there, that is waiting to be discovered and utilized. For example, a workload manager can launch non-interfering applications together (e.g., I/O intensive and CPU-intensive applications), troublemakers can be isolated and problems can be communicated to users. Applications, with bad I/O behavior can be identified and optimized. A more predictable I/O could make application runtime predictable and provide a better user experience. A deep insight in the current systems and understanding of the problem could be helpful for designing future system.

This paper is structured as follows. We start with the related work in Section 2. Then, in Section 3 we introduce the DKRZ monitoring systems and explain how I/O metrics are captured by the collectors. In Section 4 we describe the data reduction and the machine learning approaches and do an experiment in Sections 5 and 6. Finally, we finalize our paper with a summary in Section 8.

2   Related Work

Advanced HPC monitoring systems collect data from different sources and convert them into an understandable representation. Then, they compute statistics, correlate data from multiple system levels and visualize them. This allows a deep understanding of the system, application I/O profiling and anomaly detection. Often, a portion of this data, which doesn't require a deep domain knowledge, is exposed to users. Many monitoring systems work in this way, e.g., Beacon [11], XDMoD [5], or the DKRZ monitoring system [1]. Many more tools to capture and analyse I/O behavior are described in [3].

An automated workload characterization on storage was investigated in [2]. This approach includes a monitoring infrastructure, that collects storage-specific metrics from arriving I/O requests. Based on the captured data, a workload model, that represents the main aspects of I/O-intensive applications, is created. Experiments with predictive models for I/O performance and variability are conducted in [6] [10] [4].

A machine learning approach for anomaly detection was investigated by Tuncer et al. in [8]. This approach targets two aspects. First, data reduction by mapping large raw time series to a few statistics. Second, automatic anomaly detection and classification with machine learning algorithms. With appropriate data, a model can be adapted for detection of different anomaly types, e.g., hardware issues, memory leaks, system health status. In the experiments on an HPC cluster and a public cloud, a trained machine learning model shows a high accuracy (F-score higher than 0.97).

In [7] B. Seo at al. are classifying I/O traces with a data-mining approach to build a software for flash translation layer (FTL) on SSDs, with intention to improve I/O performance and to increases hardware lifetime. The traces consist of sequences of I/O requests, where each I/O request contains the logical block address, access type and number of pages to read/write. In the pre-processing step, each trace is reduced to a small and well-optimized feature vector. Then,

a data set of these vectors is used for classification and training of a prediction model.

Job runtime and I/O prediction were done in PRIONN [9]. PRIONN is a neural network that is trained with constant size 1D or 2D images, that in turn are derived from variable length job scripts. This representation allows using deep learning algorithms. In the experiments on a real HPC this approach works with 75% mean and 98% median accuracy. It is also able to predict around 50% of the data bursts.

In contrast to existing work, we segment the job data into windows of activity that are characterized and investigate unsupervised methods for the analysis.

3 DKRZ Monitoring

DKRZ maintains a monitoring system that gathers various statistics from the Mistral HPC system, that has 3,340 compute nodes, 24 login nodes, and two Lustre file systems (lustre01 and lustre02) that provide a capacity of 52 Petabyte. The monitoring system is made up of open source components such as Grafana, OpenTSDB, and Elasticseach but also includes a lightweight self-developed data collector, that captures continuously node statistics – we decided to implement an own collector when analyzing the overhead of existing approaches. Additionally, the monitoring system obtains various meta-information from the Slurm workload manager and injects selected log files. A schematic overview is provided in Figure 1. The data is aggregated and visualized by a Grafana web interface, which is available to all DKRZ users: Mainly, that are three type of information about login nodes, user jobs, and queue statistics.

In the first place, a monitoring service gives the users an overview of the current state of the system, the current load of login nodes and Slurm partitions. For each single machine, a detail view also provides information (incl. historical data) about system load, memory consumption, and Lustre statistics. Job monitoring is enabled by default in a coarse-grained mode, but the functionality can be extended by Slurm parameters. With appropriate parameters, the monitoring system can gather information about CPU usage and frequency, memory consumption, Lustre throughput, and network traffic for each compute node. DRKZ also runs XDMoD on compute nodes for viewing historical job information as well as real-time scientific application profiling.

3.1 Metrics

In our experience, collecting many metrics from thousands of nodes in short time intervals creates a noteworthy overhead (exceeding 1%). Additionally, over time, monitoring data occupies a significant amount of data space. For a relatively large system like Mistral this is of particular concern – it would take at least 800 GiB to record a single metric with a 1s interval for one year on all nodes. Therefore, we reduced the number of captured metrics to a minimum.
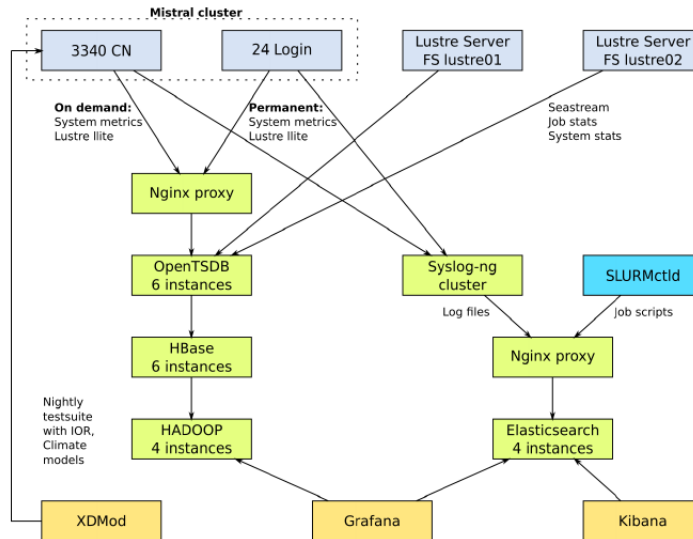
Fig. 1: Monitoring components.

To avoid much overhead, metadata and I/O metrics are selected in the following way: (1) Similar metadata counters are combined into three different groups: read, modification and other accesses. They create and unlink counters are captured separately. The exact group compositions and metric names are listed in Listing 1. (2) For I/O we just capture a set counters. The `read_*`, `write_*` and `seek` counters provide the basic information about file system access performed by the application. We also include the `osc_read_*`, `osc_write_*`, because the Lustre client transforms the original file system accesses, made by the application, to Lustre specific accesses – for instance by utilizing the kernel cache. This can have a significant impact on I/O performance (e.g., when many small I/O accesses are created but coalesced). The metrics are listed in Listing 2.

These metrics are collected in a 5sec interval for both file systems and send in JSON format to the Elasticsearch database.

4 Methodology

From the monitoring system we can map jobs to hosts and can extract a time series per host for the job runtime.

```
Source files: /proc/fs/lustre/llite/lustre*-*/stats
md_read = getattr + getxattr + readdir + statfs + listxattr + open + close
md_mod = setattr + setxattr + mkdir + link + rename + symlink + rmdir
md_other = truncate + mmap + ioctl + fsync + mknod
md_file_create = create
md_file_delete = unlink
```

Listing 1: Metadata metrics captured on compute nodes for lustre01 and lustre02.

```
Source files: /proc/fs/lustre/llite/lustre*-*/read_ahead_stats
osc_read_bytes, osc_read_calls
osc_write_bytes, osc_write_calls
read_bytes, read_calls
write_bytes, read_calls
seek
```

Listing 2: I/O metrics captured on compute nodes for lustre01 and lustre02.

However, machine learning algorithms can not be directly trained with the raw data collected from our monitoring system because most algorithms require a fixed number of inputs, but the job data is of variable length: Firstly, the job run times are variable. Secondly, jobs can run on any number of nodes producing one time series per node. In order to process them with machine learning algorithms, we convert them into a suitable fixed length representation. A pleasant side effect of the following data pre-processing step is data reduction.

In the first step, we split the job runtime in equal length windows. Since a parallel job can run on several nodes, we obtain a number of 2D segments for each metric. A schematic illustration of a 3×4 segmentation is shown in Figure 2.

In the next step, the segments are converted to $N \times N$ matrices ($N = \text{length}(v) = 12$ in our case), by computing statistics for each segment on runtime and nodes using the `stats()` function in Equation (1).

The conversion process for a segment is shown in Figure 3. Firstly, the `stats()` function is applied to all segment rows line by line, i.e., a statistics for each host's time series is computed removing the variability of job lengths. Then, `stats()` is applied to all columns of the intermediate result, i.e., a single statistics is computed across all hosts making the statistics independent of the number of hosts. The latter statistics is particularly relevant for large jobs. Finally, for each job and each metric we get a sequence of $N \times N$ matrices.

$$\text{stats}\left(\vec{v}\right) = (\min, \max, \text{mean}, \text{q01}, \text{q10}, \text{q05}, \text{q25}, \text{q50}, \text{q75}, \text{q90}, \text{q95}, \text{q99}) \quad (1)$$
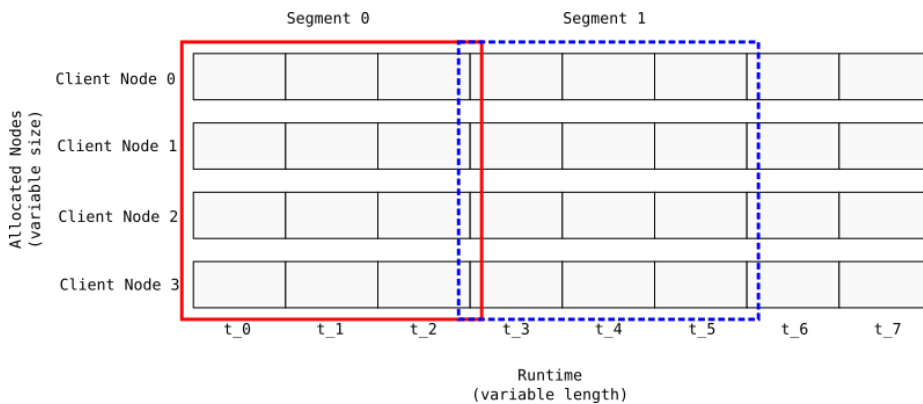

Fig. 2: Example of a job running on four nodes for eight time units and the resulting 3×4 segmentation.

In our experiments, we don't use the `seek` metric, i.e., we use the remaining 13 metrics (see Listings 1 and 2). On the whole, there are 12 stats (x-axis) · 12 stats (y-axis) · 13 metrics = 1872 statistics for each segment. A schematic representation of a statistic matrix that contains these values is given in Figure 3. On this pre-processing stage, we have a sequence of statistic matrices.
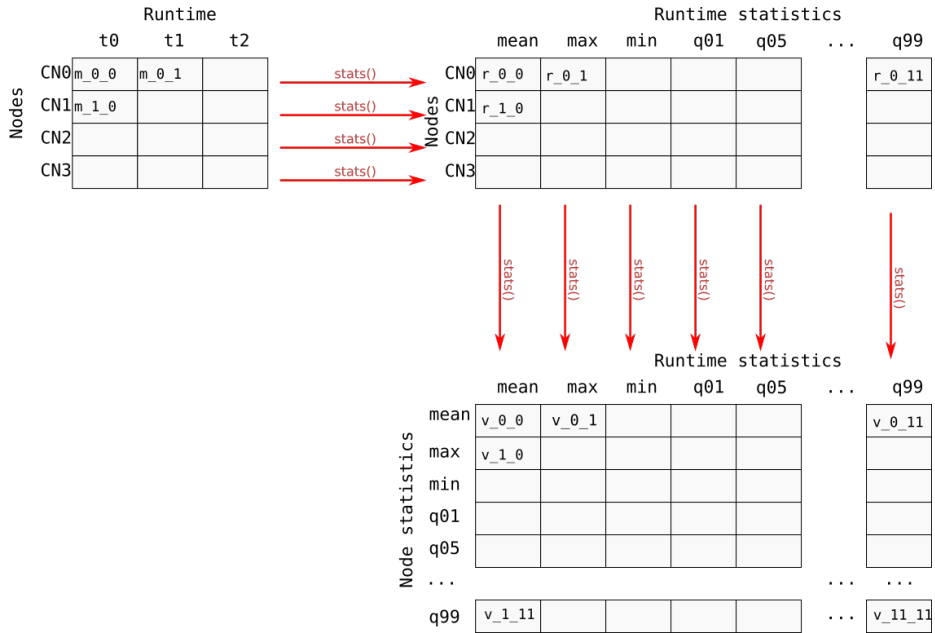
Fig. 3: Conversion of a 3×4 segment to a statistic matrix.

Our goal is to convert the sequences to fixed size vectors, called footprints according to Equation (2), where each element represents a kind of I/O behavior.

In the first step, we do the segmentation and create statistics matrices for all jobs. Then, we put them in a data pool, and group them by means of a clustering algorithm. The statistics in the same group are labeled with the same I/O class. After the identification of I/O classes, we compute footprints for each job, i.e., statistics which tell us how often the I/O classes occur in a job. In the last step, we use the clustering algorithm again to group the footprints to identify different types. (See Section 6.2 for details.)

$$\text{footprint}(\text{jobid}) = \vec{v}_{\text{jobid}}$$

(2)

with $\vec{v}$ is a fixed length numeric vector

A generic example in Figure 4 illustrates the basic idea: from a single footprint, we can compute the exhibition of four different I/O behaviors to a different

$$\text{footprint}(14400233) = \begin{pmatrix} IO1 : 20\% \\ IO2 : 10\% \\ IO3 : 30\% \\ IO4 : 50\% \end{pmatrix} \quad \begin{array}{l} \text{IO1: Metadata intensive} \\ \text{IO2: Using I/O node} \\ \text{IO3: Highly parallel I/O} \\ \text{IO4: No I/O} \end{array}$$

Fig. 4: Footprint example.

extent of the job runtime. Note that the classes IO1-IO4 must be identified and labeled manually.

5 Test data

For evaluation, we downloaded a data set for a time period of 5 days, from 2018-12-07 to 2018-12-13. It contains data of 70846 jobs. 33193 (47%) of them are jobs from `compute` and `compute2` Slurm partitions with exit status `COMPLETED`. These are used for our evaluation. Jobs not considered where either faulty, or short jobs run on the small partitions `gpu`, `miklip` and `minerva`. Additionally, ommitted jobs are from the `shared` and `prepost` partitions are shared by several users, so that monitoring data from these partitions can not be assigned unambiguously to a job.

Details of the job statistics are listed in Table 1.

JOBS	EXIT STATUS		JOBS	SLURM PARTITION
1,026	CANCELLED		**37,989**	**compute,compute2**
63,636	**COMPLETED**		241	gpu
5,753	FAILED		828	miklip
3	NODE_FAIL		34	minerva
426	TIMEOUT		31,752	shared,prepost

(a) Exit status statistics (b) Slurm statistics

Table 1: Dataset statistics of 70846 jobs captured on Mistral supercomputer in the time period of 5 days.

5.1 Data preparation

The time series of the job are split in 10 minutes large segments and converted to fixed size matrices, as described in the previous section. The leftovers at the end of job data and jobs that are smaller than 10 minutes are discarded. Finally, we apply the log10 function to all values, e.g., as a 10x increase in a statistics adds one to the distance. This shortens the distance from far-away classes and allows

clustering algorithms to combine neighbouring classes instead of grouping all observations similar to the maximum together and the rest. Theoretically, this pre-processing step would make it easier to recognize outlier classes, which are present in this data set.

After processing the monitoring data, we obtain around 128,000 statistics matrices. Out of convenience, these segment statistics are deflated to 1D-representation and stored row-wise in a file. The created dataset contains all statistics (1872 columns) and statistics matrices (around 128,000 rows). In further course of the paper use this representation for visualization and refer to them as samples.

6   Evaluation

6.1   I/O behavior classification

In the training phase, we feed a kMeans algorithm with 100,000 randomly selected samples from the dataset and obtain a model that can recognize 8 classes of I/O behavior. Table 2 shows the clustering result.

The resulting classes are quite different. We could describe only two of them, IO0 (Normal I/O) and IO3 (Intensive I/O). Five random samples for IO0 and IO3 are visualized in Figures 5a and 5b to give the reader an impression of the I/O behavior of each class. Arguably, the third and fourth example from the class IO0 still performs some read and write calls but with little data, while IO3 shows a consistenly high activity.

On the x-axis, metric names represent the whole set of segment statistics, e.g., `bytes_read` is a collective term for all statistics that were calculated statistics matrics for `bytes_read`. That are min, max, mean values and quantiles (q01, q05, q10, q25, q50, q75, q90, q95, q99) for nodes and runtime, i.e., 144 values/metric. Due space restrictions, we do not label them all, but show only the metric names. Values on the y-axis are scaled to a range between 0 and 1 for each statistic individually. This scaling was done only for the purpose of the visualization. It allows to consider all statistics in one picture.

IO0 represents non-I/O or typical storage usage class. With 91,28%, this is by far the largest class of all. Probably, several classes were combined into one

I/O Class	Size		Description
	# of Segments	in %	
IO0	117239	91.28	No I/O, Typical I/O, ...
IO1	552	0.43	
IO2	13	0.01	
IO3	471	0.37	Intensive I/O
IO4	1404	1.09	
IO5	8738	6.8	
IO6	5	0.0	
IO7	19	0.01	

Table 2: I/O class distribution.
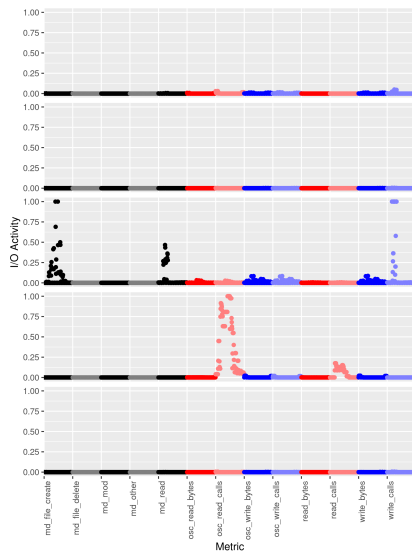
(a) IO0 - Normal I/O
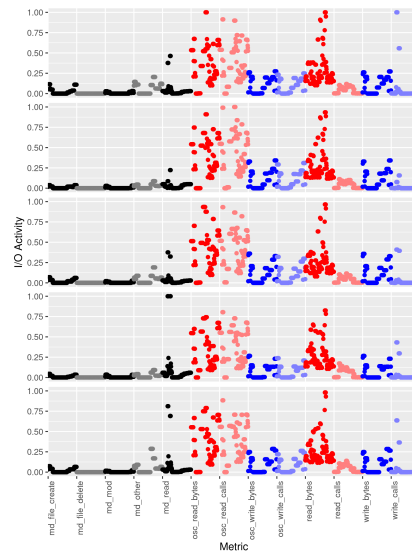
(b) IO3 - I/O intensive behavior

Fig. 5: Five randomly selected individual segments (y-axis) of the identified classes.

large class. Formation of such a large class helps to isolate outliers classes with extremely poor or intensive I/O performance.

IO3 shows an increased read I/O performance and large number of metadata reads. Both can interfere with each other, e.g., metadata and storage access are done sequentially one after each other, and neither I/O performance, nor meta data access can achieve full speed. Nevertheless, as we see later, this I/O class represents I/O intensive behavior.

The description and labeling by experts of other classes are difficult and requires further investigations. That becomes particularly apparent when the jobs are visualized individually. In Figure 5a we can see that the kMeans algorithm puts at least three different I/O behavior classes into the IO4 class. The same observation we could made for IO0 and IO5. This is at least hint, that the number of clusters was too small.

The samples in other classes are classes are similar and even for IO0 and IO3 we can not be sure, that they represent only one I/O pattern. Precise description can be done only by understanding of the rules of the trained model – it must be decomposed and analysed after each training. In practice this task is quite burdensome and very difficult to apply, hence we do not consider this approach to be beneficial by itself. We will look for alternatives in our further research.

```
io_sequence(14496682) =
[2 3 3 2 2 3 2 2 3 1 3 2 1 4 4 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 5 5 5 6 6 6 6 7 7 7 7 7]

footprint(14496682) =  [3, 44, 6, 5, 2, 3, 4, 5]
footprint_norm(14496682) = [0.04, 0.61, 0.08, 0.07, 0.03, 0.04, 0.06, 0.07]
```

Listing 3: Sequence of I/O classes.

6.2 Footprinting

After the first clustering pass, the time series can be represented as a sequence of the eight generated I/O classes. An example is shown in Listing 3. This sequence is still variable length and is not suitable for most machine learning algorithms. Therefore, we do another data reduction by counting I/O classes and obtain another fixed length vector, called (absolute) footprint. We also do a normalization of the vector, i.e., each value is divided by the sum of vector elements, i.e., in a normalized footprint the sum of vector elements is 1. Examples are given in Listing 3. Doing this for each job data, we obtain a new dataset consisting of 20,704 normalized footprints.

To group similar jobs we apply kMeans algorithm a second time on the footprint dataset and obtain a footprint classifier for 8 *job classes*. The footprints are visualized in Figure 6. As we can see, IO0 dominates in FP0 and IO3 in FP2. From this observation we conclude that FP0 represents jobs with low I/O activity and FP2 represents I/O intensive jobs. Unfortunately, without a precise description of other I/O classes we can not provide a reasonable description of the remaining footprint classes.

Class	Class size		Job score >= 6		Job score >= 7	
	# of jobs	in %	# of jobs	in %	# of jobs	in %
FP0	16003	77.29	5	0.03	0	0.00
FP1	2605	12.58	20	0.77	2	0.08
FP2	404	1.95	395	97.77	0	0.00
FP3	65	0.31	1	1.54	0	0.00
FP4	160	0.77	58	36.25	0	0.00
FP5	440	2.13	0	0.00	0	0.00
FP6	164	0.79	8	4.88	0	0.00
FP7	863	4.17	0	0.00	0	0.00

Table 3: Footprint statistics. I/O intensive jobs with score >=6 and >=7 were identified manually. (Class size percentage refers to total number of jobs. I/O intensive job percentage refers to class size.)

7 Manual identification of I/O intensive jobs

For verification, we identify manually the I/O intensive jobs. In the first step, we visualize the average performance of all metrics in density plots and determine
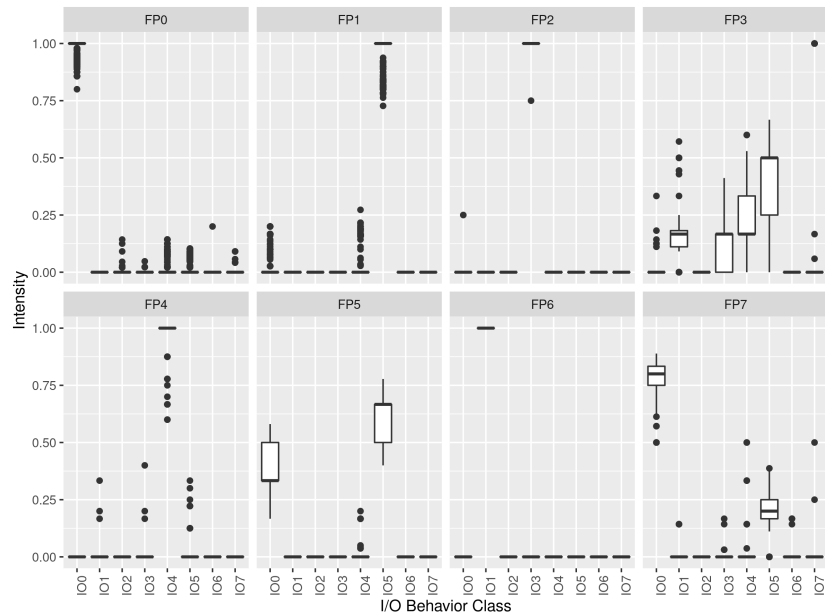
Fig. 6: Footprints of the different job classes. For each class, the percentage of each IO class is shown.

thresholds for normal, high and critical performance values. This is a highly subjective assessment without any consideration other factors, but for validation it is sufficient to pick a set of most I/O intensive jobs from our dataset. Then, we use the thresholds to label job performance, i.e., values for normal performance are labeled 0, values for high performance are labeled with 1 and for critical performance are labeled with 2. After that, for each job we obtain 13 labels. The sum of these labels is the resulting job score. The distribution is visualized in Figure 7.

According to the definition, the jobs with a high score are I/O intensive. The complete distribution is shown in Table 3. We pick jobs with high job scores and compare them with the previous results.

For job score >=7 there are 2 jobs. Both of them are located in the FP1 class, where IO5 dominates. IO5 contains several I/O subclasses, one of them is metadata sensitive. A closer look on both jobs reveals, that they are metadata intensive. This explain why kMeans put them in the FP1 class, but it is also a hint, that the amount of I/O classes should be increased.

For job score >=6 there are 487 jobs. Surprisingly, 97.77% of FP2 class are these I/O intensive jobs, and FP2 class identifies 81.1% of them. Hence, the automatically determined classes cover the most I/O-intensive jobs well, albeit it requires a manual step to first label them accordingly.
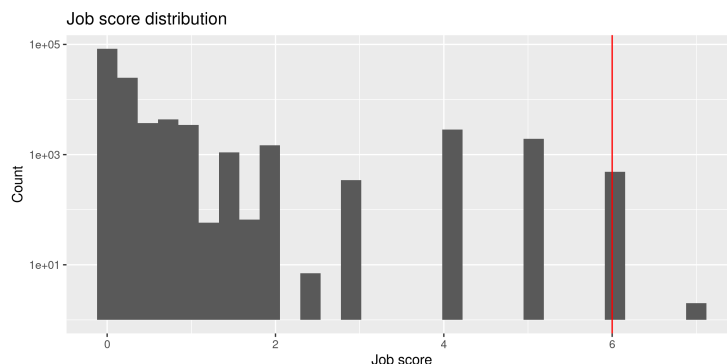
Fig. 7: Score distribution. Red line shows 99.8-quantile.

8 Summary and Conclusion

In this paper, we utilize an unsupervised machine learning approach to analyse internal I/O behavior of parallel jobs. The basic idea of the approach is to generate footprints of the jobs by splitting a timeseries of I/O metrics for a job into fixed windows and compute statistics per host and time window. Then, the kMeans algorithm is used to generate classes for each window, then the time series of a job is converted into a time series of classes and reduced to a vector that contains how often the classes occur during the job runtime. The different jobs are classified by running kMeans on the resulting time series creating a single class for each job. We then explore the result of the approach on a week's data of the Mistral supercomputer.

It turned out, that the most challenging part is to describe the automatically generated I/O classes, that are found by the clustering algorithm. We could identify three of eight of them: "normal I/O", "intensive I/O" and "other I/O", whereby "other I/O" is a container for I/O classes, that are not clear what they represent.

Surprisingly, applied to our small data set, the manual labeling lead to the situation that I/O intensive applications could be identified with a precision of 97% and a recall of 81%. However, the approach is still not completely automated and there is uncertainty in determining the number of relevant IO classes and job classes. Another weakness of this approach is that it gives no answer to the question, if it this approach works generally and how decision rules can be extracted and used for I/O class description.

As we found the unsupervised labeling not optimal, we are investigating to replace that portion of the workflow with a semi-manual labeling that is more easily comprehensible. We will keep the basic idea of the workflow, i.e., segmenting job execution into fixed window, as this allows to identify phases of I/O activity.

References

[1] Eugen Betke and Julian Kunkel. "Monitoring von Ein-/Ausgabe am DKRZ".
    https://zki2.rz.tu-ilmenau.de/fileadmin/zki/Arbeitskreise/SC/
    webdav/web-public/Vortraege/Jena2017/vortrag-Kunkel-Betke.
    pdf. 2017.

[2] Axel Busch et al. "Automated workload characterization for I/O per-
    formance analysis in virtualized environments". In: *Software Engineering
    2016*. Ed. by Jens Knoop and Uwe Zdun. Bonn: Gesellschaft fr Informatik
    e.V., 2016, pp. 27–28.

[3] Julian Kunkel et al. "Tools for Analyzing Parallel I/O". In: *High Per-
    formance Computing: ISC High Performance 2018 International Work-
    shops, Frankfurt/Main, Germany, June 28, 2018, Revised Selected Papers.*
    Ed. by Rio Yokota et al. Lecture Notes in Computer Science 11203. ISC
    Team. Frankfurt, Germany: Springer, Jan. 2019, pp. 49–70. ISBN: 978-3-
    030-02465-9. DOI: https://doi.org/10.1007/978-3-030-02465-9_4.

[4] Sandeep Madireddy et al. "Machine Learning Based Parallel I/O Predic-
    tive Modeling: A Case Study on Lustre File Systems". In: *ISC*. 2018.

[5] J. T. Palmer et al. "Open XDMoD: A Tool for the Comprehensive Man-
    agement of High-Performance Computing Resources". In: *Computing in
    Science Engineering* 17.4 (July 2015), pp. 52–62. ISSN: 1521-9615. DOI:
    10.1109/MCSE.2015.68.

[6] Jan Schmid and Julian Kunkel. "Predicting I/O Performance in HPC Us-
    ing Artificial Neural Networks". In: *Supercomputing Frontiers and Innova-
    tions* 3.3 (2016). ISSN: 2313-8734. URL: http://superfri.org/superfri/
    article/view/105.

[7] B. Seo et al. "IO Workload Characterization Revisited: A Data-Mining Ap-
    proach". In: *IEEE Transactions on Computers* 63.12 (Dec. 2014), pp. 3026–
    3038. ISSN: 0018-9340. DOI: 10.1109/TC.2013.187.

[8] Ozan Tuncer et al. "Diagnosing Performance Variations in HPC Applica-
    tions Using Machine Learning". In: *High Performance Computing*. Ed. by
    Julian M. Kunkel et al. Cham: Springer International Publishing, 2017,
    pp. 355–373. ISBN: 978-3-319-58667-0.

[9] Michael R. Wyatt II et al. "PRIONN: Predicting Runtime and IO Using
    Neural Networks". In: *Proceedings of the 47th International Conference on
    Parallel Processing*. ICPP 2018. 2018. ISBN: 978-1-4503-6510-9.

[10] Bing Xie et al. "Predicting Output Performance of a Petascale Super-
    computer". In: *Proceedings of the 26th International Symposium on High-
    Performance Parallel and Distributed Computing*. HPDC '17. Washing-
    ton, DC, USA: ACM, 2017, pp. 181–192. ISBN: 978-1-4503-4699-3. DOI:
    10.1145/3078597.3078614. URL: http://doi.acm.org/10.1145/
    3078597.3078614.

[11] Bin Yang et al. "End-to-end I/O Monitoring on a Leading Supercom-
    puter". In: *16th USENIX Symposium on Networked Systems Design and
    Implementation (NSDI 19)*. Boston, MA: USENIX Association, 2019, pp. 379–

394. ISBN: 978-1-931971-49-2. URL: https://www.usenix.org/conference/nsdi19/presentation/yang.