Identifying Relevant Factors in the I/O-Path using Statistical Methods

Julian M. Kunkel
March 2015

German Climate Computing Center (DKRZ)

Abstract. File systems of supercomputers are complex systems of hardware and software. They utilize many optimization techniques such as the cache hierarchy to speed up data access. Unfortunately, this complexity makes assessing I/O difficult. It is impossible to predict the performance of a single I/O operation without knowing the exact system state, as optimizations such as client-side caching of the parallel file system may speed up performance significantly. I/O tracing and characterization tools help capturing the application workload and quantitatively assessing the performance. However, a user has to decide himself if obtained performance is acceptable.
In this paper, a density-based method from statistics is investigated to build a model which assists administrators to identify relevant causes (a performance factor). Additionally, the model can be applied to purge unexpectedly slow operations that are caused by significant congestion on a shared resource. It will be sketched, how this could be used in the long term to automatically assess performance and identify the likely cause. The main contribution of the paper is the presentation of a novel methodology to identify relevant performance factors by inspecting the observed execution time on the client side.
Starting from a black box model, the methodology is applicable without fully understanding all hardware and software components of the complex system. It then guides the analysis from observations and fosters identification of the most significant performance factors in the I/O path. To evaluate the approach, a model is trained on DKRZ's supercomputer Mistral and validated on synthetic benchmarks. It is demonstrated that the methodology is currently able to distinguish between several client-side storage cases such as sequential and random memory layout, and cached or uncached data, but this will be extended in the future to include server-side I/O factors as well.
Revision: This paper contains a slightly extended related work section based on feedback I received.

1 Introduction

Data-driven science became the fourth scientific paradigm besides theory, experimentation and computation. Dealing with the analysis of the vast output of scientific applications and experimental and observational data, pushes the

hunger for performance of HPC storage systems. Often when running applications, performance stays behind the capabilities of the parallel file system; it is not untypical to achieve only 10% of the performance expectations. Due to the complexity of the hardware and software stack and the shared nature of the file system resources, there are many possible explanations for such an observation. On the lower POSIX level, an I/O access pattern is characterized by the sequence of the operations, each operation can be characterized by the attributes: type (read or write), offset to the last operation (is the pattern more random-like or sequential), time after the last operation and access granularity (size). Observable performance of an I/O operation depends on the access pattern but performance for repeating operations with the identical attributes can vary significantly due to optimizations deployed along the I/O path. For example, for write operations, data may fit into the client-side cache or it may trigger flushing cached data to the server. Similarly, for reads, data may reside in the client-side cache, data may need to be fetched from backend storage from the server, may reside in the RAID controller's cache. Finally, accesses causing disk I/O may involve a short or long seek of the disk's actuator. Additionally, on each involved layer, concurrent operations compete for the available processing time and lead to further queuing time.

Tools such as IOPro [15], Darshan [7], and SIOX [19] help characterizing access patterns on the application or the system side. These tools allow to capture the client-side I/O behavior, manage them in either timelines or profiles and aid in visualization of the results. File system tools such as the LMT [29], MELT [4] and vendor-specific tools such as `mmpmon`, ClusterStor Manager or BeeGFSs Admon allow to understand the utilization of the servers. None of these tools help in assessing the quality of the I/O, i.e., whether or not I/O performance is adequate has to be decided by the users. However, the assessment is the difficult part, for example, is an access time of 10ms adequate when reading 16 MiB of data? What is the (likely) cause for the long duration? Sometimes I/O is significantly slower due to an extreme congestion on a resource. While this is important from the application perspective as it slows down observed I/O performance significantly, these events are usually rare and distort analysis of expected I/O behavior. From the application perspective, an assessment of the cause for the observed performance of individual I/O operations would not only increase transparency for the user and application developers but also highlight the potential of any subsequent I/O optimization. For example, knowing the fraction of I/O operations that suffer from unexpected slowdown helps understand system congestion.

In this paper, a methodology is developed that allows assessment of the I/O only by analyzing the response times for the I/O operations on the client side. This approach could easily be implemented in a tool such as SIOX or Darshan to provide a report after application termination which contains the likely causes and elaborates the optimization potential.

The contribution of this paper is the development of a model to classify relevant performance factors. It is based on density-based clustering and linear modeling for the influence of data size for relevant performance factors. Perfor-

mance factors lead to classes of measurements with similar behavior/cause. By itself, the methodology cannot identify the cause behind the individual classes i.e., we would observe performance class 1, 2 ... However, with additional system knowledge, an administrator can customize the class labels, e.g., it is due to caching on the client/server side, to help users understand the behavior better. Overall, the methodology fosters the understanding of relevant performance factors within the I/O path and fosters I/O characterization.

The paper is structured as follows: Related work is given in Section 2. The method for density-based outlier detection is illustrated in Section 3. In Section 4, an algorithmic description of the approach is given. This includes outlier detection and an approach for predicting (previously identified) causes of I/O data using linear models. The evaluation in Section 5 validates the method on DKRZ's Mistral supercomputing system and shows that it is effective for assessing whether an I/O operation is performed on the cache. Finally, the paper is concluded in Section 7.

2 Related Work

Related studies can be classified into performance analysis & modeling and methods from statistics for outlier detection and identification of unknown factors.

Performance characterization of parallel file systems To understand and optimize the I/O stack, the analysis of application I/O and the resulting server behavior are subject to many investigations, e.g., [2, 3, 20, 28]. However, as the I/O stack is complex, most evaluations are not conducting a root cause analysis. In an attempt to identify the cause, at best, operating system statistics are looked at.

Tools for monitoring I/O operations For understanding behavior of parallel systems usually profiling and tracing tools can help. Tools such as TAU [27], Vampir [16], and Scalasca [9] record time for individual operations and potentially CPU performance events. In so called traces, the timestamps are preserved – thus each individual start and end time can be analyzed. In profiles, timestamps are irrelevant and the overall behavior is recorded showing how runtime is distributed across the program. These tools can monitor the runtime of MPI and POSIX I/O calls and shed light on the distribution of compute vs. I/O time. However, they cannot analyze the access pattern. Therefore, tools such as IOPro [15], Darshan [7], and SIOX [19] help characterizing the observed access pattern. There are also many tools to capture, generalize and replay workloads. For example, Hidden Markov Models [34] or statistics can be used to mimic the access patterns of clients better. All these tools require the user to decide the quality of the observed operation's performance. Is a particular I/O operation with a runtime of 0.2 ms fast or slow? If it is slow, we would be interested in the likely reason behind this to prevent this case. Unfortunately, no existing tool helps to assess the quality of the I/O and to identify the reason.

Tools aiding to understand the I/O path It is possible to trace the I/O path of individual operations, e.g., using for local storage `blktrace` [5] and for distributed storage tools such as ScalaIOTrace [32], PIOviz [22], and HDTrace [18]. For distributed systems, this requires additional instrumentation on the server side. Needless to say that analyzing these traces is complicated and a fine grained level of tracing comes with relevant overhead. Therefore, tools that simplify the analysis are valuable assets.

There are tools that aim to identify the implemented optimization strategy such as cache policy or RAID level. In [6], the tool Dust is introduced which uncovers the cache replacement policies, e.g., LFU and FIFO, of the operating system by applying a well defined access pattern. For each replacement strategy, a fingerprint can be created to allow identification of it. The authors use a threshold to distinguish between cached and uncached I/O. They do not analyze the distribution of access times further and classify behavior only for one access granularity. Following a similar approach, the tool Shear applies access patterns to reveal the configuration of a RAID storage [8].

Performance modeling of storage The modeling of application and system behavior fosters understanding of these complex systems and enables predictions (what-if analysis). Simple performance models, for example, based on latency and throughput [20], can be applied to identify the room for optimization. Additionally, there is a long history in modeling the data processing of these systems on the component level, usually by using queuing networks [10]. Example simulators are DiskSim [33] focusing on a single storage device, CODES [21], SST [25], and PIOSimHD [18] that allow simulations of larger systems.

Outlier detection Performance counters such as Bytes (read/written) that can be obtained from the operating system of storage servers can be used to identify issues. Kasick et al. [13] use this strategy to identify faulty resources. A system that exhibit a substantial different behavior than the pack of servers is suspected to misbehave. However, the investigation of client side performance and the causes for performance degradation is substantial different.

There exist many methods for outlier detection such as statistical approaches, Bayesian networks, and machine learning. These may also be used for identifying abnormal behavior that is of special interest [24]. Statistics offers a wide range of tests based on a given distribution, deviation, distance or density. The assumption that individual measurements follow one Gaussian probability distribution is not correct. Firstly, the queuing in any shared resource such as node-local memory or the shared file system servers are expected to lead to Gamma distributed data. Secondly, even with well-defined experimental conditions, a sequence of identical operations can trigger different I/O paths; for example, with write-behind caching, a write in the Linux kernel may trigger a flush to the storage or it may just require a copy to the page cache. When executing a call, the executed I/O path cannot be selected and is subject to the current system state. In this paper, experimental evidence is provided that substantiates this claim.

Density-based approaches such as INFLO [12] can help in this case as they can account for any distribution. However, as far as known to the author, these important techniques have not found adoption in the storage community. A light-weight approach of density-based analysis is chosen by Uselton et al. in [31]. Their ensemble analysis investigates the runtime of individual processes and investigates the resulting histograms. This allows to identify clusters of processes with similar runtime, speculate on causes and identify the optimization potential if all processes were started at the same time.

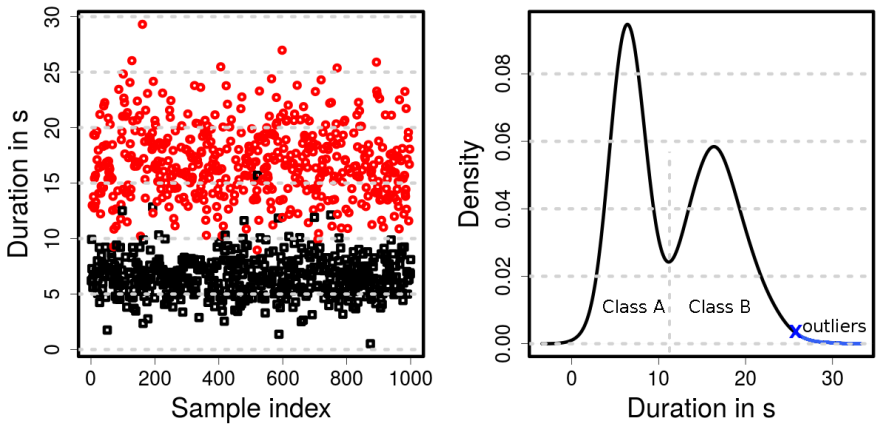
Factor analysis An easy definition is given in Wikipedia [1], Factor analysis "is a statistical method used to describe variability among observed, correlated variables in terms of a potentially lower number of unobserved variables called factors. For example, it is possible that variations in say six observed variables mainly reflect the variations in two unobserved (underlying) variables. Factor analysis searches for such joint variations in response to unobserved latent variables". The method constructs a linear model for explaining unobservable (latent) variables based on the observable random variables. Therewith, at a first glance, exploratory factor analysis is a potent approach for analyzing I/O performance.

However, the I/O path of any given operation may trigger certain behavior in the middleware and file system based on the unobservable system state. Assume, for example, starting a sequence of identical write operations, if the client-side cache is full, the write triggers write-back to the server that in turn may trigger additional processes. The underlying behavior is highly non-linear as it depends on the hidden system state. Initial analysis during the authors past work[17] has revealed these kind of patterns and lead to the research in this paper. Nonlinear factor analysis [11] is a potential solution but assumes normally distributed variables. Due to the shared nature of the compute and storage resources, the queuing of operations leads to a Gamma distributed performance. Additionally, the methods are quite difficult to understand and apply. So far the author does not know any application of this method in the storage domain. In contrast, the method developed within this paper is easier to understand and aids better in identifying latent factors.

3 Illustration of the Density-Based Clustering

Assume you have measured the duration of an I/O operation with certain parameters repeatedly. In any real system we see alternative processes but also additive components, e.g., when a cache miss triggers network operations and I/O that may follow other distributions. Since a client executes multiple processes and a file system is a shared system, the Gamma distribution is suitable to model waiting times until the operation is processed in any of these layers. Assume we have two processes that are based on Gamma distributions, one with shape=10

[1] See `https://en.wikipedia.org/wiki/Factor_analysis`; for a detailed description refer to [14].



(a) Timeline **(b)** Density reveals two classes

Fig. 1. Example demonstrating the methodology. Each of the two Gamma distributed processes is drawn with its own color.

and one with shape=1 (both with scale=1). When doing an operation based on some system settings such as cache hit or miss, the representative operation is performed where the one with shape=1 is usually 10 times as fast as the other. These two processes are overlayed with a Gaussian distribution: An additional Gaussian noise with mean 5 and standard deviation of 1 is added that is always in the path. Normally, optimization techniques such as caching are used that optimize performance significantly and those relevant processes can actually be identified in a density graphs as demonstrated.

If both processes are equally likely to happen and we repeat this process 1000 times, we obtain a figure such as Figure 1a. The observed durations can then be converted to a density that describes the duration for the particular call: The continuous density function is constructed by estimating the relative number of occurrences of each value, e.g., how often do we observe the duration 0.4s, 0.5s and so on. The result is visualized in Figure 1b; in essence, such a density plot can be thought as a smoothened histogram. By normalization, these relative frequencies are converted to obtain the probability density function (PDF), which defines the likelihood to observe individual values. The area under the curve between two values x and y is the probability that the value falls between x and y. The density function can be estimated using e.g., the statistics tool R's density function with Gaussian kernel and nrd0 bandwidth estimator[2].

[2] For the computation of the density, each x coordinate is considered to be the center of a Gaussian and observations covered under the Gaussian are taken into account for computing the actual value. The relevance of each point depends on the value of the Gaussian. The width of the Gaussian is chosen by a bandwidth factor that can

We can see that the distribution reveals two regions with clusters where most measurements happen (one up to Duration 11). While we don't know their causes yet, we can conclude that the system behavior results in two clusters and thus, for simplicity, we assume there are two different processes that we call relevant performance factors behind them. In the figure, those are marked as Class A and Class B. If we know the cause behind them, we can give them meaningful labels such as "cached on client-side" and "uncached".

We can also use the density diagram to identify outliers i.e., unexpectedly slow operations that are untypical for the I/O. For example, it may happen that a few operations take up to 2 seconds while 99.99% of the operations are served faster than 20 ms. Algorithmically, this can be done by inspecting the slope of the slowest cluster and define a cutoff point, in the figure this is marked with X. This approach is much more robust than using the standard deviation as our example does not follow the Gaussian probability distribution.

4 Methodology

The methodology consists of two parts. Firstly, we identify the relevant performance factors for individual access granularities and strip outliers. Secondly, the results are approximated with a linear model that allows to predict performance for any size. By constructing a linear model for a well known experimental setup, they describe the expected duration for the experimental setup and, thus, the cause. Multiple models are created, and trained. Later, the model (and its inherent experimental setup) that explains our observation best, is assumed to be the cause.

4.1 Identification and classification into relevant factors

We propose the following black box approach:

1. Repeatedly measure execution time of an I/O operation.
2. Construct the density graph. This effectively smoothes the histogram.
3. Identify clusters in the density graph and their extrema.
4. Define separation points, usually minima, but a split can also be the mean between two maxima.
5. Partition observations into classes based on the separation points. We assume each class is caused by (at least) one significant performance factor.
6. Optional: Identify the root causes behind classes and assign appropriate names such as "client-side cached" or "average seek time".

The approach does not aim to classify all observations correctly, but instead aims to help us understand the performance factors. Additional system knowledge can be applied on the initial black box model to label the clusters according

be automatically determined, thus, the observed smoothness depends on the selected bandwidth estimator.

to known factors such as client-side cached vs. uncached. Applying the approach to our toy example, we would see two classes, and one split point (see Figure 1b). Classification would result in several black points belonging to the second cluster and vice versa, but the majority of data points of the two Gamma processes would be classified correctly.

This approach also effective to purge outliers – i.e., abnormally slow accesses that are likely due to extreme conditions, by removing all trailing data points after the last cluster – in the figure, the point marked with X, data points with unexpected long duration that are apparently insignificant are removed. The data point X can be determined by walking downhill from the last extrema until the density falls below a threshold based on the last maximum (for example, below threshold $= 0.05 \cdot$ last maximum).

4.2 Prediction of Causal Issues

A limitation of the introduced method is that it can only be applied to investigate a particular operation. Since the duration of the I/O operation is expected to be strongly correlated with the size, a predictor for any fixed size is of limited use. In this paper, we construct predictors for several conditions bottom up:

1. Run a large set of experiments with varying sizes and access patterns that lead to well-known behavior i.e., involves expected performance factors.
2. Apply the density classification to identify clusters for each size and configuration and purge outliers.
3. Create linear models to predict duration, i.e., $d(\text{size}) = I + c \cdot \text{size}$. Where c and I are the fixed model parameters. Each performance factor is identified with its own model. Since read and write path are different, we create individual models.
4. Due to the non-linearity of the I/O behavior for sub-page sizes, we train two linear models, i.e., between 1 and 4096 bytes (page size) and one for larger accesses. This can be easily justified for Lustre, as operations are performed only in multiples of page size.

To apply the model in a real system and determine the likely cause, the duration of an observation is predicted with the ensemble of linear models each created for different performance factors. The model which yields the best approximation of the duration is assumed to explain the behavior seen. To be more accurate, the approach chosen in this paper sorts the models based on their predicted time for the outliers, then for any observation it picks the first model for which the duration $d_{observed} \leq d_{model}(size)$. For example, assume we have the following models for predicting I/O performance: discard (i.e., no I/O at all), cached by OS, hdd-cached, hdd-uncached. Each of these models predict the expected maximum time under the respective condition for each I/O size. Clearly, for a fixed size, the time needed for discarding I/O will be shorter than for cached I/O and so on. If we observe a time that is between the prediction of the model discard and cached, then we conclude that this is likely to be a cached I/O.

5 Evaluation

This section is organized as follows: First, the test system is introduced. Then, the I/O benchmark is described together with the varied parameter space. Next, the results obtained on the parallel file system Lustre are explored showing the difficulties to investigate the causes for observed behavior. Finally, results from local I/O on a single server are briefly described – this demonstrates that similar results are obtained from local I/O and, thus, the methodology can be applied across file systems.

5.1 Test system

The evaluation is conducted on DKRZ's recently installed supercomputer Mistral, which delivers more than a PFLOP of computation performance with its 1500 nodes. Each node is equipped with two Intel E5-2680v3 @ 2.50GHz each providing 30 MiB L3 cache and 64 GByte of main memory. The Lustre 2.5 storage offers 30 Petabyte of capacity and consists of 29 ClusterStor 9000 server pairs providing 58 OSS and 116 OSTs and delivers more than 300 GiB/s. The system is in production, all tests are made on our single Lustre file system.

5.2 Conducted Experiments

A new benchmark for POSIX I/O has been written called `io-model`[3]. It measures a variety of memory and file layouts, times each individual operation with high-precision and outputs these values together with offsets for convenient post-processing as CSV-file. In contrast to existing benchmarks for analyzing local or parallel I/O such as IOzone [23], fio [1], and IOR [26], the goal of this benchmark is to actually provide all the necessary input to automatically build the discussed models. Existing benchmarks lack the flexibility to vary memory access patterns and precise timing of individual I/Os – sub microsecond accuracy is needed, that is needed for this task. To allow labeling of interesting cases, a parameter space is explored; each run is parameterized with the settings:

- Mem layout: Defines how a memory buffer is accessed. In the introduced experiments, the buffer is always preallocated and initialized and used as a circular buffer (if needed) with 1 GiB of size. Settings are: off0 (we always read/write to/from the buffer at position 0), seq (sequential buffer access), stride8MiB (after accessing data, skip 8 MiB on the buffer), reverse (access the buffer from back to front), rnd (randomly access any position), rnd8,8MiB (seek 8 MiB +- uniform random up to 8 MiB)
- Disk layout: Defines the spatial access pattern used when accessing the file. Settings are identical to mem layout.

[3] See `https://github.com/JulianKunkel/io-modelling`, the documentation of automatic model creation is an ongoing effort.

– Access size: Defines the access granularity for `read()` and `write()` in Bytes. Settings are: 1, 4, 16, 64, 256, 1K, 2K, 4K, 8K, 16K, ..., 2MiB, 4MiB, 8MiB, 16MiB[4].
– File size: 10 GiB (and in a few cases 1 TB).
– Repeats: Number of I/O operations per run. 10K or 1M.

Additionally, system states are varied:[5]

– Caching: Defines the status of the page cache. *Discard* means we read from /dev/zero or write to /dev/null. *Cached* means the full file is pre-read additionally at the beginning of the batch run using *dd*. *Uncached* means the page cache is cleaned using `sysctl vm.drop_caches=3` and it is checked for having been freed[6].
  Note that the goal from measuring the time for an I/O would be to identify the actual caching state, e.g., was data cached in main memory, already loaded on CPU cache or not cached at all. Therefore, reading from /dev/zero serves the purpose as upper bound for in-memory data transfer (memory copy) between page-cache and user-space buffer.
– Thread count: 1 or 20. Defines how many independent processes of the benchmark have been run. If multiple threads are used, each accesses its own file but the aggregated file size is identical to the 1 thread case.

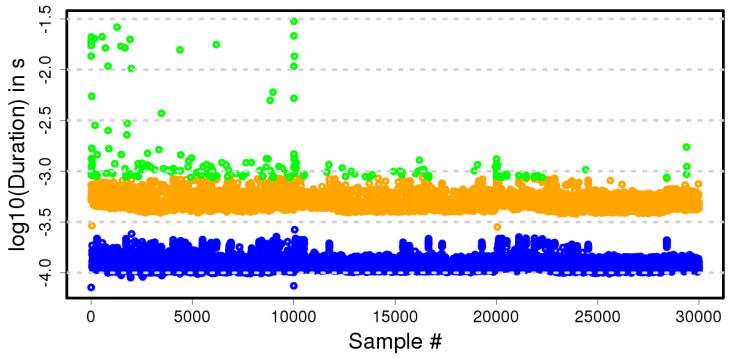
Files are reused between runs and only one stripe is used for each file, thus, a single stripe remains on the identical OST[7] for all experiments. When using 20 threads, each file is (usually) placed on a different OST by the default Lustre policy. For each setting, the benchmark run is repeated three times. This is done in an outer loop running all configurations, thus, noise that affects one run can be identified when comparing them.

5.3 Analyzing the Lustre Behavior

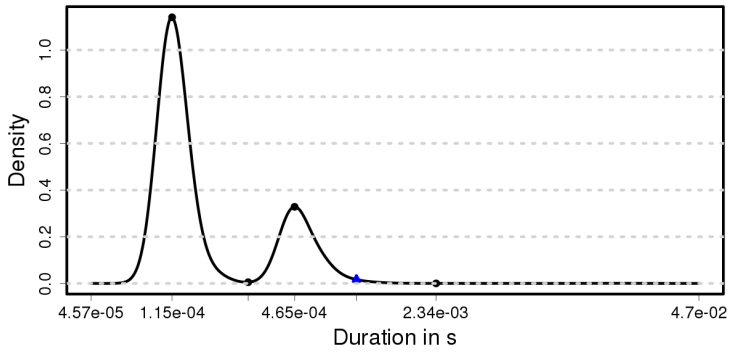
When running the experiment three times with 10000 operations, a timeline is obtained that can then be used to compute the density. An example is given in Figure 2, the upper figure shows the individual measurements in logarithmic scale and the lower figure the resulting density graph. The graph clearly shows that there are two performance factors with mean 0.115 ms and 0.465 ms corresponding to a throughput of 2170 MiB/s and 537 MiB/s, respectively. The long tail in the density corresponds to individual operations that are significantly slower than the typical operations. With the proposed method, the clusters are

[4] A preliminary experiment with non-block multiples of access sizes led to comparable results, therefore, they were not analyzed further.
[5] Additionally re-write vs. truncated files was tested but did not show relevant differences. Therefore, these results are omitted.
[6] Since Lustre does not guarantee to free the dirty pages but the call returns, this procedure is repeated until the page cache memory drops below a threshold.
[7] In Lustre, data of a file is partitioned into stripes that are distributed across Object Storage Targets (OSTs).

identified and so is the tail for unexpectedly slow operations – up to two orders of magnitudes slower. Including outliers an average throughput of 1.1 GiB/s is achieved and without 1090 MiB/s. These outliers are purged for subsequent analysis. By analyzing statistics from */proc* (see Table 2), it can be observed that about 2500 RPC reads are performed to the Lustre servers and each operation took about 1 MiB. Also it can be seen that 640k cache hits occur, thus Lustre correctly pre-fetches the sequentially accessed data and almost no cache misses (3 pages). Based on this analysis, it seems likely that the first performance factor represent data completely cached in the client cache achieving more than 2 GiB/s throughput. The question remains why many calls result in the average performance of 537 MiB/s. To understand this, a detailed inspection of the Lustre I/O path would be necessary. Presumably, additional pre-fetching is performed as part of the regular read, although it appears as cache hit. Thanks to the black box methodology, two relevant conditions could be identified and can be inspected further.



(a) Timeline for individual reads



(b) Density plot of the observations

Fig. 2. Duration for sequential reads with 256 KiB accesses (off0 mem layout).

	Results when purging outliers							Incl. outliers	
Typ/	AccessSize							AccessSize	
File layout	1	1K	4K	64K	256K	1M	16M	4K	256K
R-off0	146K	139M	478M	4.2G	5.9G	7.2G	4.8G	408M	5.5G
R-seq	146K	108M	403M	1.1G	1.1G	1.2G	1.1G	204M	1090M
R-rnd	1.2K	608K	557K	14M	33M	117M	586M	5.1M	80.9M
R-rnd8MiB	17K	653K	970K	425M	2.4G	6.4G	4.4G	7.5M	1.2G
R-stride8,8M	3.5K	164K	491K	14M	40M	133M	797M	4.4M	79M
R-reverse	144K	962K	2.0M	28M	56M	337M	851M	15.1M	95.5M
W-off0	120K	115M	419M	2.2G	2.6G	2.9G	263M	236M	2.3G
W-seq	121K	103M	287M	1G	1G	1G	948M	171M	925M
W-rnd	4.1K	3.5M	9.3M	135M	341M	582M	809M	4M	128M
W-rnd8MiB	4.7K	4.2M	12M	243M	837M	63M	332M	5.8M	61M
W-stride8,8M	4.2K	3.7M	11M	150M	381M	589M	835M	4.4M	185M
W-reverse	115K	19M	291M	990M	963M	1005M	976M	133M	883M

(a) With cached data

	Results when purging outliers							Incl. outliers	
Typ/	AccessSize							AccessSize	
File layout	1	1K	4K	64K	256K	1M	16M	4K	256K
R-off0	178K	172M	654M	4.6G	6.2G	7.1G	5.2G	612M	6.1G
R-seq	169K	164M	550M	1.1G	1.2G	1.2G	1.1G	353M	1.1G
R-rnd	275	240K	502K	5.6M	19M	134M	635M	515K	20M
R-rnd8MiB	5.6K	366K	1.7M	34M	2.6G	6.7G	4.5G	2.3M	1020M
R-stride8,8M	274	103K	393K	6.2M	20M	119M	736M	395K	20.6M
R-reverse	172K	620K	746K	10M	27M	115M	780M	723K	26.2M

(b) With clean cache

Table 1. Mean throughput in Byte/s for selected access granularities (off0 memory layout). The mean when purging the outliers is given for reference. Green is good, purple is higher than network throughput and yellow/white are slow.

Typ	Lay-out	Page cache hits	Page cache misses	read b_avg	write b_avg	osc_read avg	osc_read calls	osc_write avg	osc_write calls	Perf. in B/s
Runs with accessSize of 256 KiB										
W D	off0	0	0	201	40K	0	0	32K	0-6	1.1T
W C	off0	0	0	201	262K	0	0	256K	1.1	2.6G
W C	seq	0	0	201	262K	0	0	4M	625	1G
W C	rnd	0	0	201	262K	4096	19K	3.9M	673.6	341M
W C	rev	0	0	201	262K	0	0	4M	626	963M
R D	off0	0	0	201	40K	0	0	42K	0.4	14G
R C	off0	63	1	256K	40K	256K	1	0	0	5.9G
R C	seq	640K	3	256K	57K	1M	2543	80K	0.4	1.1G
R C	rnd	615K	16K	256K	58K	241K	20K	180K	4	33M
R C	rev	629K	10K	256K	58K	256K	9976	104K	0-3	56M
R C	rnd8,8	630K	17K	256K	5	252K	20 K	180K	4	
R U	off0	63	5	256K	40K	64K	5	0	0	6.2G
R U	seq	640K	6	256K	57K	1M	2546	0	0	1.2G
Runs with accessSize of 1 MiB and a 1 TB file, caching on the client is not possible For seq. 1M repeats are performed, for random 10k.										
W	seq	0	1.3	201	1M	0-8K	0-4	4M	250K	1007
W	rnd	0	0-3	201	1M	4097	20K	3.2M	3309	104
R	seq	255M	2	1M	2.5M	1M	1000K	3M	10	1109
R	rnd	2M	9753	1M	60K	836K	24K	100K	3	55
Accessing 1 TB file with 20 threads, aggregated, performance is reported per thread										
W	seq	0-1	0-3	201	1M	2-17K	1-3	4.1M	254K	250
W	rnd	0	0	201	1M	4096	1.8M	3.1M	320K	138
R	seq	250M	480K	1M	21-24K	1.6M	630K	717K	41	168
R	rnd	240M	900K	1M	20-23K	832K	2.3M	523K	36	47

Table 2. Deltas of the statistics from `/proc` for runs with access granularity of 256 KiB and 1 MiB (mem-layout is always off0). In the type column, D stands for discard, C for cached and U for uncached. 1TB files do not fit into the cache.

An overview of the achieved performance for selected access granularities is given in Table 1. Note that the mean arithmetic performance is computed after outliers are purged. For 4 and 256 KiB the original throughput is kept for reference. In the mean, about 2.1% of the observations are classified to be an outlier when removing the tail of observations (see Figure 2 for a complete example, the method would purge the tail after the blue triangle). In a few cases – especially for very small access granularities, up to 15% of the measurements behave very slow due to congestion and background activities of daemons. While in sequential access patterns, the unexpected slow operations do not influence the throughput much, for random patterns they degrade performance significantly, e.g., 557 KiB/s throughput for random reads with 4 KiB accesses can be observed with outliers and 5.1 MiB/s without. The reason is simple, a few operations are several orders of magnitude slower than the typical accesses. This is not crucial for the subsequent analysis as we are interested in the typical behavior and not in extreme cases.

In Table 1, colors encode the performance relative to the node's maximum Lustre performance observed so far (5 GiB/s out of the 6 GiB/s delivered by the FDR-IB network performance). When reading data from the pseudo device */dev/zero* (not shown in the table), a best performance of 15 GiB/s is achieved when using off0 memory layout and all data fits into the CPU cache. This is expected to be similar to cases in which data fits into the client cache; in both cases, the Linux kernel performs a memory copy between kernel-space and user-space buffer. Sequential memory access halves performance as data needs to be written back to main memory. Even though read is a mere memcpy, there is a slight performance difference when changing the file layout for small access sizes (not shown). This is caused by the benchmark that also measures time for `lseek()` and computing the next offset, but its effect is very small to affect the conclusions of this paper. Write to */dev/null* actually throws away the data and does not require a memcpy, therefore, they all behave similarly and achieve unrealistic speeds.

The results for experiment with cached data shows that Lustre does perform significantly worse than reading from `/dev/zero`. This is surprising as the setup (and when checking free memory), a copy of data appears to reside in the page cache. Looking at detailed Lustre statistics from */proc*, it turns out that with the exception to rnd8,8M and off0, all other access patterns trigger server activity although no other process accesses the data. Other experiments have shown that the single stream performance on our system is about 1.1 GiB/s. Comparing cached and clean cached system state, those perform similarly as the client-side cache is not used as expected. The experiment with rnd8,8M and large access granularities may access previously accessed data immediately, therefore, the cache might be fresh enough.

To understand the behavior better, during the execution OS and Lustre's llite and read-ahead statistics are fetched from `/proc` before and after each configuration is run. The deltas for the counters are shown Table 2; values are rounded and if they vary significantly between the repeats of each experiment,

the span of the observed values is given in the table. Hits and misses are Lustre statistics (in 4k pages), the next columns contain the operating system statistics for the system I/O, and the next two columns the Lustre client statistics about the number of RPCs transmitted and their average size. From the table, several assumptions about the experiments and Lustre behavior can be investigated, e.g., it can be verified that cached and uncached reads trigger similar activities, random writes cause to request the partially overwritten blocks. Each random read triggers about 12 interrupts and this case shows the highest number of memory accesses (not shown in the table).

The statistics also include data for 1 TB files and for 20 threads; the single thread performance for sequential access is similar to the one achieved with 10 GByte files. However, for random access, performance degrades substantially (to 104 MiB/s). With increase of file size, a degradation is expected as the ClusterStor 9000 back-end storage relies on a declustered RAID on HDDs and larger offsets increase the seek time of the HDDs to around 10 ms.

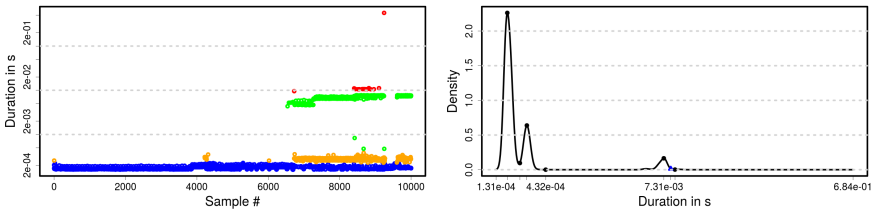
When using 20 threads[8], the achievable performance increases due to known bottlenecks within the Lustre client [30], finally 5000 MiB/s are achieved for sequential write with 20 threads. Compared to a single thread, the random performance stays on a comparable level, as each file is placed on another OST and they are accessed concurrently.

5.4 Validating the approach for local I/O

Similar timelines and density plots can be seen on other file systems as well. In this section, we will briefly look at node-local I/O. In this experiment, the benchmark is run on a server equipped with two Intel Xeon X5650 Westmere processors, 12 GB of main memory and a local HDD and formatted with an Ext4 file system. Figure 3 shows the timeline and density for sequential writes with 256 KiB of data. This figure shows only one run of the benchmark, i.e., writing 10,000 a 256 KiB block. It can be seen that there are again several clusters. In this case three, one at 0.2 ms (about 1250 MiB/s is achieved), one at 0.3 ms and one at 7.3 ms.

This time, due to the reduced complexity, the reason for the different clusters in the graph can be explained better. One potential explanation is as follows: At beginning, the write-behind cache of Linux works well and all operations end quickly (lower cluster, blue operations). After a while Linux pdflush deamons start to write out the dirty pages which only slightly defers the I/O (slightly slower cluster, caused by locking inside the kernel). Unfortunately, the benchmark is too fast and the kernel stops the write call to flush pages as well. Clearly, this is much slower and the latency of the disk with about 7 ms becomes visible. There could be other explanations clusters but most importantly with the

[8] Note that with 20 threads, over three repeats the runtime of independently started benchmarks varies between 275-325 and 196-215 seconds for sequential read and write, respectively, so synchronization between processes is not a relevant issue. For random access, write varies substantially between 300 and 710 and read between 920 and 2200s. But as we will see, this has no impact on the results.



(a) Timeline for the individual writes **(b)** Density plot of the observations

Fig. 3. Results for one write run with sequential 256 KiB accesses (off0 mem layout).

methodology we can identify the relevant performance factors and then analyze them further to find their cause.

This demonstrates that the proposed strategy is not only applicable to Lustre but other systems as well.

6 Building Models to Predict Causes

Each model is built following the general algorithmic approach described in Section 4. This requires to treat measurements for a fixed configuration by 1) determine density of the measurements, 2) locate extrema in the density, 3) locate the limit for outliers. Then, for the expected system state, i.e., discard, cached or uncached, linear models are build individually to predict performance for these system states.

In this paper, linear models for four different performance factors based on the system state are investigated: **discard** and **cached** behavior, and, orthogonal to the caching, if the memory layout is random (**rnd**) or cached (**off0**). Since it turns out that the analysis of the Lustre file system is much more complex than for a local file system, only the benchmark runs with fixed position (off-0) are investigated. In fact, on our system the Lustre client side cache seems to use a write-through policy allowing this experiment already to reveal interesting aspects.

The model for discard is built by using the measurements obtained when reading from /dev/zero and writing to /dev/null. For the cached model, regular I/O is used and the file completely fits into page cache, for reads, the complete file is pre-read into main memory to make sure it is available in page cache. Further details about the configuration and benchmark are provided in Section 5.2. The two variants of the memory access pattern are that the benchmark reads/writes data to a 1 GiB random buffer (rnd) or overwriting data in a fixed buffer (off0) that may be cachable in CPU cache. If none of these models apply, i.e., the observed time is higher than predicted from random memory access to a random file position, data is classified as **uncached**. Those operations involve additional server communication and disk I/O.

Technically, each model is built using the statistics tool R applying its `density()` function on logarithmic durations[9]. The identification of the extrema is encoded in an algorithm that identifies them in the density graph. Additionally, the cutoff point for outliers is selected when the density is below $0.05 \cdot$ last cluster maximum.

Normally, it would be necessary to treat each individual cluster as another I/O-path. However, analysis of the density in these cases showed that the distance between the clusters is small. For example, see again Figure 2; with sequential I/O there are two clusters one at 0.12 ms and one at 0.47 ms, but with random I/O the spread is much larger. Since, in this paper, we are interested in major factors, we do not separate these minor cases further. The linear models are built using the `lm()` function and fitting the determined outlier limit for each size, adding 10% to this value as they vary between the sizes [10]. The y intercept for the model for large access sizes is fixed to the predicted value of the small model for 4096 bytes[11]. Therefore, we assume that data points below the estimate are explained by the model.

6.1 Understanding the Models

The parameters for the linear models and their accuracy on their training data are given in Table 3. The table includes the average, absolute and relative prediction error (in %), and the percentage of outliers that are purged when using the

[9] It applies the Gaussian kernel; the nrd0 bandwidth estimator and adjustment factor 2 is used to avoid creation of too many clusters.

[10] Again, to distinguish minor factors, one would have to build a linear model for each cluster.

[11] As said before, using just a single model across all access sizes is suboptimal for predicting the full range of access granularities since Lustre performs I/O in the granularity of the page size.

Case	Model parameters		Abs. err.	Rel.err	Outlier%	
+ mem location	Intercept I	Factor c	in s	in %	Exp.	Model
discard R$\leq 4K$off0	2.94e-07	4.41e-11	2.99e-08	10.4	0.7	0.1
discard R$> 4K$off0	4.75e-07	7.67e-11	2.21e-05	27.4	2.1	0.2
discard W$\leq 4K$rnd	3.28e-07	2.25e-12	3e-08	10.1	0.8	0.1
discard W$> 4K$rnd	3.37e-07	2.94e-14	1.47e-07	53.5	2.8	0.0
cached R$\leq 4K$ off0	8e-06	4.18e-10	7.57e-07	10.0	3.1	1.0
cached R$> 4K$ off0	9.71e-06	1.44e-10	2.46e-05	14.0	1.8	**10.2**
cached R$\leq 4K$ rnd	8.39e-06	4.91e-10	7.97e-07	10.1	2.8	0.9
cached R$> 4K$ rnd	1.04e-05	2.99e-10	9.7e-05	15.5	2.2	0.1
cached W$\leq 4K$ off0	1e-05	3.26e-10	9.36e-07	10.1	1.6	0.7
cached W$> 4K$ off0	1.14e-05	3.93e-10	2.28e-05	18.3	1.2	**30.0**
cached W$\leq 4K$ rnd	1.11e-05	7.16e-10	1.97e-06	18.4	1.5	5.0
cached W$> 4K$ rnd	1.4e-05	5.87e-10	3.28e-05	23.6	0.6	**26.0**

Table 3. Parameters for the linear model $p(s) = I + c \cdot s$ and error metrics. The file locality is always off0.

density-based outlier removal. These are not explained by the model (because their duration is higher than the model prediction). For cached write, the linear model showed a substantial number of outliers which are accesses with 4, 8 and 16 MiB size. It turned out that in this cases performance varies significantly by two orders of magnitude (their mean performance is also substantially lower than for 1 MiB, see Table 1) – this is a performance bug in Lustre. For this reason, the linear model for write has been built based on the results up to 2 MiB access size. The resulting model correctly detects these unexpectedly slow data points as outliers and, thus, increases the percentage of outliers in the table.

For illustrative purpose, the measurements and model predictions for the read model with cached data (off0 memory and file layout) are shown in Figure 4. Note that this model is expected to behave like a memcpy as data is available in the page cache and must only be copied to the user buffer that is stored in CPU cache as long as it fits there. Figures for the other three models look similar. Adding 10% to the value for the fitting has caused the shift in the left figure (and adds naturally 10% to the relative error reported in the table).

How the four models for reads split the observed durations into their likely causes is shown in Figure 5. The figure includes models for Lustre's cached behavior and for the node-local data transfer between kernel-space and user-space buffer (discard models). The relevant performance factor is given by the model that predicts the smallest duration still larger than the observation. By comparing the models for cached-off0 with cache-rnd, it can be observed that for the Lustre data and for small access sizes, both lines are similar. Therefore, the memory location of the user space buffer does not matter and, thus, cannot be predicted. For larger accesses, the discard-rnd case behaves similar to the cached-off0 case that is measured with Lustre, thus cached I/O with Lustre to a fixed (theoretically in CPU cache fitting buffer) behaves like a memcpy from kernel space to a random memory buffer. Since Lustre's data is fully cached

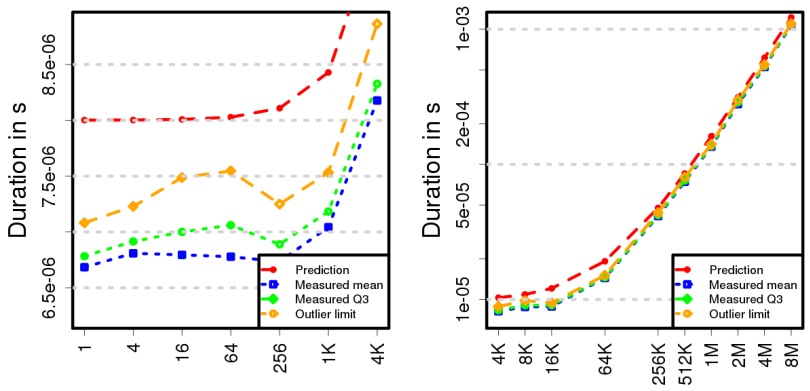


Fig. 4. Model accuracy for reading cached data (off0 locality in memory and file). Other figures look similar.

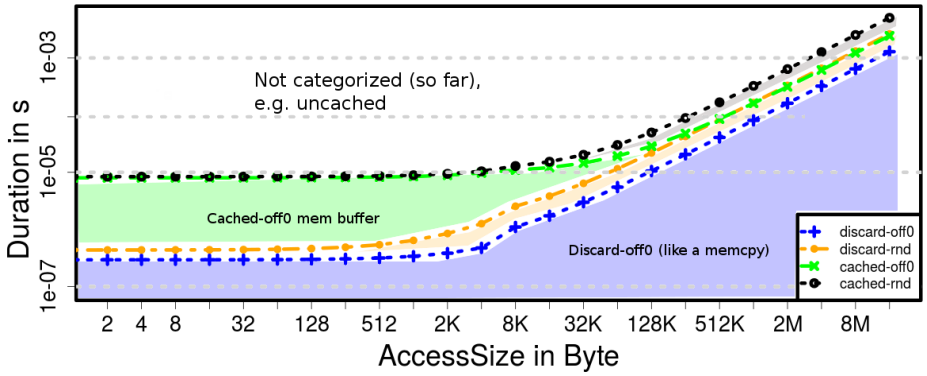


Fig. 5. Read models predicting caching and memory location.

in the page cache, required duration should go down to the discard-off0 case. This leads to the conclusion that there is still room for improvement in Lustre's I/O path even for large accesses. For smaller access granularities the situation is much worse as cached data in Lustre is more than an order of magnitude slower than reading from the pseudo device.

6.2 Applying the Model

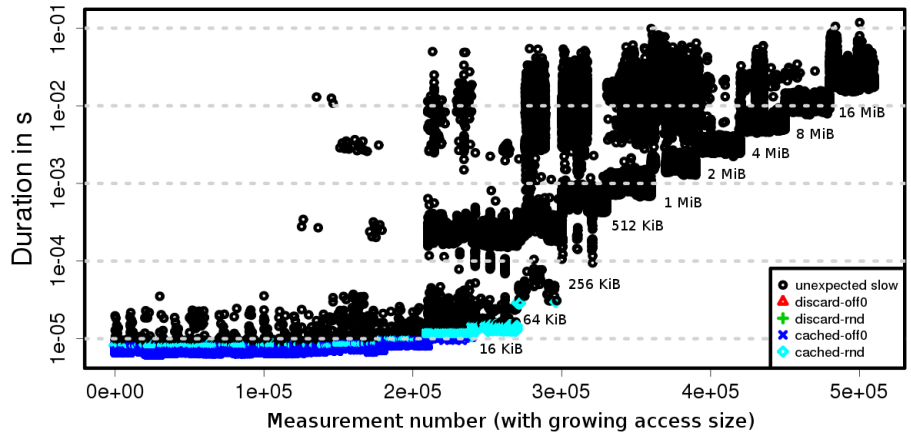


Fig. 6. Timelines for reverse reading files with various access sizes into a seq. buffer and identified classes. 10k repeats are performed per size and all three repeats are shown – the figure is sorted by access size.

Experiment state-mem-file	uncached	discard off0	discard rnd	cached off0	cached rnd
D-reverse-off0 R	0.004	46	54	0.3	0.03
C-off0-off0 R	0.29	0	34	60	6.1
C-seq-off0 R	0.31	0	0	52	47
C-seq-reverse R	54	0	0	42	4.3
C-seq-rnd8 R	26	0	0	30	44
C-seq-rnd R	68	0	0	26	5.6
C-seq-seq R	42	0	0	48	9.5
C-seq-stride8,8 R	63	0	0	28	8.8
C-off0-rnd R	80	0	2e-04	18	1.9
U-off0-rnd R	100	0	0	0.01	0.15
U-seq-seq R	37	0	0	57	6.1
C-off0-rnd W	100	0	0	0	0.003
C-off0-seq W W	42	0	0	40	17
C-seq-seq W	48	0	0	40	12
C-off0-reverse W	15	0	0	71	14

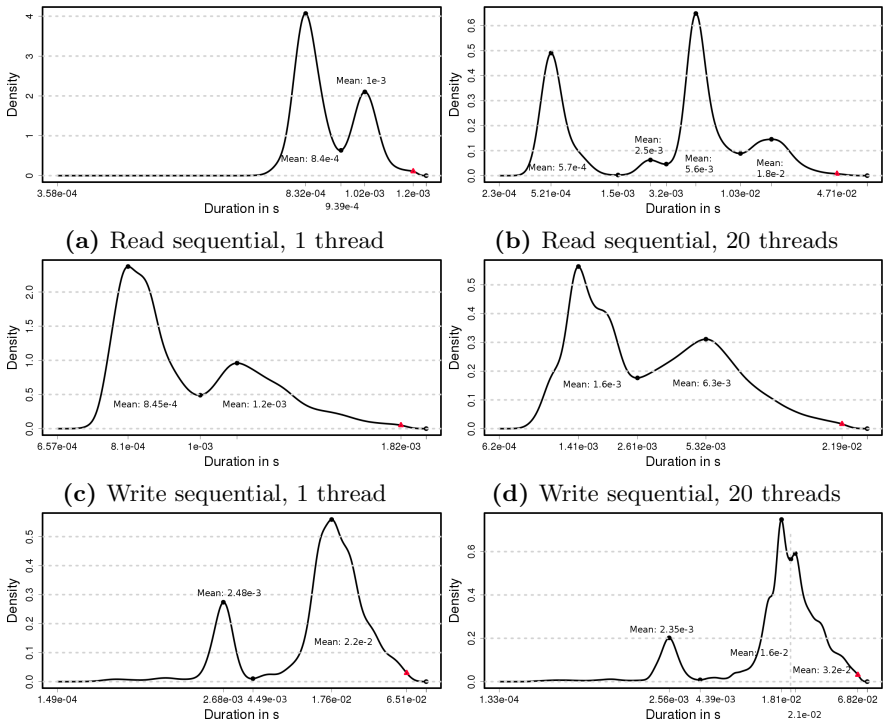
Table 4. Model predictions classes in % of data points for selected memory & file locations – access size is varied.

Now we apply the previously trained models on data that uses different access patterns. This allows us to automatically determine the relevant performance factors for each measurement. Moreover, unexpectedly slow operations – in terms of the models – are also identified; we classify them as uncached for the moment. Technically, the time for each measurement is compared with the prediction of all models and the model that predicts a higher runtime is choosen. The illustration in Figure 5 helps to understand this process. Assume we do an 32 KiB read operation, if its runtime is below the discard-off0 line it behaves like a page-cache copy, to a re-used buffer, i.e., in CPU cache. If it is above this line but below the cached-off0 line, it is classified as discard to a random memory buffer. Next, it behaves like cached I/O on Lustre or becomes uncached.

Table 4 gives an excerpt of the classification results of several access patterns where we know the system state. For example, when applying the models to all observations with sequential memory locality and reverse file access, the individual operations are classified as shown in the timeline of Figure 6. It can be seen that up to 16 KiB access granularity many data points are identified as cached and then not any more. In fact, over the full data set, 42% are classified to be cached (i.e., almost all small accesses) with data fitting in cache and 4.3% as cached with random position in memory. The reason is simple: as read-ahead only works with increasing offsets and we are reading from back to front, larger access granularities are not likely to be cached anywhere in the system. However, data is fetched in larger granularity and at page size boundaries, allowing small accesses to benefit from previously fetched data.

Inspecting the results from Table 4 further, it can be seen that the method is able to automatically classify the I/O operations in the experiments. Based on our a-priory knowledge of the access patterns, it achieves a high accuracy and still reveals interesting results. For example, for random file layout (U-off0-rnd

R and C-off0-rnd W) nearly all operations correctly are classified as uncached and thus hit the server. When data is in the read cache (C-off0-rnd R), still 80% behave like they trigger reads from the server. Similarly, in the cases for C-off0-off0 R and C-seq-off0 R, it is correctly identified that all data is served from the cache and many operations can be correctly classified to stem from a random memory location or like the direct memcpy. Therefore, the method is effective to provide information if data is stemming from the page cache or the Lustre server and even provides some hint about the memory locations.



(a) Read sequential, 1 thread **(b)** Read sequential, 20 threads

(c) Write sequential, 1 thread **(d)** Write sequential, 20 threads

(e) Read random, 1 thread **(f)** Read random, 20 threads

Fig. 7. Density plots when accessing 1 TB of data with 1 MiB access granularity. The red triangle after the last cluster is the point after which all measurements are classified to be extremely slow.

Validation with parallel programs To show that the assumptions made are reasonable and also apply to concurrent processing as well; Figure 7 shows the density plots when accessing 1 TB of data. Similarly, clusters of data emerge and usually two major factors are visible. It can be seen that maxima and mean of several clusters remain regardless of the number of threads; the clusters are a bit shifted but still recognizable. Additionally, the case with 20 threads reveals

two additional minor performance factors – which are likely to be caused by interference between threads.

7 Summary & Outlook

In this paper, a density-based analysis of ensembles is performed; firstly, this allows to identify and remove extreme slow operations that distort the overall performance analysis. Those slow operations can be caused by concurrent I/O to the Lustre servers by other jobs or background daemons. Based on the cleaned data, four linear models are created that predict different memory locations in the user-space buffer but also decide if an I/O behaves like it is cached on the client side or if it is uncached. With this model, it can be decided if a measurement is likely to reside in memory and, thus, requiring only a memcpy. If it behaves like cached data in Lustre or if it is uncached. With this approach, for example, the interesting behavior of reverse reading a file became apparent and could be explained.

In the future, these models will be extended to cover also server-sided cases such as cached by the RAID system or fast, average and slow seeks of HDDs. Since the probability distributions of these cases vary significantly, additional techniques from statistics will be applied to classify them correctly.

The paper discussed the relevance of the method that helps to identify relevant factors but does not yet automatically identify them for a given system. However, using this approach, developers or administrators could focus to identify the cause behind the relevant performance factors. We will continue to implement a tool in SIOX or Darshan that uses these models to automatically assess client-side performance. By extracting accessible knowledge from the operating system, it will narrow down the likely cause for for the observed I/O performance. Additionally, users could feed in system knowledge to provide further insight about factors hidden from the operating system or client side. Once the tool is completed, it will allow the application developer to identify the potential for performance improvements, for example, if all I/O is supported by the client-side page cache, I/O is optimal. Moreover, the potential time loss of suboptimal I/O can be estimated by subtracting the estimated runtime for cached I/O from the actual observed runtime.

Acknowledgments

We thank Venka Michaela Zimmer for her review and Charlotte Jentzsch for the fruitful discussions about statistics.

Bibliography

[1] J. Axboe. Flexible I/O Tester.

[2] C. Bartz, K. Chasapis, M. Kuhn, P. Nerge, and T. Ludwig. A Best Practice Analysis of HDF5 and NetCDF-4 Using Lustre. In *High Performance Computing*, pages 274–281. Springer, 2015.

[3] J. Borrill, L. Oliker, J. Shalf, and H. Shan. Investigation of leading HPC I/O performance using a scientific-application derived benchmark. In *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–12. IEEE, 2007.

[4] M. J. Brim and J. K. Lothian. Monitoring Extreme-scale Lustre Toolkit. *arXiv preprint arXiv:1504.06836*, 2015.

[5] A. D. Brunelle and J. Axboe. Blktrace user guide, 2007.

[6] N. C. Burnett, J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Management. In *USENIX Annual Technical Conference, General Track*, pages 29–44, 2002.

[7] P. H. Carns, R. Latham, R. B. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 characterization of petascale I/O workloads. In *Proceedings of the First Workshop on Interfaces and Abstractions for Scientific Data Storage*, New Orleans, LA, USA, Sept. 2009.

[8] T. E. Denehy, J. Bent, F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Deconstructing Storage Arrays. In *ACM SIGARCH Computer Architecture News*, volume 32, pages 59–71. ACM, 2004.

[9] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[10] A. Ha. Modelling parallel access to shared resources in a distributed file system using queueing networks. *J. Syst. Softw.*, 6(1-2):61–69, May 1986. ISSN 0164-1212. doi: 10.1016/0164-1212(86)90024-5. URL `http://dx.doi.org/10.1016/0164-1212(86)90024-5`.

[11] Y. A. Ilker Yalcin. Nonlinear factor analysis as a statistical method. *Statistical Science*, 16(3):275–294, 2001. ISSN 08834237. URL `http://www.jstor.org/stable/2676693`.

[12] W. Jin, A. K. Tung, J. Han, and W. Wang. Ranking outliers using symmetric neighborhood relationship. In *Advances in Knowledge Discovery and Data Mining*, pages 577–593. Springer, 2006.

[13] M. P. Kasick, J. Tan, R. Gandhi, and P. Narasimhan. Black-Box Problem Diagnosis in Parallel File Systems. In *FAST*, pages 43–56, 2010.

[14] J.-O. Kim and C. W. Mueller. *Introduction to factor analysis: What it is and how to do it*. Number 13. Sage, 1978.

[15] S. J. Kim, Y. Zhang, S. W. Son, M. Kandemir, W.-K. Liao, R. Thakur, and A. Choudhary. Iopro: A parallel i/o profiling and visualization framework for high-performance storage systems. *J. Supercomput.*, 71(3):840–870, Mar.

2015. ISSN 0920-8542. doi: 10.1007/s11227-014-1329-0. URL http://dx.doi.org/10.1007/s11227-014-1329-0.

[16] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The Vampir Performance Analysis Tool-Set. In *Tools for High Performance Computing, Proceedings of the 2nd International Workshop on Parallel Tools*, pages 139–155. Springer, 2008.

[17] J. Kunkel. *Simulation of Parallel Programs on Application and System Level*. Phd thesis, Universität Hamburg, 07 2013. URL http://ediss.sub.uni-hamburg.de/frontdoor.php?source_opus=6264.

[18] J. Kunkel. Simulating parallel programs on application and system level. *Computer Science - Research and Development*, pages 167–174, 05 2013. URL http://link.springer.com/article/10.1007/s00450-012-0208-2.

[19] J. Kunkel, M. Zimmer, N. Hübbe, A. Aguilera, H. Mickler, X. Wang, A. Chut, T. Bönisch, J. Lüttgau, R. Michel, and J. Weging. The SIOX Architecture – Coupling Automatic Monitoring and Optimization of Parallel I/O. In *Supercomputing*, pages 245–260. ISC events, Springer International Publishing, 2014. ISBN 978-3-319-07517-4. doi: http://dx.doi.org/10.1007/978-3-319-07518-1_16.

[20] J. M. Kunkel and T. Ludwig. Performance evaluation of the PVFS2 architecture. In *Parallel, Distributed and Network-Based Processing, 2007. PDP'07. 15th EUROMICRO International Conference on*, pages 509–516. IEEE, 2007.

[21] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–11. IEEE, 2012.

[22] T. Ludwig, S. Krempel, M. Kuhn, J. Kunkel, and C. Lohse. Analysis of the MPI-IO optimization levels with the PIOViz Jumpshot enhancement. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 213–222. Springer, 2007.

[23] W. D. Norcott and D. Capps. IOZone filesystem benchmark.

[24] A. Patcha and J.-M. Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer networks*, 51 (12):3448–3470, 2007.

[25] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, et al. The Structural Simulation Toolkit. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):37–42, 2011.

[26] H. Shan and J. Shalf. Using IOR to analyze the I/O performance for HPC platforms. *Lawrence Berkeley National Laboratory*, 2007.

[27] S. S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2): 287–311, 2006.

[28] R. Singhal, M. Nambiar, H. Sukhwani, and K. Trivedi. Performability Comparison of Lustre and HDFS for MR Applications. In *Software Reliability*

Engineering Workshops (ISSREW), 2014 IEEE International Symposium on, pages 51–51. IEEE, 2014.

[29] A. Uselton. Deploying server-side file system monitoring at NERSC. *Lawrence Berkeley National Laboratory*, 2009.

[30] A. Uselton, G. Paciucci, and J. Xiong. Demonstrating the Improvement in the Performance of a Single Lustre Client from Version 1.8 to Version 2.6.

[31] A. Uselton, M. Howison, N. J. Wright, D. Skinner, N. Keen, J. Shalf, K. L. Karavanic, and L. Olike. Parallel I/O performance: From events to ensembles. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–11. IEEE, 2010.

[32] K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth. Scalable I/O Tracing and Analysis. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 26–31, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-883-4. doi: 10.1145/1713072.1713080. URL `http://doi.acm.org/10.1145/1713072.1713080`.

[33] B. Worthington, G. Ganger, and Y. Patt. The disksim simulation environment. *University of Michigan, EECS, Technical Report CSE-TR-358*, 98, 1998.

[34] N. J. Yadwadkar, C. Bhattacharyya, K. Gopinath, T. Niranjan, and S. Susarla. Discovery of Application Workloads from Network File Traces. In *FAST*, pages 183–196, 2010.