Analysis of Input/Output Phenomena using Statistics and Machine Learning

Julian M. Kunkel

2017-05-31

Outline

High-Performance Computing (HPC)

Definitions

- HPC: Field providing massive compute resources for a computational task
  - Task needs too much memory or time on a normal computer
  - $\Rightarrow$ Enabler of complex scientific simulations, e.g., weather, astronomy
- Supercomputer: aggregates power of 10,000 compute devices
- File system: provides a hierarchical namespace and “file” interface
- Parallel I/O: multiple processes can access distributed data concurrently

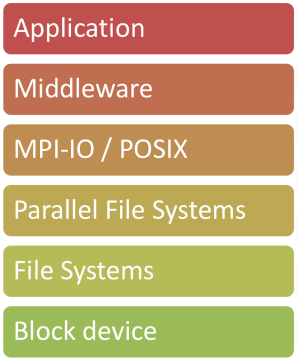
Example Supercomputer of DKRZ: Mistral

- Compute: 3,000 dual socket nodes
- Storage: 52 Petabyte (ca. 10,000 HDDs, 300 I/O servers)
- Cost for I/O system: 6 Million Euro

The I/O Stack

- Parallel application
  - Is distributed across many nodes
  - Has a specific access pattern for I/O
  - May use several interfaces
    File (POSIX, ADIOS, HDF5), SQL, NoSQL
- Middleware provides high-level access
- POSIX: ultimately file system access
- Parallel file system: Lustre, GPFS, PVFS2
- File system: EXT4, XFS, NTFS
- Operating system: (orthogonal aspect)

The layers provide optimization strategies and tunables



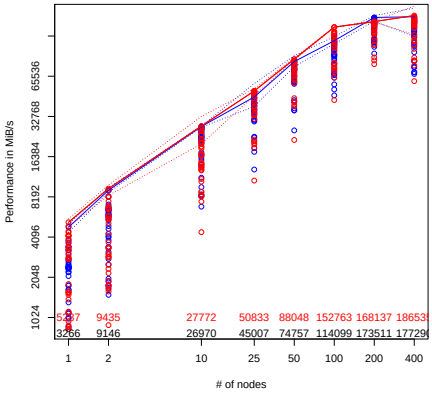
Example I/O stack

Parallel I/O

- I/O intense science requires good I/O performance
- DKRZ file systems offer about 700 GiB/s throughput
  - However, I/O operations are typically inefficient: Achieving 10% of peak is good
  - Unfortunately, prediction of performance is barely possible
- Influences on I/O performance
  - Application's access pattern and usage of storage interfaces
  - Communication and slow storage media
  - Concurrent activity – shared nature of I/O
  - Tunable optimizations deal with characteristics of storage media
  - Complex interactions of these factors
- The I/O hardware/software stack is very complex – even for experts
- Chances for machine learning and statistics
  - Diagnosing causes
  - Predicting performance, identification of slow performance
  - Prescribing tunables/settings

Illustration of Performance Variability

- Best-case benchmark: optimal application I/O
  - Independent I/O with 10 MiB chunks of data
  - Real-world I/O is sparse and behaves worse
- Configurations vary:
  - Number of nodes the benchmark is run
  - Processes per node
  - Tunable: stripe size, stripe count
  - Read/Write accesses
- Optimal performance:
  - Small configuration: 6 GiB/s per node
  - Large configurations: 1.25 GiB/s per node
- Best setting depends on configuration!

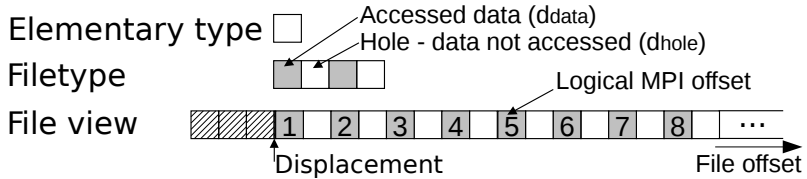


A point represents one configuration

Example Pattern: Non-Contiguous Data Access

- Parallel applications distribute data across processes
- Each process access only parts of the file
  - Skipped parts are called “holes”
- Data-sieving optimization
  - Ignores (small) holes and reads all data; discard unnecessary data
  - But: Complex to parameterize as benefit depends on hardware and pattern

Example non-contiguous access pattern



Outline

I/O Modeling and Diagnosing Causes with Statistics

Issue

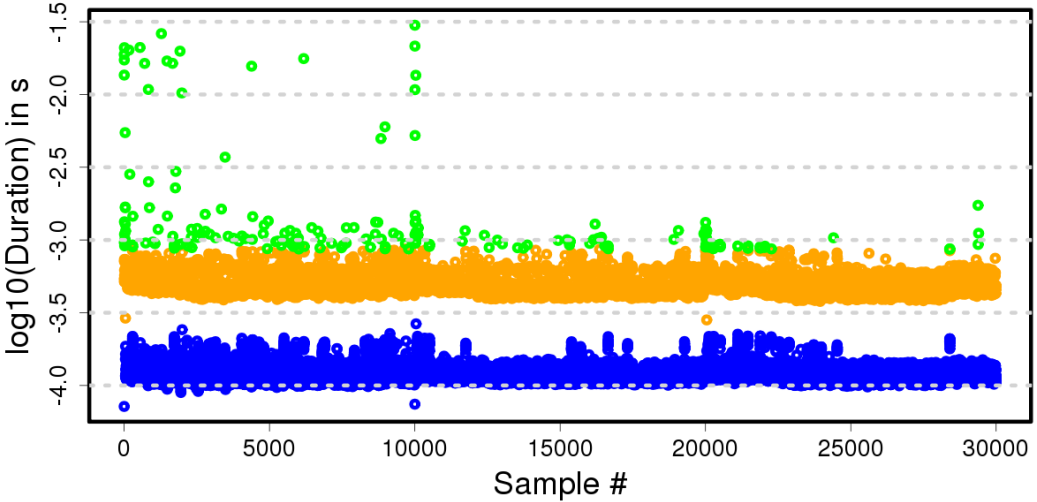
- Measuring the same operation repeatedly results in different runtime
- Reasons:
  - Sometimes a certain optimization is triggered, shortening the I/O path
  - Example strategies: read-ahead, write-behind
- Consequence: Non-linear access performance, time also depends on access size
- It is difficult to assess performance of even repeated measurements!

Goal

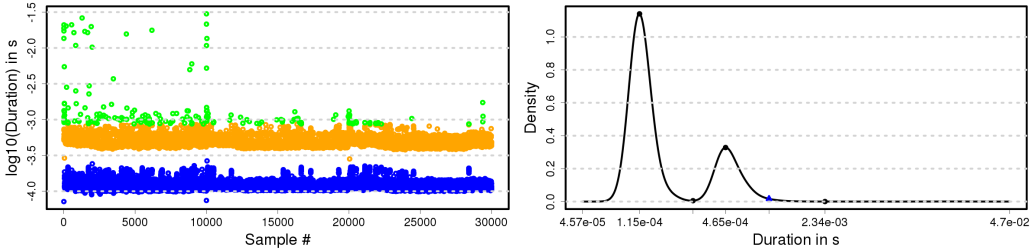
- Predict likely reason/cause-of-effect by just analyzing runtime
- Estimate best-case time, if optimizations would work as intended

Example: Measuring Sequential Read

Duration for individual runtime – 256 KiB accesses (off0 mem layout)



Comparing Density Plot with the Individual Data Points

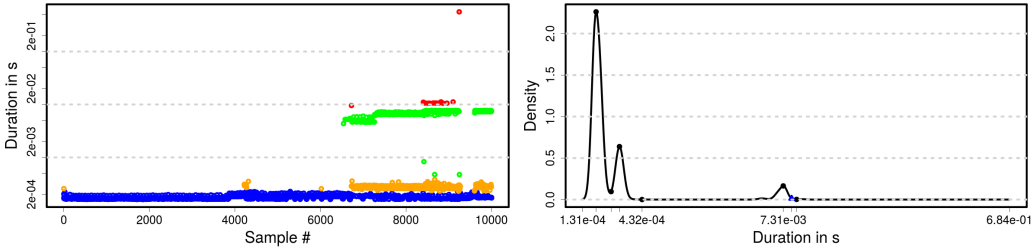


Duration for sequential reads with 256 KiB accesses (off0 mem layout)

Algorithm for determining classes (color schemes)

- Create density plot with Gaussian kernel density estimator
- Find minima and maxima in the plot
- Assign one class for all points between minima and maxima
- Rightmost hill is followed by cutoff (blue) close to zero $\Rightarrow$ outliers (unexpected slow)

Write Operations



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

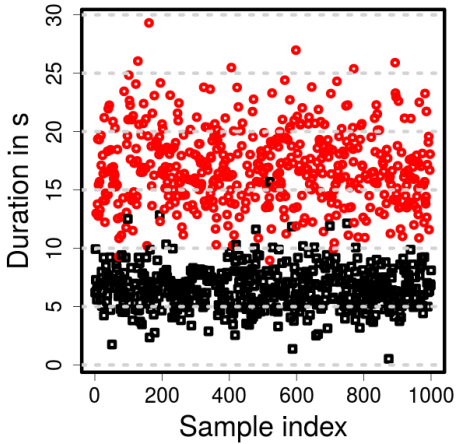
Known optimizations for write

- Write-behind: cache data first in memory, then write back
- Write back is expected to be much slower

This behavior can be seen in the figure !

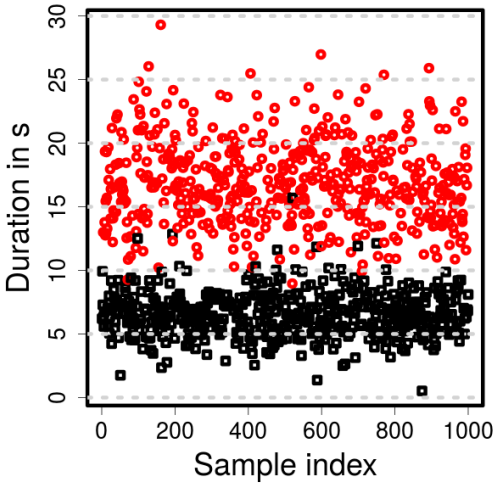
Simulation of this Behavior

- Assume we have two components
  - Component A is faster than B
  - Either A or B transfer data
  - Cache miss of A leads to transfer for B
- Overlaying 3 stochastic processes:
  - Two gamma distributions with scale=1
  - Normal distribution (little impact)

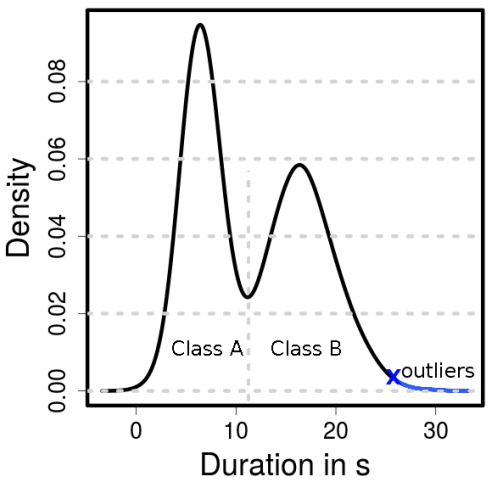


Resulting time for 1000 data points

Simulated Access Time and Resulting Density



(a) Timeline



(b) Density reveals two classes

Approach

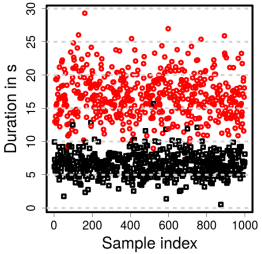
Assumptions

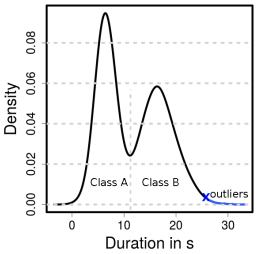
- Each “class” is caused another optimization/technology
  - Assign an observation to the likely class
  - This may lead to (tolerable) errors
- Behavior not visible on the density plot is irrelevant

⇒ The strategy identifies relevant “performance factors”

Concept

1. Repeatedly measure time for I/O with a given size
2. Construct the density graph and identify clusters
3. A class is caused by (at least) one performance factor
4. Build a model to assign the cluster across “sizes“
5. Optional: Identify the root cause for the cluster
6. Assign appropriate names, e.g., “client-side cached”





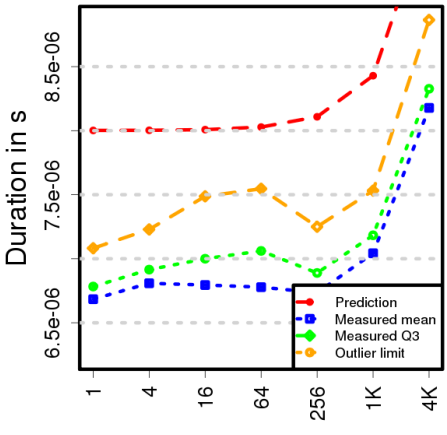
Approach: Models

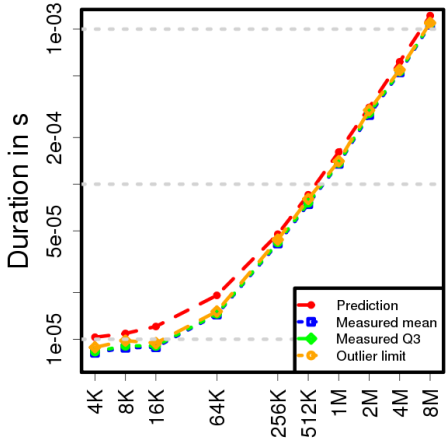
- Apply a family of linear models predicting time; $lm(size) = c + f(size)$
  - Assume time correlates to the operation's size
  - Each model represents a condition C (cached, in L1, ...)
  - $t_C(size) = lm(size) + lm'(size) + ...$ and check $min(|t_C - \hat{t}|)$
- Assume the conditions for the closest combination are the cause
- Ignore the fact of large I/O requests with mixed conditions
  - i.e., some time of the operation may be caused by C and some by C'

Example models

- $t(size) = m$: Data is discarded on the client or overwritten in memory
- $t(size) = m + c(size)$: Data is completely cached on the client ...

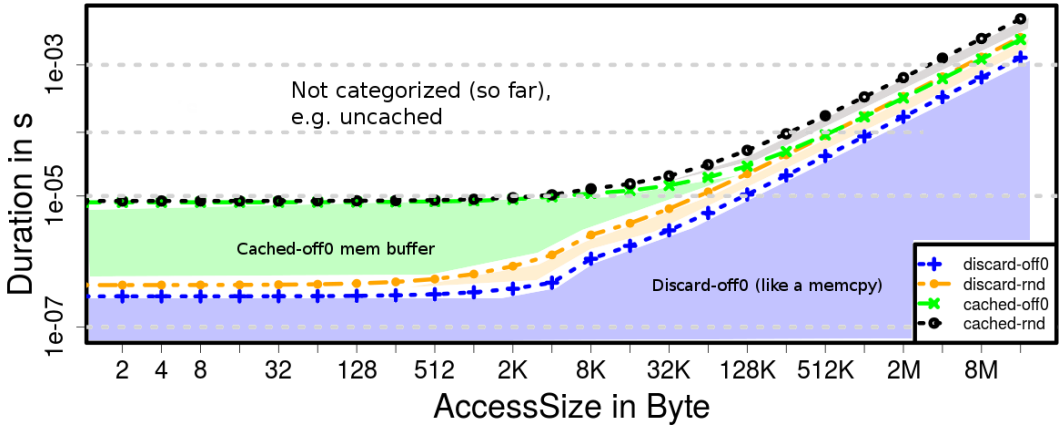
Model for Reading Cached Data





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

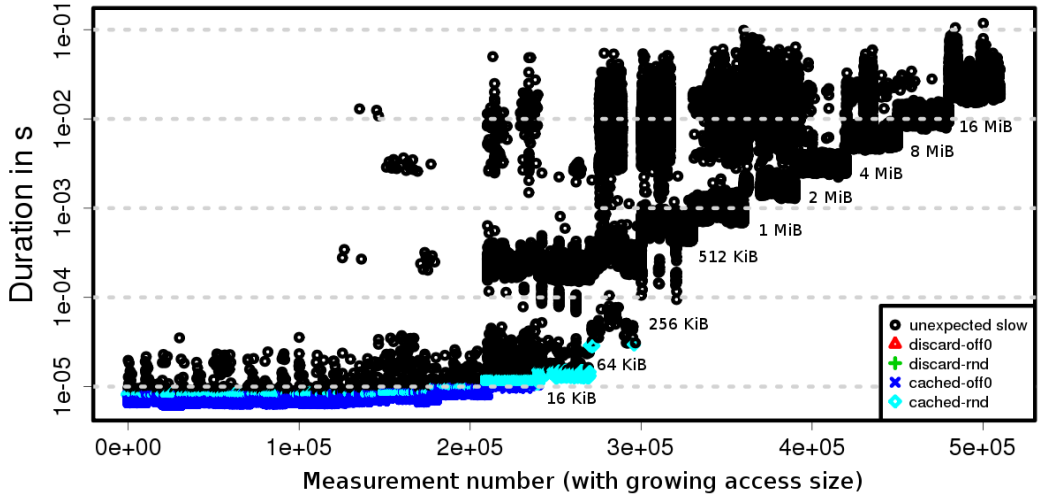
Resulting Performance Models for Read



Read models predicting caching and memory location.

Using the Model to Identify Anomalies

Using the model, the figure for reverse access shows slow down (by read-ahead)



Validation: Classify Different Patterns

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

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

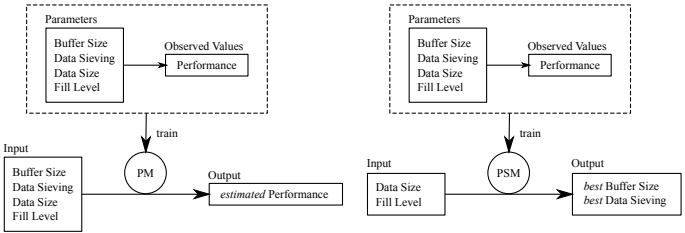
Outline

Predicting Non-Contiguous I/O Performance

Goal: Predict storage performance based on several parameters and tunables

Alternative models

- Predict performance based on parameters
- Predict best (data sieving) settings



(a) Performance Model (b) Parameter Setting Model

PM provides a perf. estimate, whereas PSM provides the “tunable” variable parameters to achieve it

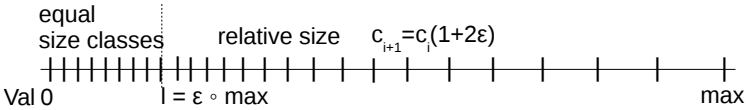
Transformation of the Problem

- Aim to apply alternative methods from machine learning
- Many require classification problems instead of regression

⇒ Performance values need to be mapped into classes

Mapping

- Create 10 classes with the same length up to 5% of max. performance
- Then increase performance range covered by 10% each



Evaluation Data

We analyzed the validity of the approach on two systems

System 1: WR cluster

- Lustre 2.5
- 10 server nodes
- 1 Gb Ethernet
- 1 client node (max performance 110 MiB/s)

System 2: DKRZ porting system

- Lustre 2.5 provided by Seagate ClusterStor 9000
- 2 servers
- FDR-Infiniband
- 1 client node (max performance 800 MiB/s)

Classification Tree: Validation on Data of the WR Cluster

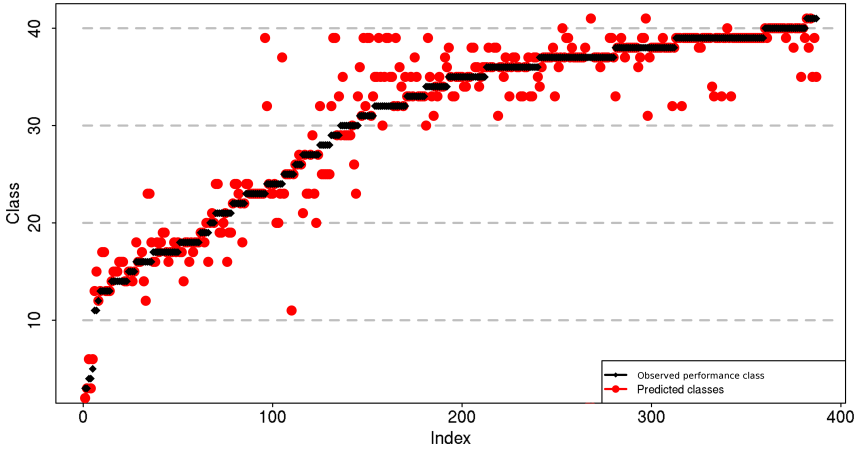
- Apply k-fold cross-validation
  - Split data into training set and validation set
  - Train model with all (k-1) folds and evaluate it on 1 fold
  - Repeat the process until all folds have been predicted
- A baseline model is the arithmethic mean performance (54.7 MiB/s)
  - Achieves an arithmethic mean error of 28.5 MiB/s
- Linear models yield a mean error of $\geq$ 12.7 MiB/s

Classification tree results

k	Performance errors in MB/s			Class errors		
	min	mean	max	min	mean	max
2	6.74	6.80	6.87	1.46	1.59	1.72
4	5.19	6.25	6.92	0.94	1.34	1.72
8	4.67	5.66	6.77	0.87	1.19	1.62

Prediction errors for training sets under *k*-fold cross-validation. Min & max refer to the folds' mean error. Values for k=3..7 lie in between

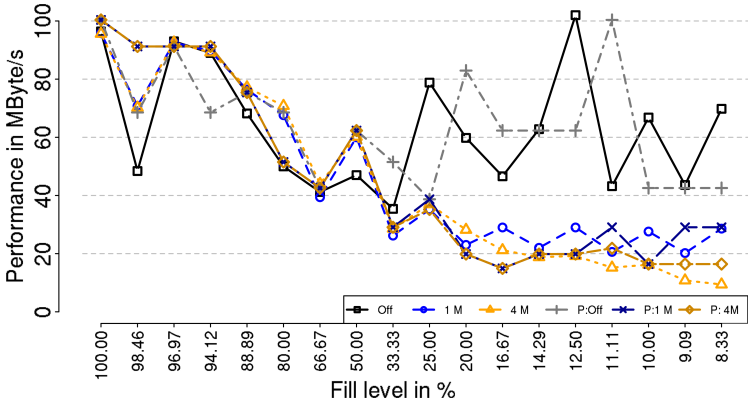
Comparing Prediction with Observation



Performance classes and error for k=2, sorted by the observed performance class. Trained by 387 instances, validated on the other 387 instances.

Comparing Prediction with Observation

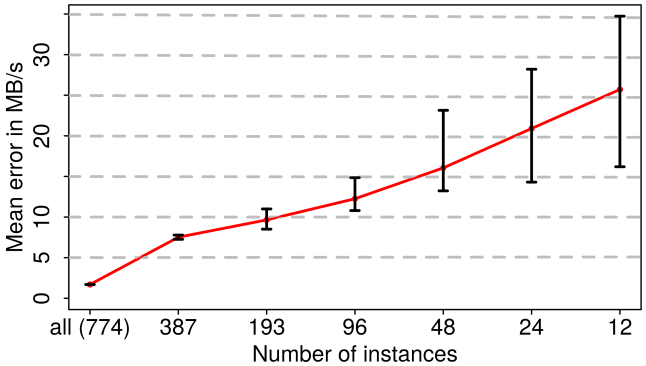
- Mispredictions due to sparse training data
- Non-linear performance behavior causes errors



Performance prediction for $d_{\text{data}} = 256\,\text{KiB}$, 387 instances

Investigating Training Set Size

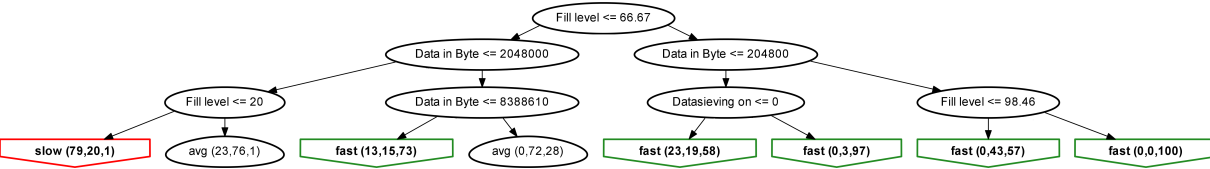
- Inverse k-fold validation: learn from 1 fold and test on (k-1)
- With $\geq$ 96 instances better than the linear model



Mean prediction error of PM by training set size under inverse *k*-fold cross-validation. Class prediction errors show similar behavior

Extracting Knowledge

- Rules can be easily extracted from decision trees
- Consider a performance prediction in three classes
- Rules (this is common sense for I/O experts)
  - Small fill levels and data sizes are slow
  - Large fill levels achieve good performance
- Surprising anomaly: smaller fill level, large access sizes are slower than medium



First three levels of the CART classifier rules for three classes slow, avg, fast ($[0, 25]$, $(25, 75]$, > 75 MB/s). The dominant label is assigned to the leaf nodes – the probability for each class is provided in brackets.

Outline

Prescriptive Analysis: Learning Best-Practises for DKRZ

- Performance benefit of I/O optimizations is non-trival to predict
- Non-contiguous I/O supports data-sieving optimization
  - Transforms non-sequential I/O to large contiguous I/O
  - Tunable with MPI hints: enabled/disabled, buffer size
  - Benefit depends on system AND application
- Data sieving is difficult to parameterize
  - What should be recommended from a data center's perspective?

Measured Data

- Simple single threaded benchmark, vary access granularity and hole size
- Captured on DKRZ porting system for Mistral
- Vary Lustre stripe settings
  - 128 KiB or 2 MiB
  - 1 stripe or 2 stripes
- Vary data sieving
  - Off or On (4 MiB)
- Vary block and hole size (similar to before)
- 408 different configurations (up to 10 repeats each)
  - Mean arithmetic performance is 245 MiB/s
  - Mean can serve as baseline “model”

System-Wide Defaults

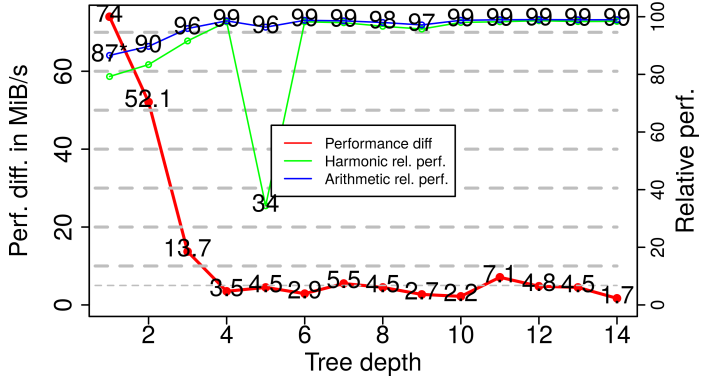
- Comparing a default choice with the best choice
- All default choices achieve 50-70% arithmethic mean performance
- Picking the best default default for stripe count/size: 2 servers, 128 KiB
  - 70% arithmetic mean performance
  - 16% harmonic mean performance ⇒ some choices result in slow performance

Default Choice			Best	Worst	Arithmethic Mean			Harmonic Mean	
Servers	Stripe	Sieving	Freq.	Freq.	Rel.	Abs.	Loss	Rel.	Abs.
1	128 K	Off	20	35	58.4%	200.1	102.1	9.0%	0.09
1	2 MiB	Off	45	39	60.7%	261.5	103.7	9.0%	0.09
2	**128 K**	**Off**	87	76	**69.8%**	209.5	92.7	8.8%	0.09
2	2 MiB	Off	81	14	72.1%	284.2	81.1	8.9%	0.09
1	128 K	On	79	37	64.1%	245.6	56.7	15.2%	0.16
1	2 MiB	On	11	75	59.4%	259.2	106.1	14.4%	0.15
2	**128 K**	**On**	80	58	**68.7%**	239.6	62.6	**16.2%**	0.17
2	2 MiB	On	5	74	62.9%	258.0	107.3	14.9%	0.16

Performance achieved with any default choice

Applying Machine Learning

- Building a tree with different depths
- Even small trees are much better than any default
- A tree of depth 4 is nearly optimal; avoids slow cases

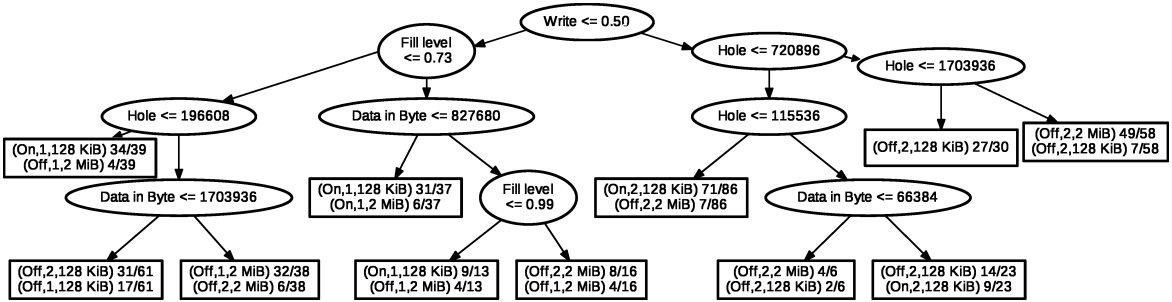


Perf. difference between learned and best choices, by maximum tree depth, for DKRZ's porting system

Decision Tree & Rules

Extraction of knowledge from a tree

- For writes: Always use two servers; For holes below 128 KiB $\Rightarrow$ turn DS on, else off
- For reads: Holes below 200 KiB $\Rightarrow$ turn DS on
- Typically only one parameter changes between most frequent best choices



Decision tree with height 4. In the leaf nodes, the settings (Data sieving, server number, stripe size) and number of instances for the two most frequent best choices

Outline

Determining Data Characteristics

- Data characteristics:
  - Proportion of a given (scientific) file format
  - Performance behavior when accessing file data
  - Compression characteristics (ratio, speeds)
- Understanding these characteristics is useful
  - Proportions of a file format to identify relevant formats
    - Starting point for optimization of format
  - Conducting what-if analysis
    - Estimate the influence storage compression has
    - Performance expectations when applying a new strategy
- Existing studies use a manual selection of “data” for representing stored data

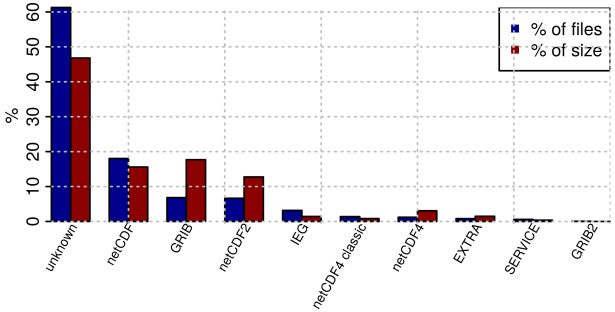
Representative Selection

- Analyzing large quantities of data is time consuming and costly
  - Scanning petabytes of data in $>$ 100 millions of files
  - With 50 PB of data and 5 GiB/s read, 115 node days are needed
  - Scanning DKRZ data with a few compression algorithms cost 4000 €

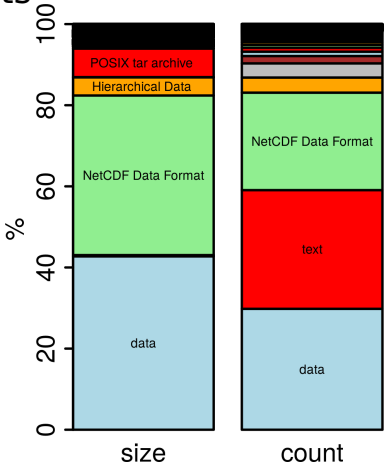
  $\Rightarrow$ Working on a representative data set reduces costs
- Conducting analysis on representative data is non-trivial
  - What data makes up a representative data set?
  - How can we infer knowledge for all data based on the subset?
    - Based on file number/count (i.e., a typical file is like X)
    - Based on file size (i.e., 10% of storage capacity is like Y)

Example: Determine Scientific File Formats

- Notice the difference by file count and capacity



(a) Scientific file types



(b) File types according to file magic

Contribution

Goal

- Design a method based of statistical sampling to estimate file properties
- Conduct a simple study to investigate compression and file types

Approach

1. Scanning a large fraction of data on DKRZ file systems
   - Analyzing file types, compression ratio and speed
2. Investigating characteristics of the data set Filetype, compression ratio, ...
3. Statistical simulation of sampling approaches
   - We assume the population (full data set) is the scanned subset
4. Discuss the estimation error for several approaches

Statistical Sampling

- Can we determine the error when analyzing only a fraction of data?
  - We simulate sampling by drawing samples from the totally analyzed files
- Statistics offers methods to determine confidence interval and sample size
- We analyze random variables for quantities that are continuous or proportions
  - Proportions: fraction of samples for which a property holds

Sample size and confidence intervals

- For proportions Cochran's sample size formula estimates number of samples
  - Number works for extremely large population sizes
  - Error bound $\pm 5\%$ requires 400 samples (95% confidence)
  - Error bound $\pm 1\%$ requires 10,000 samples
- For continuous variables
  - Models require to know the distribution of the value
  - A-priori unknown, usually not Gaussian, difficult to apply $\rightarrow$ out-of-scope (here)
  - Nevertheless, we will demonstrate convergence

Efficient Sampling Strategies

Sampling to Compute by File Count

1. Enumerate all files
2. Create a simple random sample
   - Select a random number of files to analyze without replacement
   - For proportional variables, the number of files can be computed with Cochran's formula

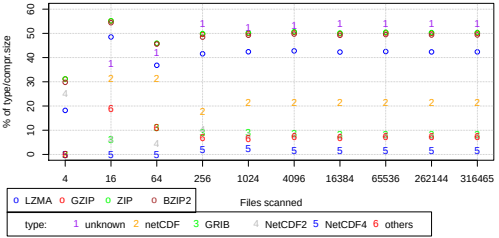
Sampling to Compute by File Size

1. Enumerate all files AND determine their file size
2. Pick a random sample based on the probability $\frac{filesize}{totalsize}$ with replacement
   - Large files are more likely to be chosen (even multiple times)
3. Create a list of unique file names and analyze them
   - Either scan full file (once) or measure feature on a random file section (chunk)
4. Compute the arithmetic mean for the variables
   - If a file has been picked multiple times in Step 2., its value is used multiple times

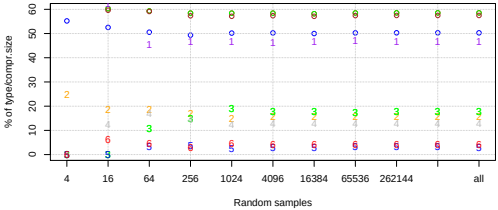
Demonstration of the Strategies

- Apply the approach with an increasing number of samples
  - Compare true value with the estimated value

Running one simulation for increasing sample counts



(c) Compute mean by count



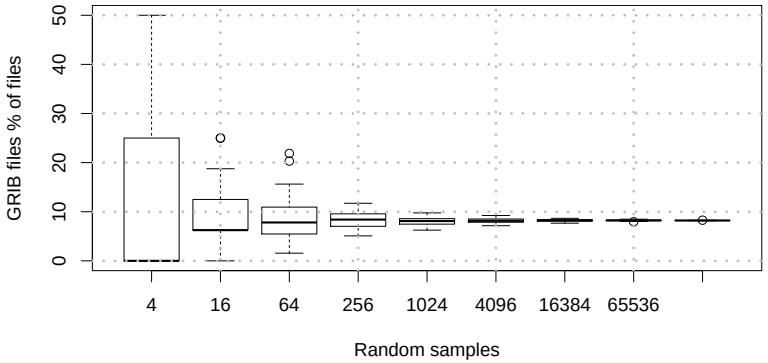
(d) Compute mean by size

Evaluating various metrics (proportions) for an increasing number of samples

- This suggests that the results converge quickly but how trustworthy is one run?

Investigating Robustness: Computing by File Count

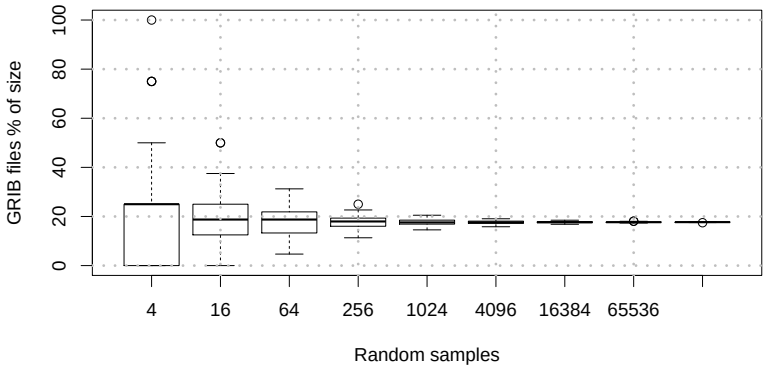
- Running the simulation 100 times to understand the variance of the estimate
- Clear convergence: thanks to Cochran's formula, the total file count is irrelevant



Simulation of sampling by file count to compute compr.% by file count

Investigating Robustness: Computing by File Size

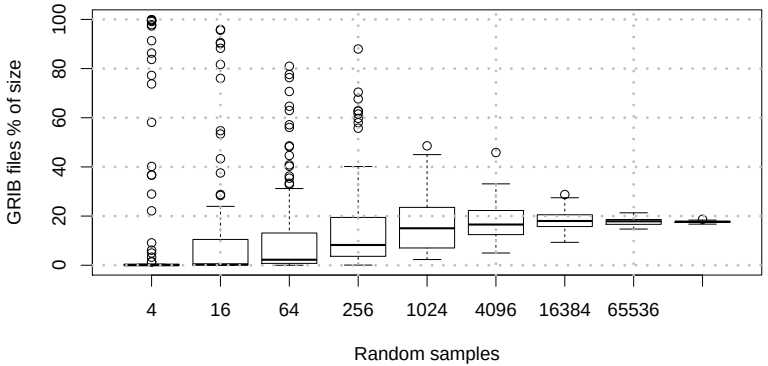
- Using the correct sampling by weighting probability with file size



Simulation of sampling to compute proportions of types by size

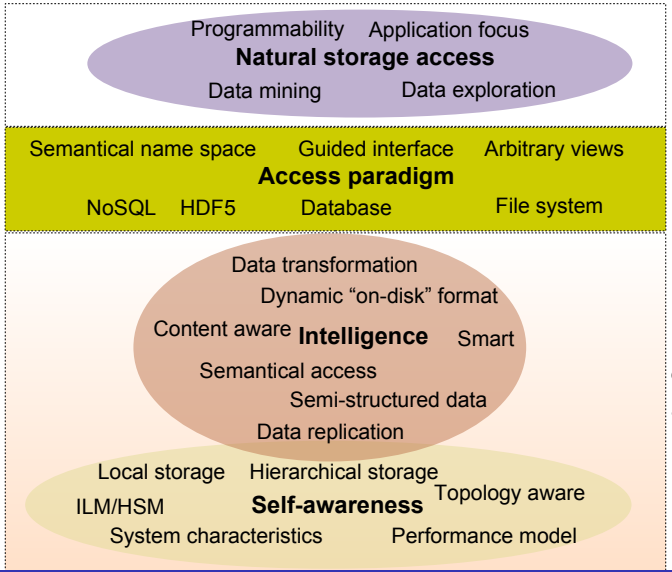
Investigating Robustness: Computing by File Size

- Using the WRONG sampling by just picking a simple random sample
- Almost no convergence behavior; you may pick a file with 99% file size at the end



Simulation of sampling to compute proportions of types by size

Personal Vision: Towards Intelligent Storage Systems and Interfaces



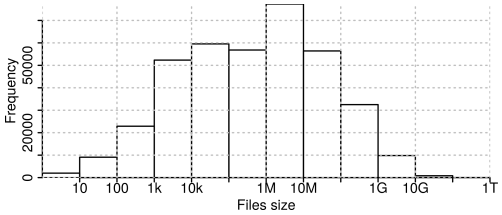
Summary

- Parallel I/O is complex and performance difficult to assess
- Statistics and machine learning help with key aspects:
  - Diagnosing causes and identify anomalies
  - Predicting performance
  - Prescribing best practices
- Additional: Sampling methods to estimate data characteristics

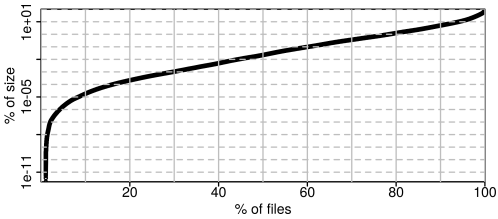
Appendix

Distribution of File Sizes

- For now, we analyze all scanned files!
- File size follows a heavy tailed distribution
- 90% of files consume roughly 10% capacity



(a) Histogram (logarithmic x-axis)



(b) Cumulative file sizes (y-axis in log scale)

Sampling of the Test Data

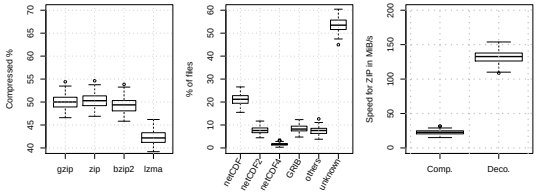
- DKRZ usage: 320 million files in 12 PB, 270 project dirs
- Scan of user accessible data (scan is done by a regular user)
  - Accessible data: 58 million files, 160 project dirs
- Scanned files: 380 k files (0.12%) in 53.1 TiB (0.44%) capacity
  - Discrepancy since home directories contain very small files

Scanning Process

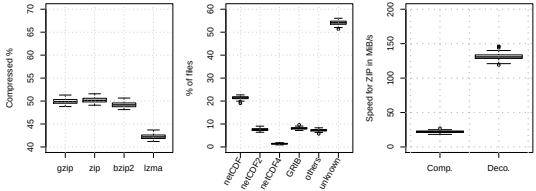
1. Run a find for each project directory, store it is a file
2. Select up to 10 k files from each project randomly (scan list)
3. Permutate the scan list
4. Partition the scan list into chunks (file lists)
5. Run multiple processes concurrently, each working on a file list
6. Terminate the processes after a couple of days

As we will see this approach is not optimal for analyzing by capacity

Additional Characteristics Computed by File Count

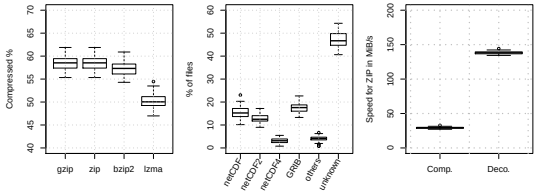


(c) Sampling 316 = 0.1% of files)

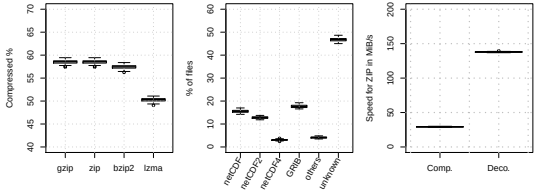


(d) Sampling 3164 = 1% of files)

Additional Characteristics Computed by File Size



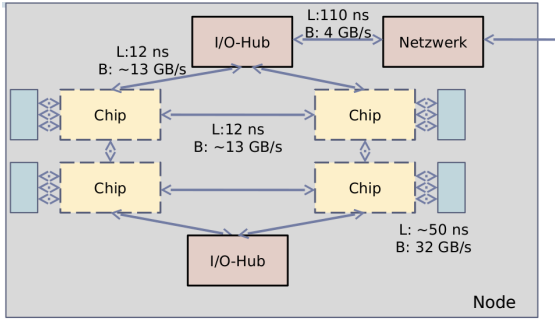
(e) Drawing 256 samples with replacement



(f) Drawing 4096 samples with replacement

System Model: Time costs in the I/O path

- $t(size) = m + p(size) + n(size) + c(size) + d(size, state)$
- t depends on the operation type
- m: mode switch between user mode and kernel mode
- Time for data transfer depends on data locality



System Model: Time costs in the I/O path

- $p(size)$: copy data; user space and kernel space (page cache):
  - p_{R-L1}: Register to L1
  - p_{L1-L1}: L1 to L1
  - p_{L2-L2}: L2 to L2 ...
  - p_{m-m}: memory to memory
  - $p_{numa-numa}$: memory one CPU to another CPU
- $c(size)$: copy data between page cache and device cache
  - c_r: copy data between register and device
  - c_{L1}: copy data between L1 and device ...
  - c_m: copy data between memory and dev using DMA
  - c_{numa}: copy data between another CPU's memory and dev
- $d(size, state)$: time for device io $= d_{seq}(size) + d_{prep}(size)$
  - $d_{seq}(size)$: sequential I/O
  - $d_{prep}(size)$: preparation time, seek time, flush erasure block
- $n(size)$: network transfer data between client and server
  - Also based on memory locality in respect to the I/O port