Analyzing and Assessing I/O Performance

Julian M. Kunkel

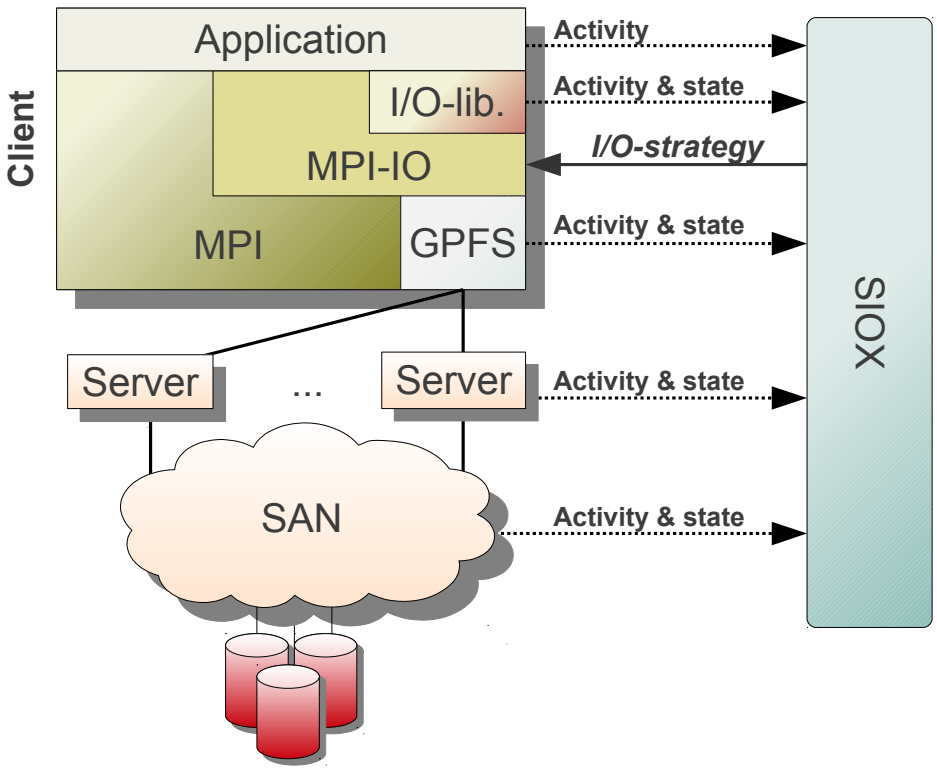
kunkel@dkrz.de

German Climate Computing Center (DKRZ)

30-07-2015

Outline

SIOX Goals



SIOX will

- collect and analyse
  - activity patterns and
  - performance metrics

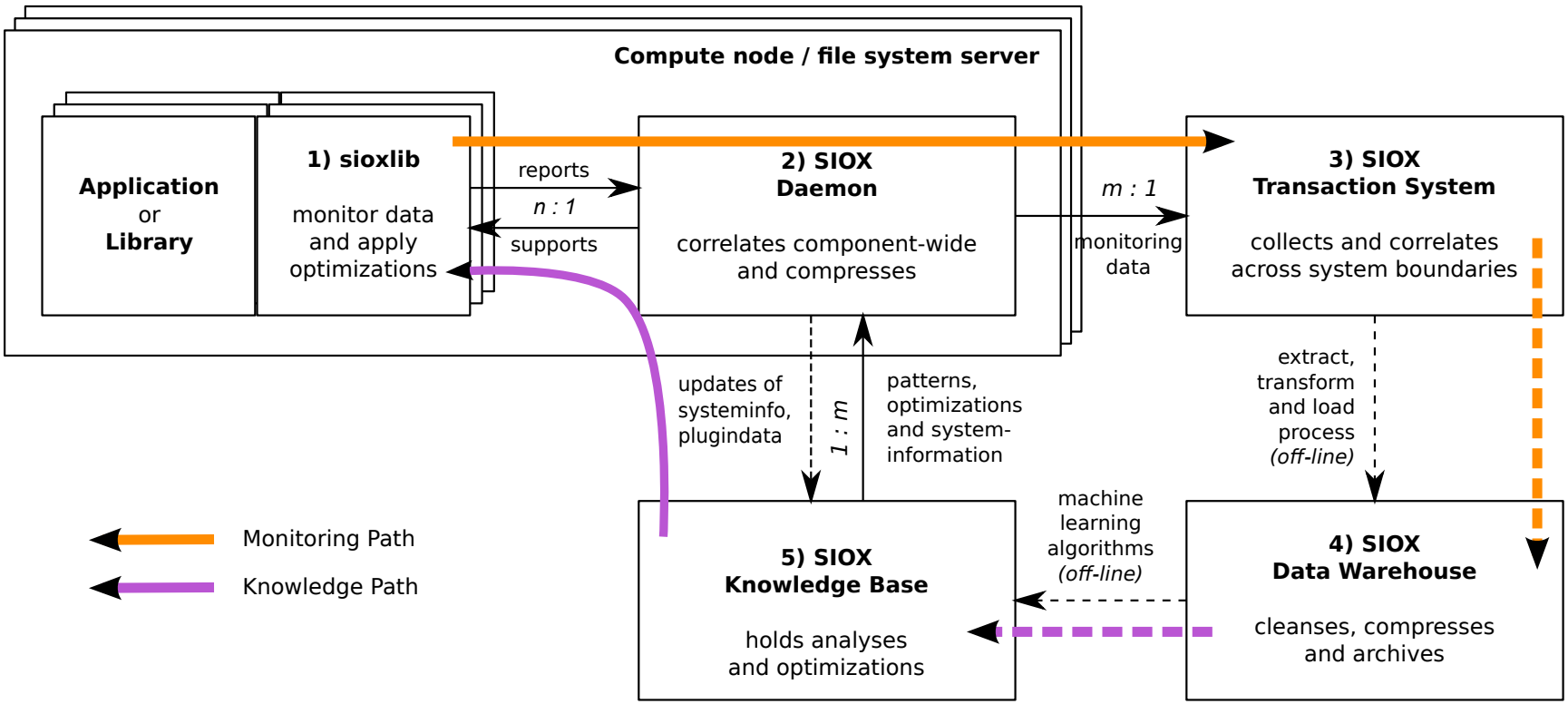
in order to

- assess system performance
- locate and diagnose problem
- learn & apply optimizations
- intelligently steer monitoring

Modularity of SIOX

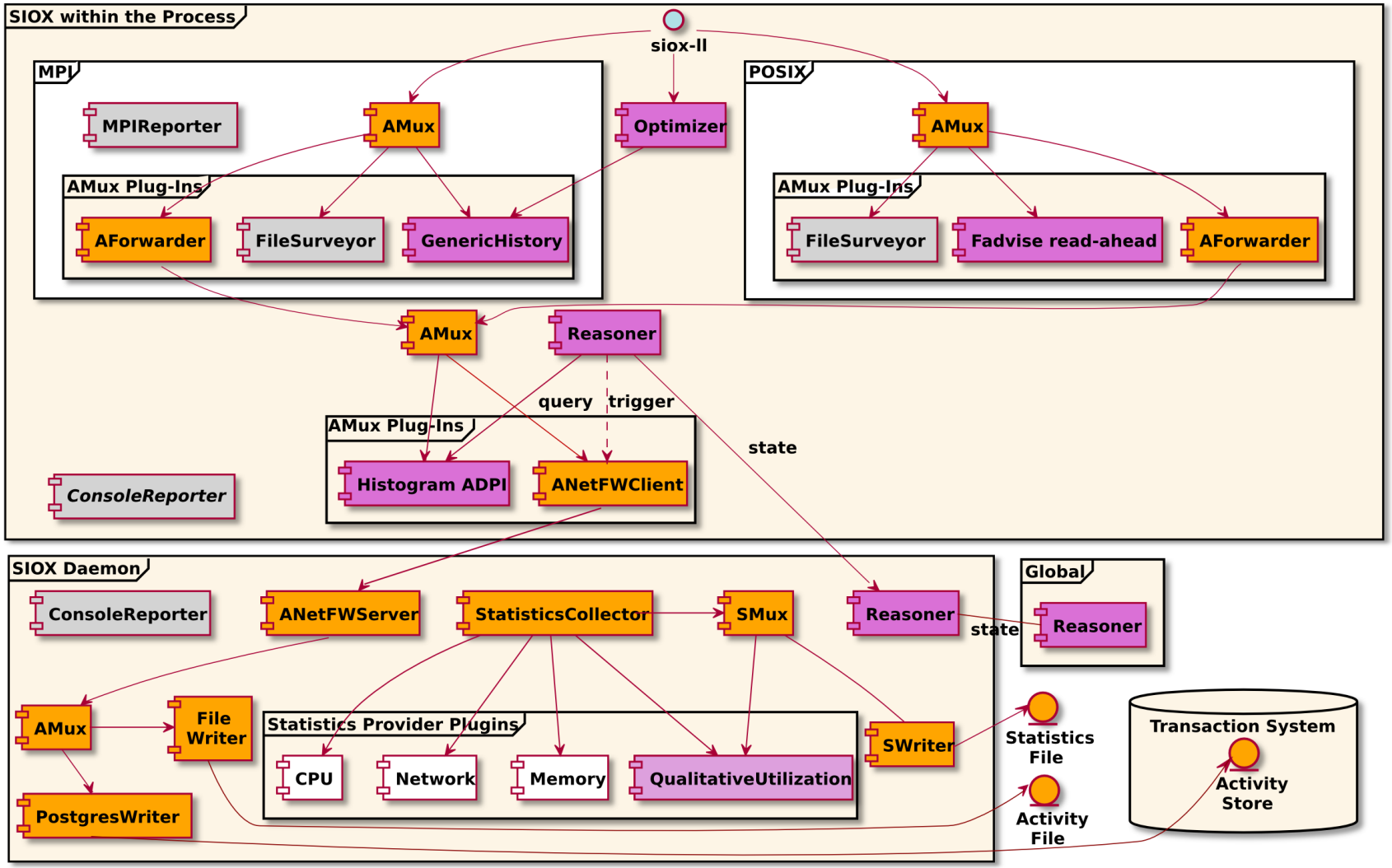
- The SIOX architecture is flexible and developed in C++ components
- License: LGPL, vendor friendly
- Upon start-up of (instrumented) applications, modules are loaded
- Configuration file defines modules and options
  - Choose advantageous plug-ins
  - Regulate overhead
- For debugging, **reports** are output at application termination
  - SIOX may gather statistics of (application) behavior / activity
  - Provide (internal) module statistics

Faces of SIOX: General System Architecture



- Data gathered is stored via the *monitoring path*
- Components receive the knowledge gleaned via the *knowledge path*

Module Interactions of an Example Configuration



A few facts about the prototype

- Monitoring
  - Application (activity) behavior
  - Ontology and system information
  - Data can be stored in files or Postgres database
  - Trace reader
- Daemon
  - Applications forward activities to the daemon
  - Node statistics are captured
  - Energy consumption (RAPL) can be captured
- Activity plug-ins
  - *GenericHistory* plug-in tracks performance, proposes MPI hints
  - Fadvise (ReadAhead) injector
  - *FileSurveyor* prototype – Darshan-like
- Reasoner component (with simple decision engine)
  - Intelligent monitoring: trigger monitoring on abnormal behavior
- Reporting of statistics on console or file (independent and MPI-aware)

Virtual Laboratory for I/O Investigation

Virtual Lab: Conduct what if analysis

- Design new optimizations
- Apply optimization to application w/o changing them
- Compute best-cases and estimate if changes pay off

Methodology

- Extract application I/O captured in traces

1. Allow manipulation of operations and replay them in a tool
2. Allow on-line manipulation

So far: Flexible Event Imitation Engine for Parallel Workloads (feign)

- Helper functions: to pre-create environment, to analyze, ...
- A handful of mutators to alter behavior
- Adaption of SIOX is ongoing to allow on-line experimentation

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?

Paper: *Predicting Performance of Non-contiguous I/O with Machine Learning. Kunkel, Julian; Zimmer, Michaela; Betke, Eugen. 2015, Lecture Notes in Computer Science*

Measured Data

- Captured on DKRZ porting system for Mistral
  - Evaluate if machine learning could be useful for our next system
- What Lustre and data sieving settings are useful defaults?
- Vary lustre stripe settings
  - 128 KiB or 2 MiB
  - 1 stripe or 2 stripes
- Vary data sieving
  - Off or 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

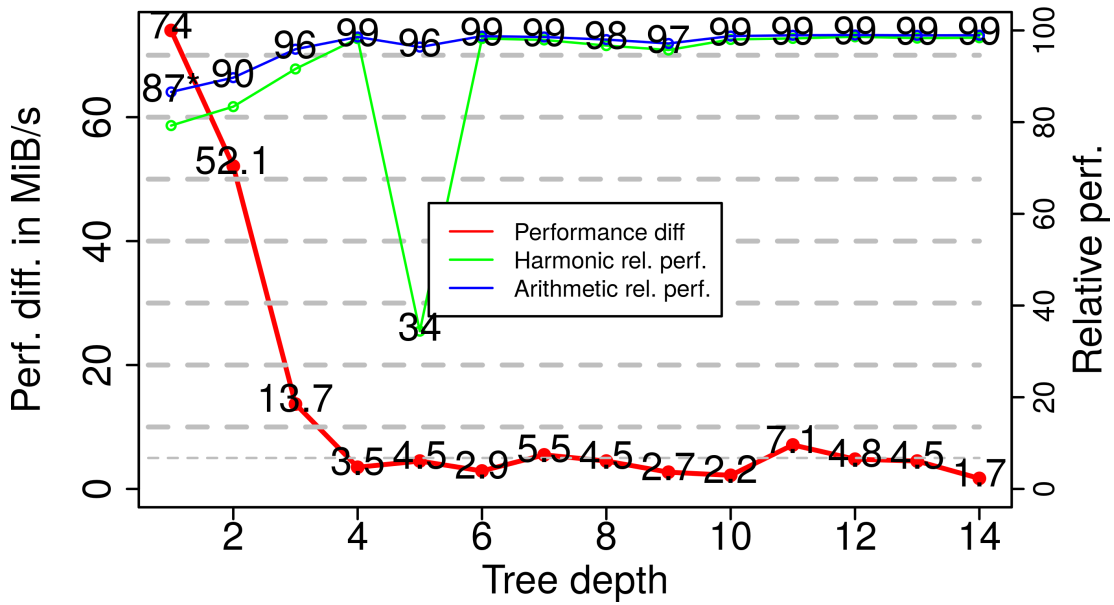
- All choices achieve 50-70% arith. mean perf.
- Picking the best default default choice: 2 servers, 128 KiB
  - 70% arithmetic mean performance
  - 16% harmonic mean performance

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

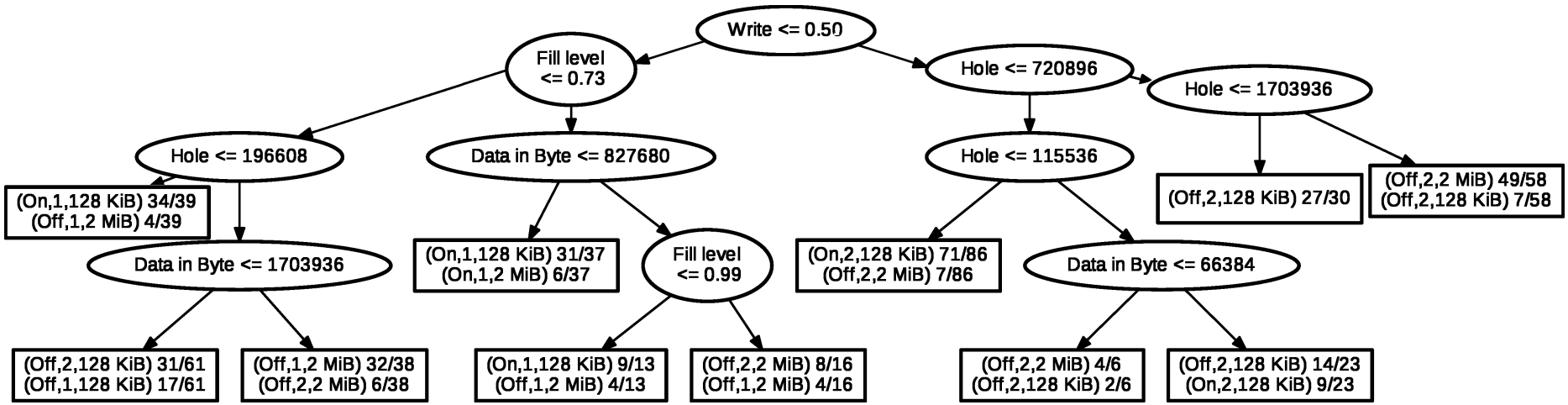


Perf. difference between learned and best choices, by maximum tree depth, for DKRZ's porting system

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

Goal: Assessing I/O performance

- Upon process termination reporter shall output assessment
- Qualitative overview based on system model

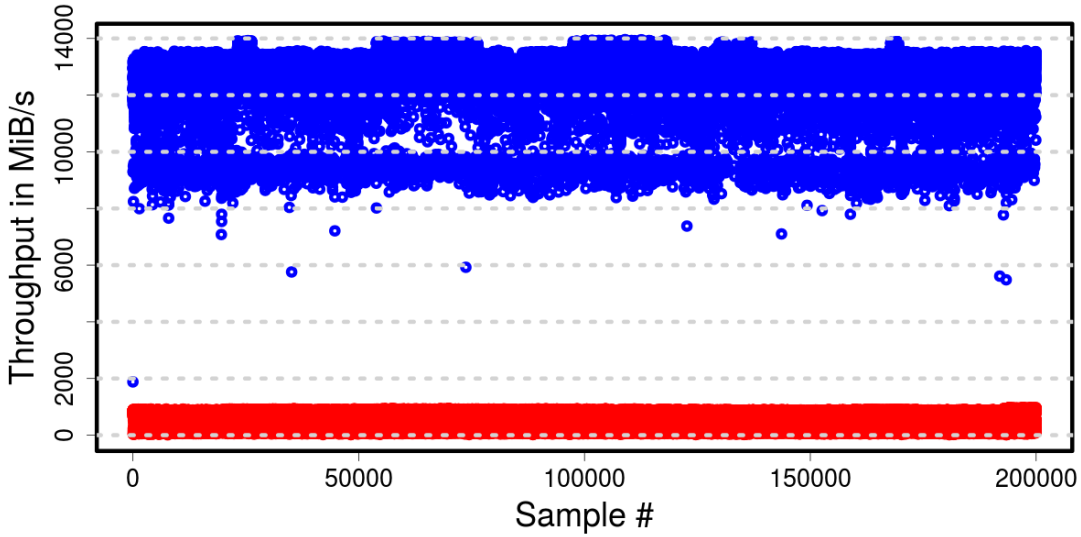
Example: Reporter Output

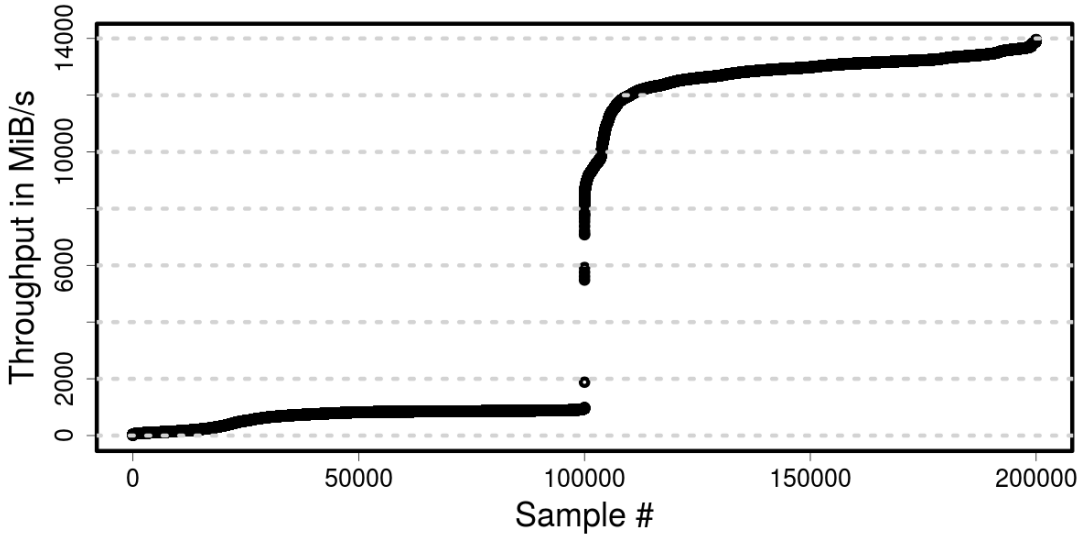
```
Your Read I/O consisted of:
200 calls/100 MiB
10 calls/10 MiB were cached in the system's page cache
10 calls/20 MiB were cached on the server's cache
100 calls/40 MiB were dominated by average disk seek time
...
5 calls/100 KiB were unexpected slow (3.5s time loss)

I/O time: 20.1s
Optimal I/O time: 5s
```

Difficulties in localizing causes in measurements

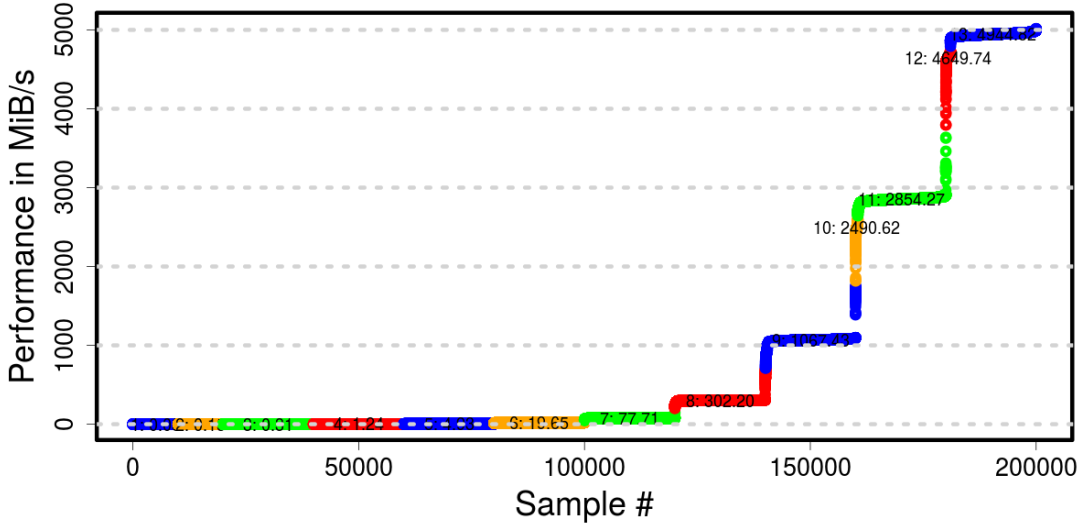
dd if=/dev/zero of=testfile bs=1024k count=100000

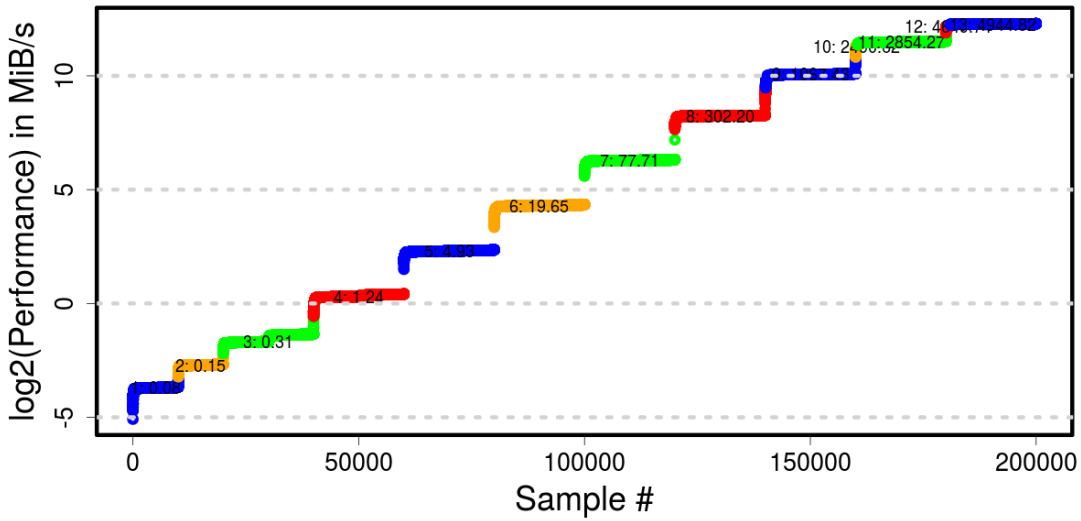




Difficulties (2)

Varying access granularity leads to a different picture



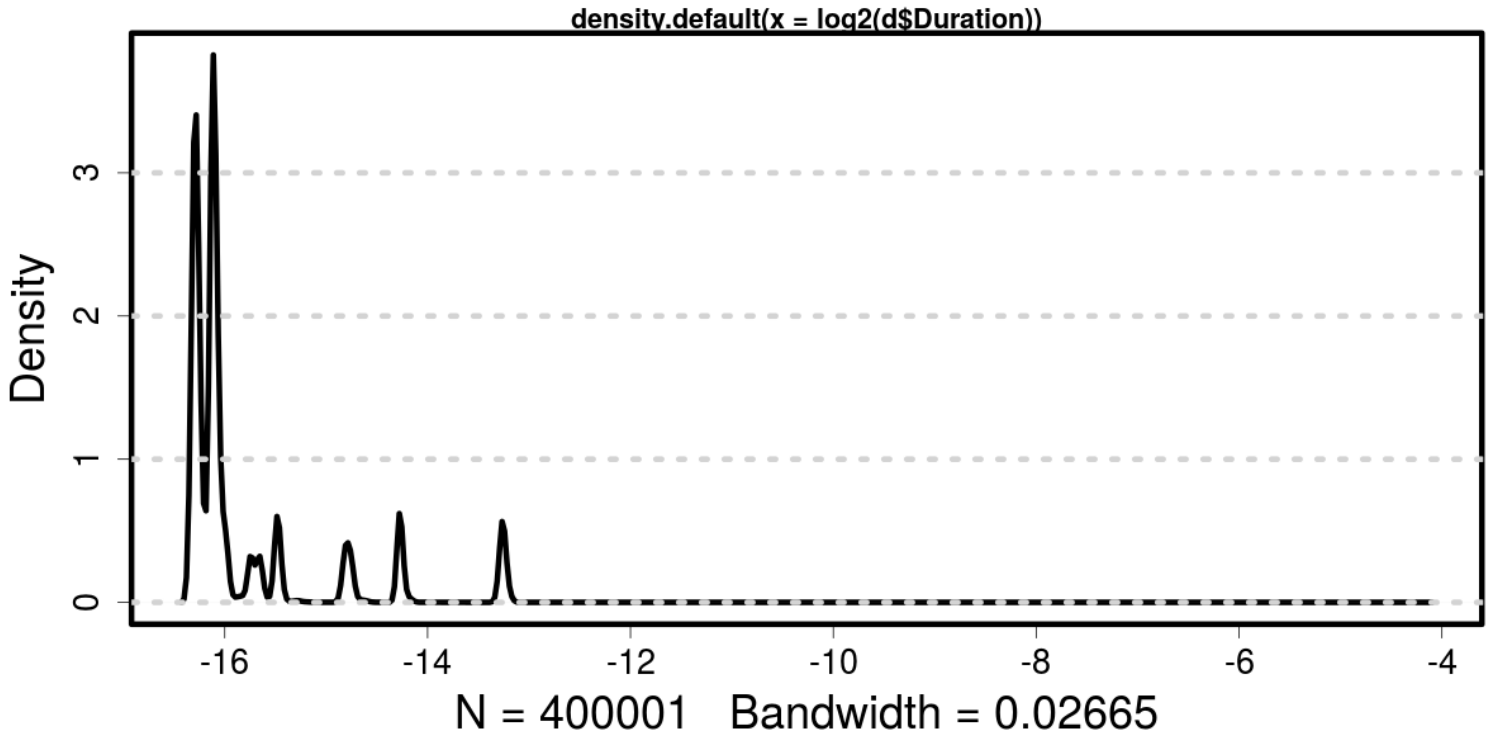


Identification of relevant issues

Apply density estimation of observed performance distribution

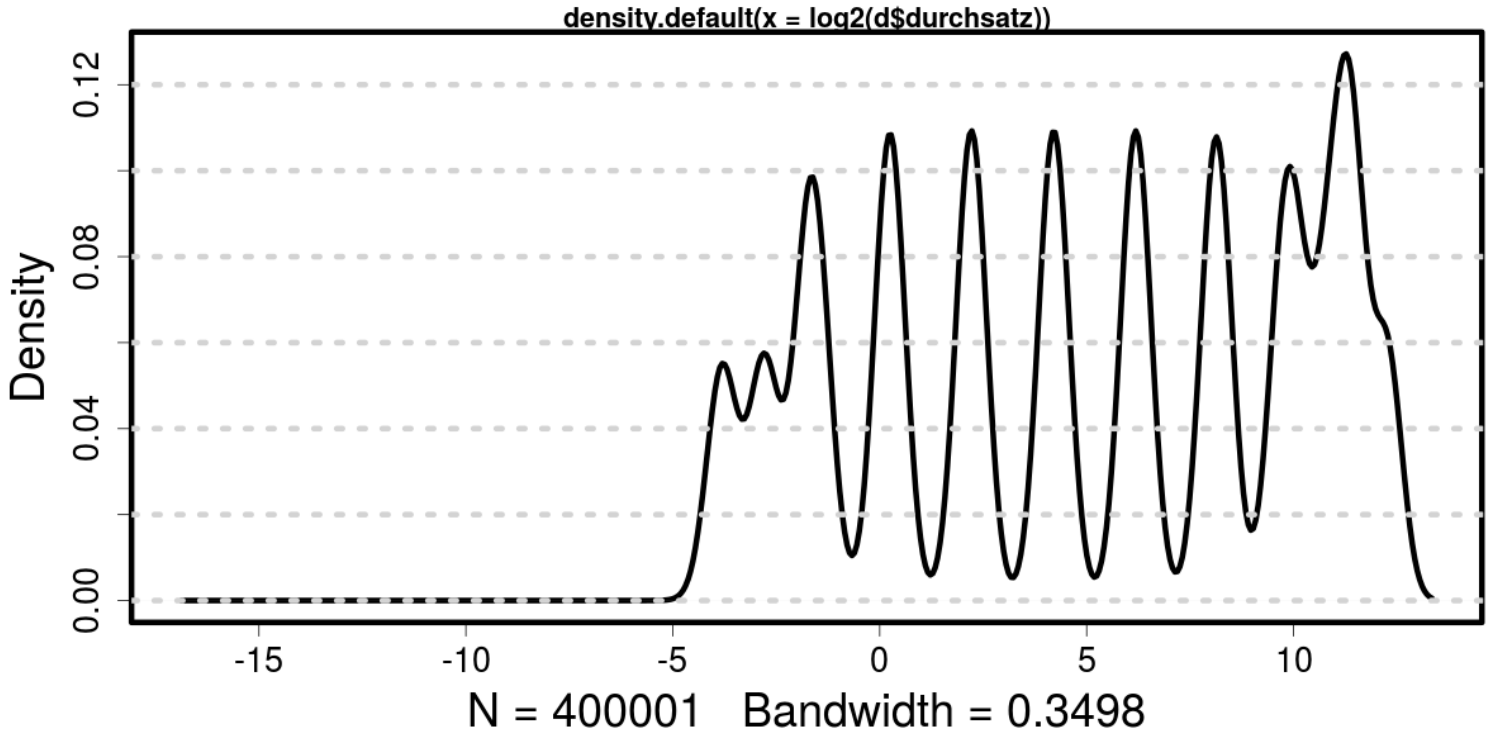
- Reveals performance-relevant behavior
- Technique was used to created classes(colors) on graphs before

Density based on throughput



Density graphs on throughput

Normalize density graph – based on throughput



How to identify causes?

Problem

The storage abstraction layer hides causes

Available information sources to identify causes

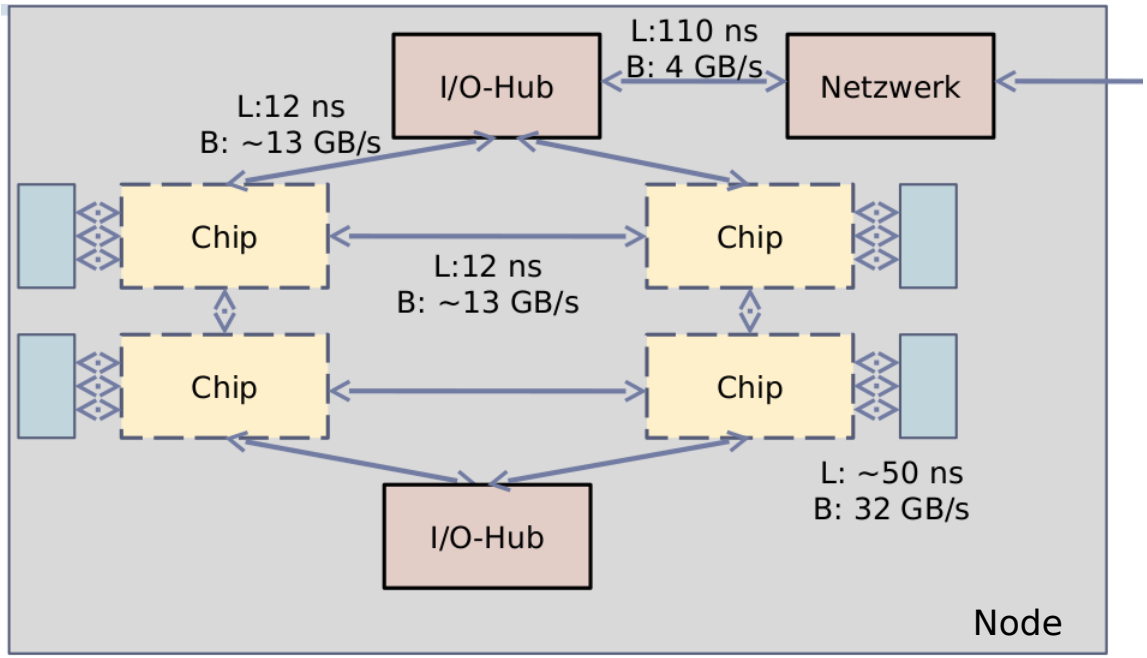
- /proc/self/io (# IOOps and bytes, ũncached/cached)
- Systemtap to capture kernel events/calls e.g. BlockIO
- ?

Approach

- Model performance and cause based on expert knowledge
- Recursively extract relevant “issues” from density graphs

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

From Observation to Likely cause

- We observe $\hat{t}$ for size, where is data localized and what happened?
- Apply a family of linear models predicting time; $lm(size) = c + x(size)$
  - Assume time correlates to operation size
  - Each model represents conditions 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 partial conditions
  - i.e. some time caused by C and some by C'

Example models

- $t(size) = m$: Data is discarded on the client or not overwritten
- $t(size) = m + p(size)$: Data is completely cached on the client ...

Training the Model

- Try to create well known conditions
- Built the model bottom up
  - Account for optional and alternative conditions
- Vary access granularity

Isolated analysis is non-trivial

- Writes may result in write-back
- Sometimes even re-read of data may be uncached
- Many different locations of data in the CPU caches
- Background daemons cause unexpected slow down

Identifying causes

- Semi-automatic identification of performance-critical issues
- Determine residual ($\delta_t = \hat{t} - t$) and analyze results

Evaluation

So far a simplified version of the model has been tested.

- Calibration on one memory benchmark with varying size
- One buffer pre-allocated, overwrite of file (lseek(0))

$\Rightarrow$ Estimates in CPU cached I/O

- Based on remaining time classify into:
  - No I/O: “discard”, e.g. /dev/null
  - Cached in CPU
  - Other levels are manually coded based on remaining time
    - Cached memory: Faster than 0.2 ms
    - Fast I/O: Faster than 0.4 ms
    - Normal I/O: Faster than 7 ms (avg. latency)
    - Slow I/O: Faster than 25 ms (slow disk seek and/or contention)
    - Else: Unexpected slow
  - Correct linear models will be created in the future

Evaluation

Benchmarks on Pitbull – Mistral Porting system

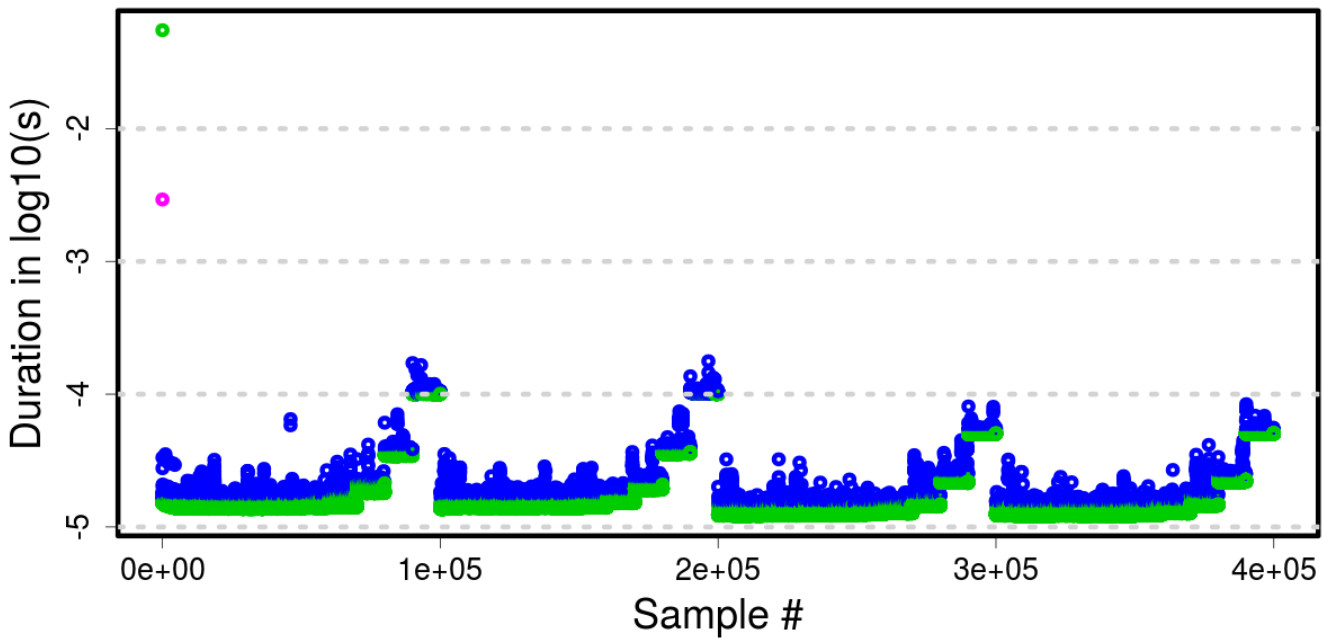
- I/O path and mode switch (@ 2.5 GHz)
  - read(), write() with 0 payload
  - on /dev/shm, write: 0.207255 micro, read: 0.175997 micro
  - on lustre: t.modeSwitch = 1.005506 micro

  $\Rightarrow$ Instructions consumed
    - on tmpfs 440
    - on lustre 2513
- Cached I/O with data fitting in CPU cache
  - Access size: 1, 4, 16, 64, 256, 1024, 4096, ..., 10 MiB
  - and size + 1 byte
  - Use a linear model, e.g. linear regression
  - read latency: 11.4 micro (28k cycles)
  - write latency: 12.9 micro (32k cycles)
  - read s per byte: 0.14 nano (2.8 bytes/cycle)
  - write s per byte: 0.33 nano (1.2 bytes/cycle)

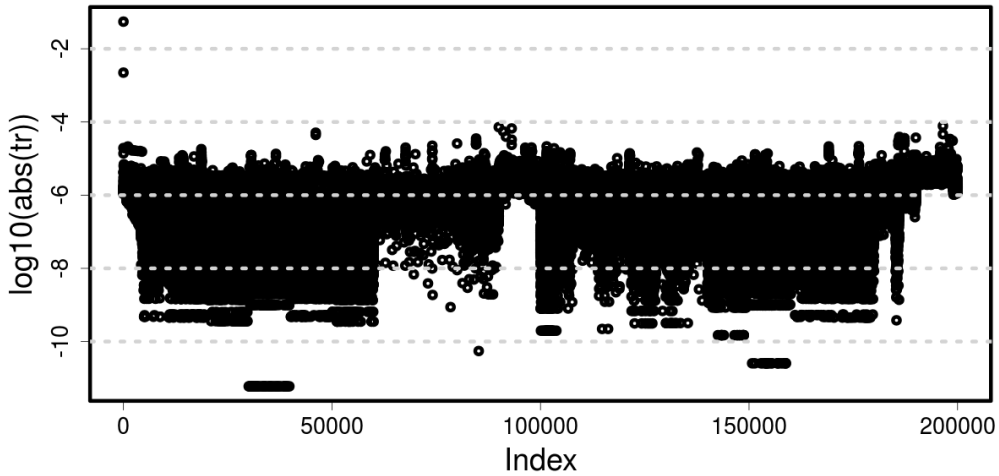
Applying the model to its training data

Classified samples

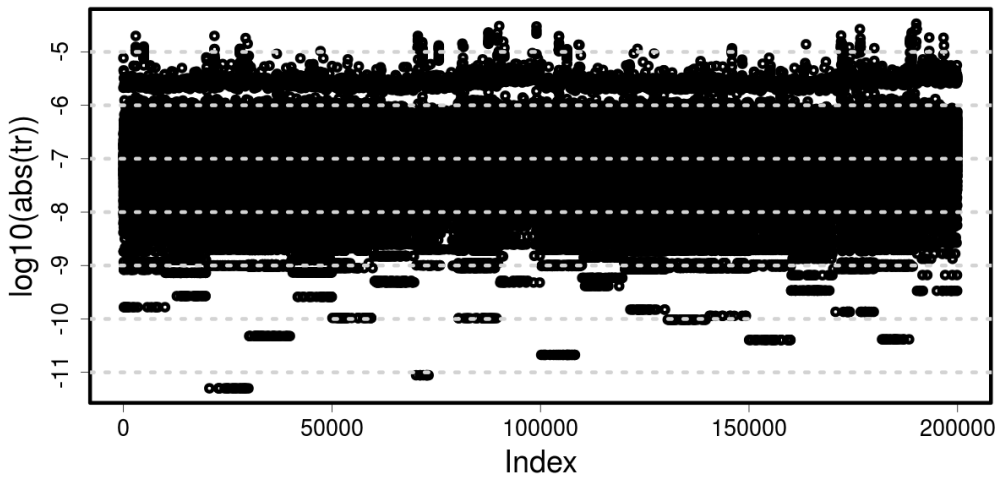
- cached CPU :379472
- cached memory : 20527 (should be cached CPU)...
- fast I/O : 1
- unexpected slow: 1
- others: 0



Subtracting prediction: Remaining time



For write

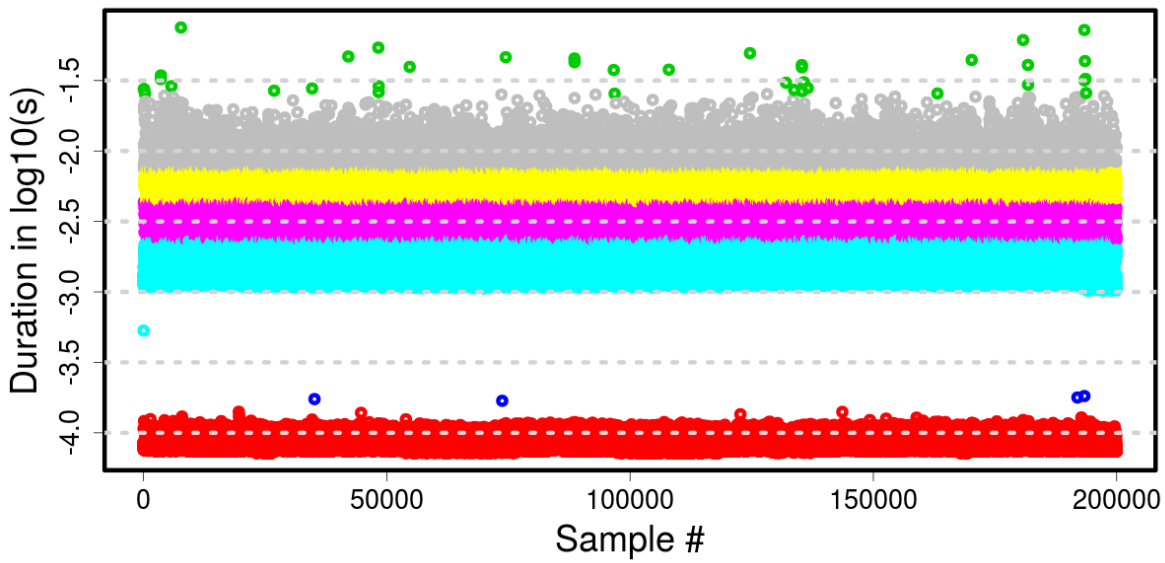


For read

Running DD

dd if=/dev/zero of=testfile bs=1024k count=100000

- discarded :99995
- cached storage :77994
- slow I/O : 7723
- normal I/O : 7665
- fast I/O : 6581
- unexpected slow: 38



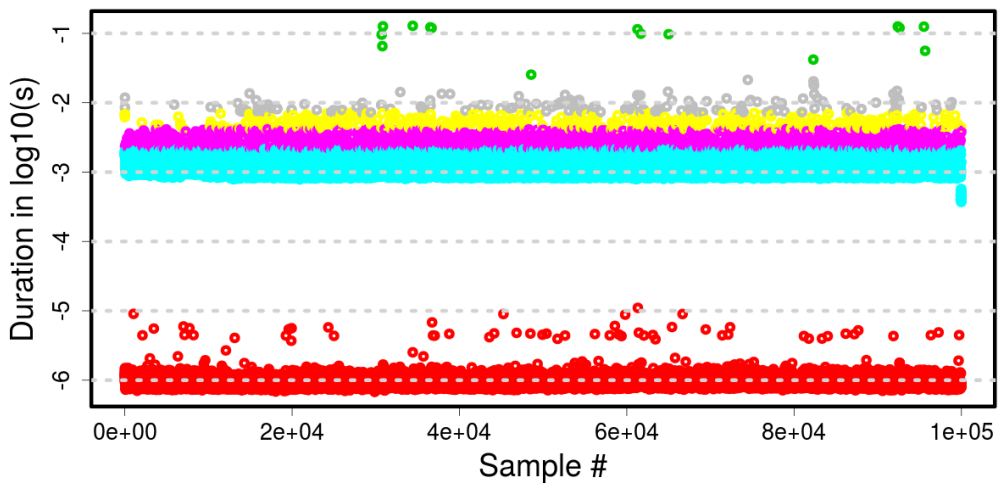
Running DD for reads

Cached read: dd of=/dev/null if=testfile bs=1024k count=50000

- discarded :50000
- cached memory :49999
- cached storage: 1

Uncached read: dd of=/dev/null if=testfile bs=1024k count=50000

- discarded :50000
- cached storage :47990
- fast I/O : 1412
- normal I/O : 432
- slow I/O : 151
- unexpected slow: 15



Summary

- SIOX serves as research vehicle
  - Supports analysis and evaluation with a plugin architecture
- Virtual laboratory assists in evaluating benefit of I/O methods
  - Ongoing work: integration into SIOX for online transmutation
- Best-practises can be extracted with machine learning
- Ongoing work: performance models identify reasons
  - Already the simple models assist in performance debugging
  - A comparison of simple models vs. advanced models will be done