Using a Relative Index of Performance (RIP) and Random Forest to determine optimum configuration settings

Diane Feddema
Red Hat Inc

Special thank you to Andrew Theurer, Red Hat Inc for providing the data used in this presentation.
Overview – Why do we need this?

• Determine best set of default configuration settings that benefit the most use cases

OR

• Choose best configuration settings for a particular application.
Talk Overview

1. Background
   a) What questions are we trying to answer?
   b) How are we collecting the performance data?

2. Case study using QEMU IO performance Dataset

3. Using Random Forest Algorithm to find important features
   a) What features are most important for performance?
   b) What is random Forest and why use it
   c) Example application

4. Using Relative Index of Performance (RIP) to measure performance between experimental set-ups
   a) What makes a good performance metric?
   b) What is RIP and why use it?
   c) Example application

5. Some problems encountered

6. Conclusions
Background (how to collect the data)

• Performance data is often collected to answer and single question and is usually not reused.
• We need better tools to collect and organize relevant information so that we can draw additional conclusions from the data
• A solution to this problem is being developed by open source project pbench, written by Andrew Theurerer. Peter Portante is creating a common name space for this data and archiving it in elastic search. Benchmark data collected in pbench is used in this talk.
Comparing threads vs. native async IO

• Hypervisor Qemu offers two different methods to submit IO requests. The default method IO=threads uses a thread pool to submit these requests synchronously, while the alternate method IO=native uses a single thread to submit these request asynchronously
• 26 disk configurations were tested
• For each configuration “threads” and “native async IO” were run (52 directories of output data -- 78 TB in total)
• Each run consists of 35 io tests (FIO benchmark) seq read/write, random read/write, read write mix multiple request sizes 4, 16, 64, 256K
Background (what question to ask?)

• The question we are addressing is “Disk I/O: which performs better Native or threads”

• Run an experiment with Random Forest to identify the important features (configuration settings) affecting performance

• Use the Relative Index of Performance (RIP) to compare “Threads” vs “Native async IO” performance across different configuration settings.
Random Forest: A Machine learning algorithm

Use a set of input variables to predict an outcome without understanding the exact relationship between the inputs and outcomes.

Breiman 2001 “Statistical Modeling: The Two Cultures”
Random Forest Algorithm

- Ensemble Classifier using many decision trees
- Developed in the mid-1980s
- Popularized by Leo Breiman, UC Berkeley
- Forests of decision trees “vote” on the best answer (final classification)
- Example applications include:
  - Automatic Image Analysis
  - Land cover classification
  - Automatic diagnosis in medical applications
  - The source of delay in criminal trials in state court systems
Using Random Forest to Find Important Variables (Config Settings)

- Random Forests can be used for
  - regression
  - classification
  - rank the importance of variables in a regression or classification problem.
- Nodes near the top of the tree contain the most important variables.
- Each variable gets an importance score. Variables are used to split the data at each node.
- Sub-samples of training data and their classification outcomes (dependent variable) are used to train the system
- Splits that maximize information gain are chosen
How Random Forest Works

• Each decision tree is grown using a **random** bootstrap sample of the training data (each tree gets diff data)

• A number m is specified which tells RF how many features to select at **random** (without replacement) out of p possible variables to use when splitting a node into sub-nodes.

• Each decision tree “votes” on the final classification of input feature vectors.

• Final classification is given by majority voting.
How Random Forest Works

From Criminisi et al (2010)
“Voting” on final classification

From Criminisi et al (2010)
System Configuration settings considered in the QEMU IO Dataset

Single vs multiple (16) VMs running

Storage types:
- SSD-fs
- New_SSD-fs
- SSD-img
- SSD-qcow2
- HDD

QEMU io request method type:
- Threads
- Native async io

Profile Type:
- Preallocate
- Sync
- Libaio

IO Test:
- Read
- Write
- Randread
- Randwrite
- Randrw

OS/Patches:
- Base-rpm-w/o-perf
- Stefan-w/o-perf
- Stefan-rhel72-w/o-perf
- Stefan-rhel72-perf
- perf-fix-w-aio-patch
- Patch
- fullrun
- Fullrun-iodepth

FS types:
- xfs
- ext4
- lvm
Binary Classification from features in QEMU IO dataset

<table>
<thead>
<tr>
<th>Test</th>
<th>Num vms</th>
<th>Thrd or Nat</th>
<th>SSD or HDD</th>
<th>Fstype</th>
<th>Profile</th>
<th>OS/ Patches</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>d1</td>
<td>e1</td>
<td>f1</td>
<td>g1</td>
<td>Slow</td>
</tr>
<tr>
<td>a2</td>
<td>b2</td>
<td>c2</td>
<td>d2</td>
<td>e2</td>
<td>f2</td>
<td>g2</td>
<td>Fast</td>
</tr>
<tr>
<td>a3</td>
<td>b3</td>
<td>c3</td>
<td>d3</td>
<td>e3</td>
<td>f3</td>
<td>g3</td>
<td>Slow</td>
</tr>
<tr>
<td>a4</td>
<td>b4</td>
<td>c4</td>
<td>d4</td>
<td>e4</td>
<td>f4</td>
<td>g4</td>
<td>Slow</td>
</tr>
<tr>
<td>a5</td>
<td>b5</td>
<td>c5</td>
<td>d5</td>
<td>e5</td>
<td>f5</td>
<td>g5</td>
<td>Fast</td>
</tr>
<tr>
<td>....</td>
<td>....</td>
<td>....</td>
<td>....</td>
<td>....</td>
<td>....</td>
<td>....</td>
<td>....</td>
</tr>
</tbody>
</table>
Constructing a tree: using sub-samples

Are you running single VM?

YES

No

Is your profile libaio?

YES

No

Are you Threads?

YES

No

Are you xfs?

YES

No

Are you Storage type SSD?

YES

No

Do you have Stefan patch?

YES

No

Are you xfs?

1

0

1

0

1

1

1

1

Outcome FAST or SLOW (Training data)
Creating a Forest: repeat the tree process

Is your FS type ext4?
- YES
- NO

Are you HHD?
- YES
- NO

Are you Native?
- YES
- NO

Are you running single VM?
- YES
- NO

Are you SSD?
- YES
- NO

Outcome FAST or SLOW (Training data)

A Forest!

2 Trees!
RF applied to Virtual Disk Data configuration settings

- Example CSV file with Virtual Disk Data

```
iteration,rw_kb_sec,num_vms,t_or_n,ssd_or_hd,fsystype,ospatches,throughput
1-read-4KiB,423976.94,fio_multi-vm-cache,500,1000,ext4,fullrun-node-p10-run-1,<= 1067024
2-read-16KiB,519849.68,fio_multi-vm-cache,500,1000,ext4,fullrun-node-p10-run-1,<= 1067024
3-read-64KiB,518008.48,fio_multi-vm-cache,500,1000,ext4,fullrun-node-p10-run-1,<= 1067024
4-read-256KiB,556735.88,fio_multi-vm-cache,500,1000,ext4,fullrun-node-p10-run-1,<= 1067024
5-read-1024KiB,547706.63,fio_multi-vm-cache,500,1000,ext4,fullrun-node-p10-run-1,<= 1067024
6-read-4096KiB,579141.7,fio_multi-vm-cache,500,1000,ext4,fullrun-node-p10-run-1,<= 1067024
7-read-16384KiB,1268846.57,fio_multi-vm-cache,500,1000,ext4,fullrun-node-p10-run-1,> 1067024
8-write-4KiB,276940.59,fio_multi-vm-cache,500,1000,ext4,fullrun-node-p10-run-1,<= 1067024
9-write-16KiB,280747.03,fio_multi-vm-cache,500,1000,ext4,fullrun-node-p10-run-1,<= 1067024
10-write-64KiB,278721.15,fio_multi-vm-cache,500,1000,ext4,fullrun-node-p10-run-1,<= 1067024
11-write-256KiB,279752.15,fio_multi-vm-cache,500,1000,ext4,fullrun-node-p10-run-1,<= 1067024
12-write-1024KiB,282276.64,fio_multi-vm-cache,500,1000,ext4,fullrun-node-p10-run-1,<= 1067024
```
Preprocessing the data: One-Hot encode categorical data in Pyspark

categoricalColumns=["iteration","num_vms","t_or_n","ssd_or_hd","fsystype","ospatches"]
stages = []
for categoricalCol in categoricalColumns:
    stringIndexer = StringIndexer(inputCol=categoricalCol, outputCol=categoricalCol+"Index")
# OneHotEncoder converts categorical varaibles into numeric varaibles
encoder = OneHotEncoder(inputCol=categoricalCol+"Index", outputCol=categoricalCol+"classVec")
    stages += [stringIndexer, encoder]
e.g.

FS types:
• xfs
• ext4
• lvm

xfs ➔ 1 0 0  
Ext4 ➔ 0 1 0  
Lvm ➔ 0 0 1
val df = sqlContext.read.load("/home/dfeddem/AllCategoricalIncludeProfiletypeexcludethuputfromfeatvect.parquet")

val rf = new RandomForestClassifier().setLabelCol("label").setFeaturesCol("features").setNumTrees(20)

val Array(trainingData, testData) = df.randomSplit(Array(0.7, 0.3))
val rfModel = rf.fit(trainingData)
val predictions = rfModel.transform(testData)
Feature importance by index

- println(rfModel.featureImportances)

(115, [0,1,2,3,4,5,6,7,8,9,11,12,13,14,15,16,17,18,19,20,23,24,25,26,28,29,30,31,32,34,38,43,50,52,55, 56,58,61,62,63,66,69,70,71,72,75,78,81,83,84,85,86,87,88,89,90,91,92,93,94,95,96,98,99,100,102, 103,104,105,106,107,108,109,110,111,113,114],
[0.0012838188848223575,0.005210803734568604,0.0010423898130666538,0.007294162268360384,0.003597311304473944,0.0017692880960633712,0.0030163378230193072,0.0026138259576503307,0.010358004163238775,0.00248492885411117917,0.022037216541414256,5.09422444951224E-5,0.0016718915655253366,0.002338759791261354,3.477889950131344E-5,0.010484829870617041,0.03765392486968075,0.005581929997827416,1.0452988130154432E-4,0.030711391193927228,0.0012672374202991978,0.002244968038570229,0.019781534841138344,3.3979490071466245E-5,3.184993222495474E-4,0.011440932397213738,0.00492847964454524,0.010294296447750498,2.584966955489007E-4,0.006655726392365051,0.002295918571258585,6.150156652772081E-4,8.272133022656789E-4,0.0024058441589414246,6.479629330599128E-4,0.01426606379587654,4.516105291023641E-4,0.0024030294958581264,0.002847078581148401,0.002876821413616359,0.013174878315431718,0.001175917225351889,9.823136120354359E-6,1.6924306867187937E-4,6.572382208864174E-6,1.2393617584132184E-4,8.998853655472037E-5,1.8574657391220994E-5,2.9909471756857938E-5,0.002948569577859674,0.025821678504687334,0.0024015550997397534,0.0024182323774033206,0.0700813124109098,0.010642396928730584,0.1054382062551498,0.00479431386439967,0.1455098698168787,0.01836376454142565,0.034805448548903314,0.00598137322390217,0.0067710133135019555,0.03781499374732001,0.022800769795242645,0.002503769778959118,0.04612436501196291,0.013389280716803753,0.0037075732661612007,0.05409288637586559,0.04187035404000151,0.01005129434435067,0.007384851815964782,0.007498443039453527,0.0546351462077412,0.030310721149049546,0.030006534574314497,0.02932143740813991]}
Random Forest Results: 10 tree vs 20 tree
Moving from Random Forest to identify important features to using RIP to measure the relative change in performance.
Relative Index of Performance (RIP)

Assume:

- **Initial** = performance value for a metric (e.g. Throughput or Duration) with initial set up
- **Experiment** = performance value for a metric (e.g. Throughput or Duration) with experimental set up

Separate cases for Throughput and Duration variables

For Throughput variables

- (higher throughput = better)

  if (Experiment < Initial) then
    RIP = Experiment / Initial – 1.0
  else
    RIP = 1.0 – Initial / Experiment
  endif

For Duration Variables

- (shorter time = better)

  if (Experiment < Initial) then
    RIP = 1.0 – Experiment / Initial
  else
    RIP = Initial / Experiment – 1.0
  endif

* For this project the throughput version is used
Advantages of the RIP index

- Doubling or halving gives same numeric value
- Relative performance difference is measured equally in either direction
- Differences between performance numbers are additive and can be averaged
- Relative performance differences are the same across metrics with different units
- Index differences for variables with different units can be averaged

e.g.
Comparing RIP to ratio and percent change values

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Initial</th>
<th>RIP</th>
<th>Ratio</th>
<th>Δ %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>1.00</td>
<td>1000.00</td>
<td>99900</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0.90</td>
<td>10.00</td>
<td>900</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0.80</td>
<td>5.00</td>
<td>400</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.50</td>
<td>2.00</td>
<td>100</td>
</tr>
<tr>
<td>1.5</td>
<td>1</td>
<td>0.33</td>
<td>1.50</td>
<td>50</td>
</tr>
<tr>
<td>1.1</td>
<td>1</td>
<td>0.09</td>
<td>1.10</td>
<td>10</td>
</tr>
<tr>
<td>1.0</td>
<td>1</td>
<td>0.00</td>
<td>1.00</td>
<td>0</td>
</tr>
<tr>
<td>0.9</td>
<td>1</td>
<td>-0.10</td>
<td>0.90</td>
<td>-10</td>
</tr>
<tr>
<td>0.66</td>
<td>1</td>
<td>-0.34</td>
<td>0.66</td>
<td>-34</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>-0.50</td>
<td>0.50</td>
<td>-50</td>
</tr>
<tr>
<td>0.2</td>
<td>1</td>
<td>-0.80</td>
<td>0.20</td>
<td>-80</td>
</tr>
<tr>
<td>0.1</td>
<td>1</td>
<td>-0.90</td>
<td>0.10</td>
<td>-90</td>
</tr>
<tr>
<td>0.0</td>
<td>1</td>
<td>-1.00</td>
<td>0.00</td>
<td>-100</td>
</tr>
</tbody>
</table>
Comparing RIP to ratio and percent change values

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Initial</th>
<th>RIP</th>
<th>Ratio</th>
<th>Δ %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.00</td>
<td>1.00</td>
<td>0</td>
</tr>
</tbody>
</table>
Comparing RIP to ratio and percent change values

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Initial</th>
<th>RIP</th>
<th>Ratio</th>
<th>Δ %</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0.50</td>
<td>2.00</td>
<td>100</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>-0.50</td>
<td>0.50</td>
<td>-50</td>
</tr>
</tbody>
</table>
Comparing RIP to ratio and percent change values

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Initial</th>
<th>RIP</th>
<th>Ratio</th>
<th>Δ %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>1.00</td>
<td>1000.00</td>
<td>99900</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0.90</td>
<td>10.00</td>
<td>900</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0.80</td>
<td>5.00</td>
<td>400</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.50</td>
<td>2.00</td>
<td>100</td>
</tr>
<tr>
<td>1.5</td>
<td>1</td>
<td>0.33</td>
<td>1.50</td>
<td>50</td>
</tr>
<tr>
<td>1.1</td>
<td>1</td>
<td>0.09</td>
<td>1.10</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.00</td>
<td>1.00</td>
<td>0</td>
</tr>
<tr>
<td>0.9</td>
<td>1</td>
<td>-0.10</td>
<td>0.90</td>
<td>-10</td>
</tr>
<tr>
<td>0.66</td>
<td>1</td>
<td>-0.34</td>
<td>0.66</td>
<td>-34</td>
</tr>
<tr>
<td>0.5</td>
<td>1</td>
<td>-0.50</td>
<td>0.50</td>
<td>-50</td>
</tr>
<tr>
<td>0.2</td>
<td>1</td>
<td>-0.80</td>
<td>0.20</td>
<td>-80</td>
</tr>
<tr>
<td>0.1</td>
<td>1</td>
<td>-0.90</td>
<td>0.10</td>
<td>-90</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-1.00</td>
<td>0.00</td>
<td>-100</td>
</tr>
</tbody>
</table>
Using RIP to evaluate the relative performance improvement of Threads v.s. Native async IO with different IO request sizes for all config settings.

Threads vs Native: Performance variability by IO request size averaged for all config settings

Positive values = Native has higher throughput than Threads.
Negative values = Threads has higher throughput than Native
Using RIP to evaluate Threads v.s. Native async IO with different IO request sizes and # VMs
Using RIP to evaluate Threads v.s. Native async IO with different IO request sizes and Storage devices

Change of Scale
RIP = 0.6 = 2.5 times increase

HHD
SSD
SSD-new
SSD-img
Using RIP to evaluate outliers and exceptional cases

0.8 = 5 fold improvement

Some of these (2) runs appear to be duplicates to other less improved simulations with no feature differences to explain the increased performance

Other related runs
Problems Encountered

• Random Forest Important Features are returned to the user as a index. Must reverse engineer to get feature names.

• We need to systematically collect more performance data with a uniform name space in order to fully implement automated performance tuning.

• The pbench data contained some reruns of test with highly variable results and yet both results were in the data.
Conclusions

• Random Forest is an useful method for identifying which configuration settings are good candidates for improving overall performance.

• RIP lends it self well to automation and for comparing and aggregating performance metrics with different units

• In combination these methods with a systematic data collection effort could provide significant insight into performance issues and identify potential solutions