THE UNIVERSITY OF CHICAGO

FREP: FLEET PERFORMANCE EVALUATION PIPELINE FOR IDENTIFYING SERVER DEGRADATION IN LARGE-SCALE DATACENTERS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER

DEPARTMENT OF COMPUTER SCIENCE

BY
RUIDAN LI

CHICAGO, ILLINOIS
2023

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I am deeply grateful to my advisor, Haryadi Gunawi, for his insightful guidance and support throughout this research journey. I would also like to express my appreciation to Jack Nugent, who has been assisting with this project from many perspectives. Lastly, I extend my sincere thanks to my friends and family for their unwavering support and encouragement.

ABSTRACT

Stable and predictable performance of fleet servers is critical for providing reliable services. To identify and remove underperforming servers, performance data must be recorded and then systematically analyzed at scale. In this thesis, we report on our methodology for collecting and analyzing performance data at COMPANYX *for the purposes of identifying and removing underperforming datacenter servers. We present (and will release), to the best of our knowledge, the largest dataset on CPU and memory benchmarks. We show how off-the-shelf statistical techniques must be customized for the nature of our dataset, resulting in a custom pipeline,* FREP*, that leverages a wide range of statistical techniques.* FREP *comprises a variety of modules that detect variances, divergences, and outliers, and has provided various types of important recommendations to* COMPANYX *operators and helped improve their understanding of their fleet behaviors.*

CHAPTER 1

INTRODUCTION

1.1 Introduction

In hyperscale server fleets, stable and predictable performance is critical for service reliability and efficient use of hardware. There are many reasons why a server's performance may become *worse* over time, including: a BIOS update may reduce performance; a new kernel version may not be optimized for the datacenter's specific characteristics; and a server's physical, circuit-level properties can gradually degrade. Failing to detect underperforming servers compromises service reliability and reduces efficiency at scale. Conversely, falsely identifying and removing high-performance servers from production results in poor fleet capacity management.

To identify and remove underperforming servers, performance data must be recorded and systematically analyzed *at scale*. Using an existing application as the basis for measuring performance has the appeal of most closely matching existing business requirements. However, using a more generic benchmark can better support changing requirements in the future. Either way, the overhead of performance testing must be minimized.

A given server's performance history provides a basis for determining whether it is a candidate for remediation—either through software or a hardware component swap. In particular, when a separate benchmark is used, assessing this performance is overhead (in comparison to production workloads), so it is unrealistic to expect a rich history of performance results for each server. As such, it is more likely that baselines consist of results from comparable servers. A fundamental challenge here is: given the significant software and hardware heterogeneity as the datacenter evolves, how can these equivalence classes be created/managed, and, within an equivalence class, what level of performance difference warrants removing a server from production?

In this paper, we present our methodology for collecting and analyzing performance, and how those analyses enable the identification and removal of underperforming datacenter servers. Our

dataset contains over 4 million benchmark results ("observations") on millions of servers ("assets") running production workloads at COMPANYX. For this pilot study, we use the publicly available CoreMark-PRO [3] and Stream [46] benchmarks for CPU and memory, respectively. Running these benchmarks is not trivial as it requires pausing and relocating all running services, displacing production workloads. For each asset, we also collect 20 features including asset-level (model ID, BIOS version, Linux kernel release, OS name), hardware-level (frequency scaling status, DIMM count, memory size), and other test and deployment features. To the best of our knowledge, this is the largest dataset of CPU and memory performance scores of real production servers.

To extract value from this dataset, we design and implement a system to identify anomalous machines in a fleet. A key design goal is minimizing the false positive rate. Prior to this system, identification of underperforming systems was limited to only the worst performers: send alerts if (a) the performance score is two standard deviations away from the mean or (b) 40% less than the overall peak performance. While this approach was effective, many potentially problematic servers were missed, and those identified required *ad hoc* human investigation to determine whether the server should be removed.

To develop our systematic approach, many important questions must be addressed. Beyond outliers at the asset level, are there observable patterns at the feature level, such as inter- and intra-model variation, performance drifts introduced by configuration changes, or firmware updates? Among all the features, can we automatically find the significant ones that provide more insightful analyses? What are the most suitable statistical techniques to achieve these goals? Can we take these techniques directly off-the-shelf or is customization needed to address the nature of our dataset?

To meet these requirements, in the last 1.5 years we have carefully crafted a fleet performance evaluation pipeline, FREP, that provides specific recommendations to COMPANYX fleet operators (the "domain experts") to inspect the fleet's performance behaviors. As described in Section 3.1, FREP begins the analysis with the "feature analysis/grouping" stage. We first rank **feature importance** with information gain and entropy approaches to separate determining, significant, and negligible features. From there, we identify `ModelID` as the most important feature and use it to perform a **top-level grouping** that categorizes the millions of assets into groups where the intra-group similarity among the assets can be analyzed. We then perform a finer-grained analysis via **subgrouping**, specifically breaking each top-level group (assets of the same model) into subgroups based on other significant and potentially high-impact features, such as `BIOS`, `ChipMV`, `FreqScale`, `UUT`, and `Kernel`.

With assets grouped in such a meaningful way, as described starting in Section 4.1, FREP then performs three types of analysis (and recommendations) to detect and address anomalous behaviors at the model, subgroup, and asset levels. **(a)** The **variance** module performs high-level anomalous detection at the *model level*. **(b)** The **divergence** module enables pairwise behavior comparison at the *subgroup level (2nd-level feature)* (*e.g.*, to analyze performance behaviors of different `Kernel` versions within the same model). **(c)** The **outlier** module detects unexpected behavior at the *asset level*. When the detection entity is a model group or a subgroup, the outlier detection could be inaccurate due to natural cross-asset variety. Therefore, this detection focuses on analysis and discovery at a machine level for better accuracy. This last module also enriches our current human-driven heuristics.

We have applied FREP to the dataset described in this paper and summarized a list of key observations illustrating the impact of FREP. We have also formed a list of anomalous server definitions that capture the individual characteristics more accurately as compared to currently used heuristics, which suffer from over-generalization and, as a result, are likely missing servers whose performance has been degraded.

In summary, we make the following contributions:

1. We present (and will release), to the best of our knowledge, the largest dataset on CPU and memory benchmarks, specifically CoreMark-PRO and Stream results.

2. We show how off-the-shelf statistical techniques must be customized for the nature of our dataset, resulting in a custom pipeline, FREP, that leverages a wide range of tools.

3. FREP has provided various types of important recommendations to COMPANYX operators (the domain experts) and helped improve their understanding of their fleet behaviors.

CHAPTER 2

DATASET OVERVIEW

2.1 Dataset Overview

2.1.1 Data Collection and Scale

What data is collected? We measure the health of the hardware by monitoring our **assets** (machines) running two benchmarks, CoreMark-PRO [3] for CPU benchmarking and Stream [46] for memory benchmarking. We chose these two representative benchmarks because of their lightweight footprint, stability, and popularity. Each of these benchmarks produces a **score** reflecting the health of the hardware. For each asset, we collect **20 features**, detailed in the next section. When an asset is benchmarked, all other processes are halted to ensure the high fidelity of the benchmark results.

How big is the dataset? We have two datasets, which we simply call **old** and **new** datasets. These two datasets unfortunately *cannot* be combined currently because they use different hashing algorithms when the data is anonymized, as detailed later in the anonymization paragraph. **Table 2.1** describes the scale of these datasets. We define an **observation** as the pair of results from running the two benchmarks above sequentially on an asset.

How long and often is data collected? Each dataset is collected over at least one month and contains 6,261-89,562 observations per day (27,796 on average). Every asset is benchmarked ("**observed**") 1-2 times on average.[1] More specifically, for the old dataset, nearly 98% of the assets are observed exactly once/twice, while only 0.14% (630 assets) and 0.02% (89 assets) have more than 10 and 100 observations, respectively. There are more heavily benchmarked assets in the new dataset as only 94% of assets fall into the first category, and 0.6% (11,358 assets) and 0.04% (832 assets) for the two latter categories. Benchmarks are run infrequently on production machines due to the cost of running them (servers must be drained to ensure high fidelity).

1. We use "asset/machine" interchangeably, as well as "benchmark/observe/test."

Scale	Old data	New data
# Observations	719,739	3,106,525
# Assets	435,754	1,882,518
Duration (days)	30	92
# Observations per day	23,993	33,048
# Observations per asset	1.65	1.65
# Features	20	20

Table 2.1: **Dataset scale (§2.1.1).** *The scale of the old and new datasets. The new dataset is more than four times larger than the old one, and has higher daily observation count.*

How is the data anonymized? Scores are normalized to between 0 and 1 (the higher, the better), which suffices to our analysis as it compares their relative performance. For legal and security reasons, both datasets are anonymized using deterministic and reversible methods to enable follow-up internal investigation. The two datasets are not merged since they are obfuscated in different fashions, for the following reason. Initially, COMPANYX used hash values that do not preserve temporal information. For example, when comparing two different kernel hash values in the first (old) dataset, we cannot tell which represents the newer release, and thus progression/regression analysis cannot be performed. Given this feedback, in the second (newer) dataset, COMPANYX uses numeric mapping (*e.g.*, 1, 2, 3) to retain the temporal relation. Because of this, and the time gap of 6 months between the two datasets, we analyze both datasets separately.

2.1.2 Data Features

This section describes the 20 features available in our datasets. For simplicity of presentation, we categorize them into four major classes: test, asset, CPU/ memory, and deployment features. For non-sensitive feature values that are not hashed, we list some of them inside curly brackets for better clarity.

Test features: This set is about the test itself. `TestType` has two values {CPU, memory} representing the CoreMark-PRO and Stream benchmarks, respectively. `TestName` extends the former feature with specific version information. For the two datasets that we have, the version remains

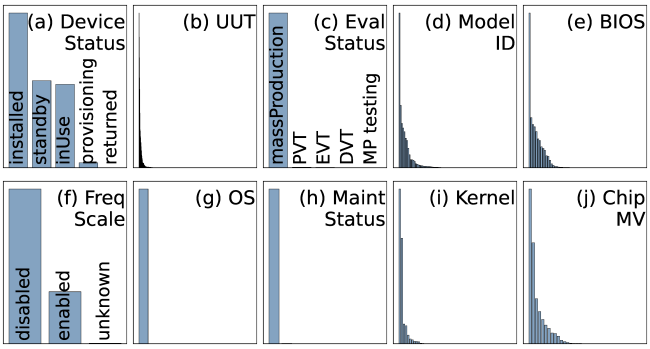

Figure 2.1: **Feature value count/distribution (§2.1.3).** *x-axis is the unique values of the feature, and y-axis is the count (anonymized), which is sorted.*

unchanged for both throughout the dataset. `TestScore` is the resulting score, depending on the test type. `EventTime` and `TestDuration` record when the test started and how long it took, respectively. Finally, `RunUUID` is a unique identifier for every test, shared by both the CPU and memory tests that together comprise one observation.

Asset features: These features describe the asset being tested. `AssetID` is a unique identifier for every asset. `ModelID` represents an asset's model. `UUT` (unit under test) captures aggregate information of the hardware being tested. For CPU, the UUT is a hash of the CPU model number, frequency, and microcode name and version. For memory, the UUT is a hash of DIMM count, vendor name, and model number. `BIOS`, `Kernel`, and `OS` fields are hashes of BIOS version, Linux kernel release, and OS name, respectively.

CPU/memory-related features: For CPU, `ChipMV` represents the firmware version of the co-processors that are designed to assist the main CPU and `FreqScale` denotes whether CPU over-clocking is {enabled, disabled or unknown}. For the new dataset, the latter has an additional value {lowFreq}. For memory, the trace also has `DIMMCnt` and `MemSize` containing the hashed DIMM count and memory size, respectively.

Deployment features: Our dataset also includes deployment-related information such as `DeviceStatus` with possible values of {installed, inUse, provisioning, returned, and standby}; `EvalStatus` {massProduction, EVT, DVT, PVT, or MP testing} representing codes for product development stages in a standard New Product Introduction (NPI) process [2]; `MaintStatus` (maintenance status) {none, draining,

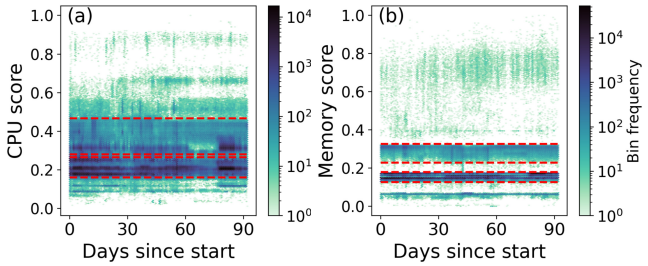

Figure 2.2: **Test scores over time (§2.1.3).** *CPU and memory performance scores over time. The log-scale color bar describes the number of observations in each bin. Darker color indicates more observations per bin. The dark bands highlighted by the red lines (two bands in each figure) are the two most common score ranges for CPU and memory tests.*

in-repair, testing}; and `LemonStatus` {true or false}, denoting whether the asset has been flagged as an "outlier" with a basic heuristic: either of the scores is 40% less than the population peak, or is two standard deviations away from the population mean.

2.1.3 Data Statistical Summary

Before we dive into technical methodologies, this section summarizes the high-level statistical information and provides a graphical overview of the dataset.

First, every bar graph in **Figure 2.1** shows the value distribution of a feature in the old dataset using histograms sorted by frequency. Every bar in the x-axis represents a possible value of the feature and the y-axis represents the relative size (values are not shown for a more compact view). For string/hash-type values, we sort the values (the bars) from the highest to the lowest. For example, in Figure 2.1a, the `DeviceStatus` field contains roughly 47% of `installed` (the plurality), 26% of `standby`, 25% of `inUse`, and so on. As another example, Figure 2.1f shows that the `FreqScale` is disabled most of the time (75% disabled vs. 25% enabled). Bars that are almost invisible represent a very low count.

Second, **Figure 2.2** shows a temporal view of the CPU and memory scores over the duration of our dataset. We plot the data using hexbin with 100-group configuration, meaning that the full range of scores from lowest to highest is divided into 100 groups with equal score range, *e.g.*, {0–0.01,0.01–0.02,...} within the normalized range 0–1. Hence, a dot in the figure represents a range

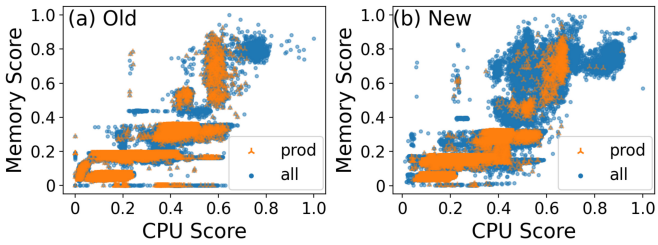

Figure 2.3: **Joint CPU/memory scores (§2.1.3).** *x-axis is CPU score and y-axis is memory score (normalized). Each blue dot is an observation from the dataset without any filter and each orange mark is an observation from production. There are more high score observations in the new dataset.*

and the color represents the density (the number of scores observed within the range). The most common CPU score ranges are 0.16–0.27 and 0.28–0.47 (indicated by the red lines in Figure 2.2a) while it is 0.13–0.18 and 0.23–0.33 for memory scores (indicated by the red lines in Figure 2.2b) for the new dataset. Higher CPU and memory scores are observed only intermittently, indicating that these are likely the results of a few models or even a few assets. The memory bands are also tighter, indicating that the memory scores may be highly predictable under certain conditions. The trend is relatively the same for the old dataset while there are fewer high-score observations.

Finally, **Figure 2.3** shows a spatial view of the combined CPU-memory scores observed in every test. The score is normalized as well. Here, each dot represents a unique observation. The blue dots are all the observations while the orange ones are the production observations (as later specified in Section 3.1.5). Although the CPU and memory tests themselves should be effectively independent, one might expect that machines designed for high performance will display both high memory and CPU performance. While this trend largely holds true, the pattern does not always show a simple and obvious relation between the two. For example, in both datasets, for the middle memory band (around `y=0.2`), the CPU scores vary extremely between `x=0.1--0.6`. The trend of the production observations aligns with the general population.

CHAPTER 3

FEATURE SELECTION PROCESS

3.1 Feature Selection Process

Given a large dataset with 20 features, the next question to address is which features would give us the most information. The conjecture is that perhaps not all features would matter equally to the test scores and not all of them would give us insights that could help identify defects or unstable assets and models. Defining peers among features is the first step for reaching this goal: should we group the observations/assets based on `ModelID`, `DeviceStatus`, or other features? Imprecise grouping will lead us to incorrect analysis. The next section describes our methodology for deciding which features provide more insight into performance (§3.1.1). We next identify the first feature group (§3.1.2) and then repeat this methodology iteratively to explore dynamic grouping that considers various feature dimensions, rather than just a fixed set (§3.1.3).

3.1.1 Information Gain and Entropy

To rank the features by their importance, we use two methods, *entropy* and *information gain*. Entropy [56] measures the uncertainty contained/captured by the data. The higher the entropy is, the more potential information/noise is contained within the collected dataset. On the other hand, low entropy indicates high uniformity, and such features can be excluded. While entropy is not about predictiveness, its value will guide us in discarding features with low significance, as we describe with more examples below. Information gain [29], which builds on top of entropy, is a technique that ranks given an output value to predict (*e.g.*, the `TestScore` in our case), which of the input features will give us the highest level of predictiveness (*i.e.*, the highest information gain).

The four graphs in **Figure 3.1** show the resulting information gain and entropy (left and right) for CPU and memory tests (top and bottom), respectively. From these results, we can now categorize them into determining, significant, negligible, and "everything-in-between" (but could poten-

tially be useful) features.

"Determining" feature: A "determining" feature is one that has the largest impact on an asset's performance prediction. Therefore, these features are the first we should use to break and group the assets. Figure **3.1a** shows that `ModelID` gives the highest information gain for both CPU and memory tests. COMPANYX's engineers, as the **domain experts**, also confirm that, indeed, the hardware performance is highly dependent on the asset's `ModelID`. In other words, the performance of all assets within the same `ModelID` should be similar or, at least, should fall within an expected range, and the comparison of asset performance across different `ModelID`s will not yield valuable insights and is not considered a high priority (also confirmed by the domain experts). For the entropy, Figure **3.1b** also shows that `ModelID` has a relatively high entropy compared to other features. This is because `ModelID` has large amount of unique values (as shown in Figure 2.1d) instead of having one or two dominating values (such as Figure 2.1c), confirming that `ModelID` could potentially be highly informative.

Significant features: After `ModelID`, we see other features with high entropy, such as `BIOS` and `ChipMV` versions. Hence, they are the candidates for **subgrouping** in subsequent steps (§3.1.3) for a tighter and finer-grained analysis.

Negligible features: The figure also quantifies features that have almost no impact such as `OS`, `MaintStatus`, and `EvalStatus`. For example, the `OS` name has almost zero information gain and entropy because a single OS name appears in more than 99% of all the assets (as shown earlier in Figure 2.1g).

"Everything in between:" Finally, there are "everything-in-between" features that neither have high nor low information gain or entropy, features such as `MemSize` and `DIMMCnt`. We can triage these features as needed. In our analysis, we get important insights with just the significant features, so we tend to deprioritize these "in-between" features unless later steps suggest otherwise.

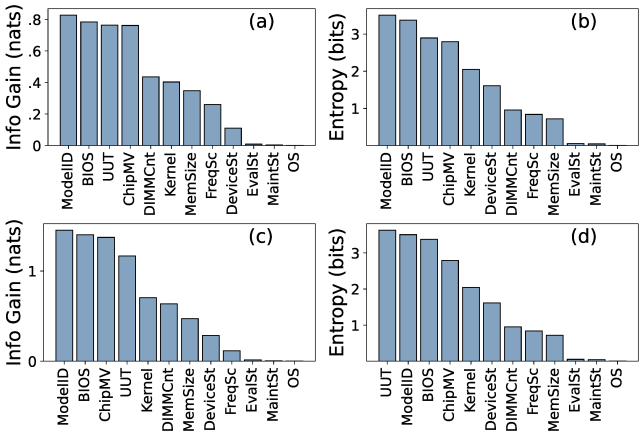

Figure 3.1: **Information gain (§3.1.1).** *The first row is the information gain and entropy for CPU tests and the second row is for memory tests. High information gain indicates potential predictiveness and low entropy hints on minimum level of insight that could be drawn from the given feature.*

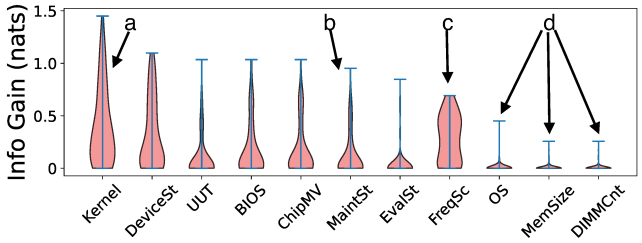

Figure 3.2: **Group by model ID (§3.1.2).** *Info gain (CPU) after grouping by model ID. There are four types of shape.*

3.1.2 Top-Level Grouping with Model ID

Based on the previous section, we now group the assets by their `ModelID`. Among these model groups, which represent a class of similar machines, the sampling distribution is considerably less skewed. As shown in Figure 2.1d, out of the 87 total models (on the x-axis), 8 have more than 20,000 observations, 12 have between 2,000 and 20,000 observations, and 17 have between 200 and 2,000 observations. Although there are a few infrequently observed models that each is only corresponding to a single asset, a typical model is associated with many assets—the median model group contains 133 observations and 87 assets. This more balanced sampling enables meaningful inter- and intra-model comparisons, and, for this reason, `ModelID` forms the basis of the grouping step.

We next run the information gain and entropy formulas again after we group the assets by

ModelID. **Figure 3.2** shows the result of information gain for CPU test. Note that this time, we can show a *violin plot* (vs. the simple bar graph in Figure 3.1). The violin plot is a visual representation of each feature's intra-group information gain distribution. Their shapes provide useful information which we categorize into three types.

First, label **3.2a** in the figure points to the Kernel feature with a high information gain and a *cone* shape. This shape implies that the values of Kernel among the assets within each ModelID are diverse, not highly skewed to one value. This suggests that we can further **subgroup** the assets within a ModelID group by their Kernel versions for a tighter analysis.

Second, label **3.2b** points to the MaintStatus feature that can reach a relatively high information gain *but* unlike the previous one, the violin shape is more skewed to the bottom of the vertical line, suggesting that this feature is relatively homogeneous and hence may not be a useful candidate for subgrouping the assets further. There are other features that look similar, such as EvalStatus. For these two status features, interestingly while in the previous section, they almost show no information gain, in this section they show some gain, but their shape suggests limited predictive power.

Third, label **3.2c** reveals an interesting "**8-shape**" violin for the FreqScale status. The violin shape tells us of an almost *pure bimodal distribution*, strongly reflecting that both sides of the values can provide strong predictiveness. Indeed, that is the case for CPU tests; within a ModelID population, assets with FreqScale=On will deliver higher CPU scores compared to the other assets with FreqScale=Off. Hence, although the information gain is relatively "medium" and its entropy is low (see Figure 3.1b), we should provide special treatment for the FreqScale feature during subgrouping, for a more precise analysis. Note that this special treatment is only valid for analyzing CPU tests later on. For memory tests, FreqScale does not display the 8-shape violin (not shown in the figure), which is indeed the case as the feature does not have any noticeable impact on memory scores (as verified by the domain experts at COMPANYX).

Finally, label **3.2d** points to features with low information gain such as OS, DIMMCnt, etc., con-

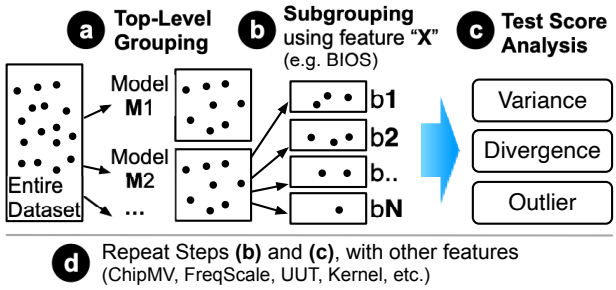

Figure 3.3: **Subgrouping methodology (§3.1.3).** *Four step subgrouping is used before feeding data to the downstream.*

firming again that they will not be relevant even after we group assets by `ModelID`.

3.1.3 Subgrouping Methodology

Based on the findings from the previous section, we will use the following **5 subgrouping (second-level) features**: `BIOS`, `ChipMV`, `FreqScale`, `UUT`, `Kernel` for a more fine-grained analysis. **Figure 3.3** illustrates our subgrouping methodology. **(a)** We categorize the entire population (observations) based on their `ModelID`, acting as the top-level grouping. **(b)** We then take the population of each `ModelID`, and subgroup them further based on each of the subgrouping (second-level) features mentioned above. Thus, when we say that we "*subgroup using feature X*," it means we break the assets with the same model ID into subgroups. For example, when we use the `BIOS` feature (with N possible `BIOS` values), then there will be N subgroups, *e.g.*, `ModelID=m1,BIOS=b1`, `ModelID=m1,BIOS=b2`, ..., `ModelID=m1,BIOS=bN`. Then in **(c)**, we compare their score behaviors using various methods such as variance, divergence, etc., as explained in the next section. We then **(d)** repeat the same process by subgrouping using other features.

Subgrouping is important in our analysis because we cannot just compare assets based on their model ID. Throughout the lifetime of an asset, the model ID will remain the same, but the other (second-level) features might change. Furthermore, within the same model ID, the second-level features likely have varying values (*e.g.*, assets of the same model might use different BIOS versions). Beyond subgrouping (2nd-level grouping), we do not perform a 3rd level grouping

because the population of each "sub-subgroup" would be overly small and prevent statistically meaningful conclusions.

3.1.4 Analysis and Recommendations

Based on the subgrouping method, we will build three analysis modules to cover anomalous behaviors at model, subgroup and asset levels, as defined below.

Variance: The variance module in Section 4.1 *identifies model(s) with abnormally high variance*. Hence, this module focuses on the detection at the *model level*. Although different models tend to vary in their overall performance, reliability remains a key concern despite the expected cross-model variability. For example, a model with high variability could negatively impact workload balancing even though its performance is higher than the majority of the models. Therefore, model-wide variance is used to capture the unreliable models and lower-level root-cause analysis is performed to further assist the triage process.

Divergence: The divergence module in Section 5.1 module *enables pairwise performance comparisons at the subgroup level to identify highly divergent subgroup pairs.* This module is based on the assumption that peer subgroups that are within the same model group should display high similarity. While the variance module detects top-level anomalies, the divergence analysis goes beyond that level, and provides a more comprehensive as well as refined anomaly detection, which can highlight performance differences with even subtle distributional drifts.

Outlier: While the previous two modules cover the model-level and subgroup-level anomalies, the outlier module in Section 6.1 *detects the unexpected performance degradation over time at the asset level*. This is done by locating the outlier observation cluster that shows downgraded performance. When the detection entity is a model group or a subgroup, the outlier cluster detection could be highly inaccurate due to the natural variety–a normal benchmark score from one asset could fall in the "anomalous" range for another one, although their configurations are almost identical, and the outlier benchmark score can also be hidden for the same reason. Therefore, this

Figure 3.4: **Variance pipeline (§4.1).** *Two-stage variance pipeline to detect model IDs with high variance.*

detection focuses on the detection at the finest level for the best accuracy.

In the remainder of this paper, we use the following **terminologies**: **pY** means the Y% percentile, *e.g.*, p25 means the 25th percentile. To improve readability, when naming a group or a subgroup, we shorten the hash (anonymized) values into two characters with one lower-case prefix that represents the feature. For example, subgroups `b0X` and `b93` represent two subgroups with different BIOS ("b") hash values "0X" and "03." We use "v" in "*AvB*" to imply we are comparing subgroups *A* vs. *B*.

3.1.5 Production Only Focus

After confirming the feature selection result with the domain experts at COMPANYX, we prioritize analyzing "in-production" assets, given their importance to end users. Thus, we apply a "production" filter, specifically defined by three features: `DeviceStatus=inUse` && `EvalStatus=massProduction` && `MaintStatus=none`.

For the first feature above, even though `DeviceStatus` information gain is high as shown in Figure 3.2 (hence, might be interesting to analyze), we focus only on `inUse` assets because the implication of other `DeviceStatus` values is already known from the domain experts' view (*e.g.*, `DeviceStatus`=provisioning represents assets that are still being set up but not ready for full production use, hence are not currently useful to be analyzed).

The latter two features (`EvalStatus` and `MaintStatus`) are selected because of the insight provided by their values based on Figure 3.1, where the information gain of both features is near zero, and Figure 3.2, where the highly skewed distribution reveals homogeneity. Hence, COMPANYX recommends us to focus on just the assets with `massProduction` evaluation status and no ("`none`") special maintenance status, and leave the rest as future work.

In addition, the production observations are more stable than the rest, such as ones drawn from testing assets, where configurations would change more drastically and frequently. This trend can be observed in Figure 2.3, where the pattern of the production joint scores is less noisy than the overall population.

With this filter, in the next two sections, our results are based only on production assets, more specifically a total of 878,768 observations across 772,569 assets to analyze. Even though our detailed analysis is restricted to this prioritization, we will release the entire dataset to support many possible future directions (*e.g.*, understanding performance instability of `inRepair` assets).

CHAPTER 4

TOP-LEVEL VARIANCE

4.1 Top-Level Variance

We now begin with "top-level variance," with the goal of finding specific model IDs whose population exhibits high benchmark score variance. **Figure 3.4** shows our two-stage variance pipeline. First, we use a *knee detection* threshold-based algorithm (§4.1.1) to select models that have a high variance of performance scores. For those that are above the threshold, we then apply *inter-quantile range (IQR)* analysis (§4.1.2) on the subgroups within each of them. The outcome of this pipeline is three types of recommendations, explainable, semi-explainable, and non-explainable variance (§4.1.3).

4.1.1 Knee Detection

For each model ID, we first compute its score variance among all the assets in the model. We normalize the variance by $N-1$ so that it is unbiased [5]. **Figure 4.1** shows the sorted test score variance (in the y-axis) from the highest to the lowest for every model ID (anonymized in the x-axis). High variance implies some inconsistency among the assets in the model. We then run a knee detection algorithm, Kneedle [51] with a configurable sensitivity level. The sensitivity level that yields the most stable threshold is chosen, meaning changing the value slightly will not cause a drastic difference in knee threshold. The resulting knee threshold is the bold dashed horizontal line.

Finding #0: *Memory test scores are more stable (less variant) than CPU test scores* . In **Figure 4.1b**, the memory score variance has a clear "knee" with a spike in variance on the left side and many low variances on the right side of the knee. However, in **Figure 4.1a**, the CPU score variance is less stable across various models. `FreqScale` status is one of the reasons for this variance instability, we will discuss more on `FreqScale` later. The cases above the knee threshold

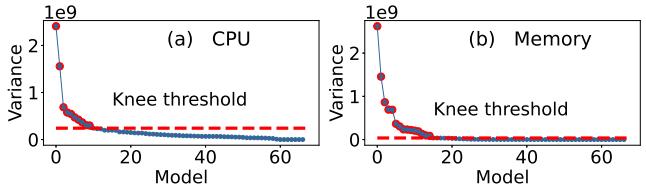

Figure 4.1: **Knee thresholds (§4.1.1).** *Each dot represents a model ID's score variance (sorted). The model IDs with score variance above the threshold will be analyzed in later steps.*

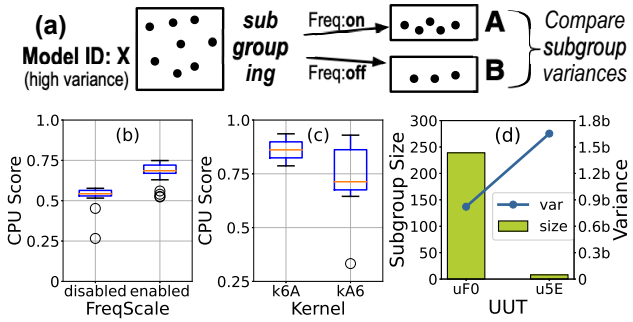

Figure 4.2: **Subgroup variance and IQR analysis (§4.1.2).** *(a) The subgrouping method, (b) no IQR overlap, (c) IQR overlap, and (d) higher variance but smaller size subgroup.*

will be streamlined to our next analysis modules.

4.1.2 (Non)Explainable Subgroup Variance and IQR Analysis

Models with high variance do not necessarily imply anomalies, as long as they are "explainable." For example, due to the impact of `FreqScale` on CPU performance, it is reasonable that a model involving a mixture of `FreqScale` enabled and disabled assets displays higher variance than one having solely enabled assets. Thus, to differentiate explainable and non-explainable variance, we perform *subgroup variance and IQR analysis*. As illustrated in **Figure 4.2a**, for every model ID with high variance (*e.g.*, model *X*), we take all the assets and categorize them into subgroups based on a series of important features that have been selected in the feature selection process (Section 3.1.3). For example, the figure shows the `FreqScale` feature where we categorize the assets of a model ID into two subgroups *A* and *B* (`On` and `Off` values, respectively). We then compute the score variance in each of the subgroups and compare them, and depending on the result, we also perform IQR analysis. The three outcomes are as follows.

Case 1 (explainable via no IQR overlap): In this case, the variance of subgroup A is similar to B ($vA \sim vB$), hence the feature being used is not helping out. Thus, we use *inter-quantile range* (IQR) analysis to see if the test score ranges in A and B *overlap* or not. More specifically, if the p25 score in A is above the p75 score in B, there is no overlap. The whisker plot in **Figure 4.2b** illustrates this using the `FreqScale` feature example. The left whisker bar (representing the assets with `FreqScale=On`) has higher CPU performance scores and their range does not overlap with the right bar (`FreqScale=Off`).

Instances of Case 1 are marked as *explainable*, because the tool has identified two subgroups with non-overlapping p25-p75 score ranges as a source of the high variance in this model ID. The tool does not throw alerts for these occurrences. The percentile values are also configurable (*e.g.*, to p20-p80).

Case 2 (non-explainable due to similar variance and IQR overlap): Following on the IQR outcome above, another outcome is an overlapping IQRs from the two subgroups (*e.g.*, in **Figure 4.2c**), while at the same time, they have similar score variance. In this case, we mark them as non-explainable and punt them to the domain experts to be analyzed.

Case 3 (semi-explainable by one high-variance subgroup): Here, the variance of subgroup A is higher than B ($vA \gg vB$), regardless of their population sizes. By "higher," we mean the variance difference between the two subgroups is above the 50^{th}percentile (which is configurable) among all the subgroups. In this case, the tool recommends the domain experts to just focus on analyzing assets in subgroup A as opposed to the entire assets with this model. For example, in one model, the subgroup with UUT feature with `u5E` (anonymized) value has a high variance (as compared to the other subgroup `uF0`). The domain expert analyzes that and agrees to monitor this subgroup in the future.

While so far we use an example of two subgroups A and B, in reality, some feature has multiple values, such that there will be multiple subgroups (*e.g.*, the `BIOS` feature has 36 unique values). To generalize our approach to more than two subgroups, our tool performs all the pairwise compar-

	CPU	Mem
# of model IDs	120	119
# of high variacnce model IDs	45	17
# Case 1 (expl., no IQR overlap)	26	7
# Case 2 (non-expl., IQR overlap)	8	3
# Case 3 (semi-expl.)	11	7

Table 4.1: **High variance recommendations (§4.1.3).** *Recommendations made based on variance analysis.*

isons, A vs. B, A vs. C, B vs. C, and so on, and sort them based on how many times ("votes") the subgroup has a higher variance than the other one. To assist further, we also add *subgroup size comparisons* to increase the priority of the votes (hence, the reports). For example, **Figure 4.2d** shows a higher priority report where A's variance is higher than B's (the bar graph) but A's population size is much smaller than B (the line plot), which is considered a statistical anomaly (*i.e.*, a smaller population is less likely to lead to a significantly higher variance).

4.1.3 Recommendations

Table 4.1 shows the overall results of variance analysis, combined for both the old and new datasets. For CPU and memory tests respectively, the knee algorithm delivers a total of 45 and 17 model IDs that have high variance, the IQR analysis removes 26 and 7 explainable cases, and mark 8 and 3 cases as non-explainable, and finally, the variance pipeline reports 11 and 7 cases as semi-explainable given the high-variance dominance in one or some of the subgroups.

We also experiment with various configurations for the knee detection algorithm to measure the sensitivity. That is, if we relax knee sensitivity value, will it spuriously report too many high-variance models? We found that memory results are more sensitive (in a positive way) such that changing the configuration can lead to suddenly more results, meaning that there is a clear "knee" (as one can also see in Figure 4.1b).

CHAPTER 5

DIVERGENCES

5.1 Divergences

We now move from variance to divergence analysis. **Figure 5.1** shows the overall design. **(a)** Here, for each model ID, we break the population into subgroups (based on various features and values as defined earlier in Section 3.1.3) and compare their CPU/memory test scores in a pairwise manner. However, before applying various divergence methods, Section 5.1.1 will describe the importance of adding `FreqScale` as a second top-level feature for CPU test score. **(b)** We apply various divergence methods, Kullback-Leibler divergence[30], Jensen-Shannon divergence[38], and Wasserstein distance[28] as elaborated more in Section 5.1.2. **(c)** We then adjust the scores (denoising) with multipliers (weighting) based on subgroup population sizes, as motivated in Section 5.1.3. **(d)** and finally provide the recommendations, as shown in Section 5.1.4.

5.1.1 Modified Top-Level Grouping

As we embark on using divergence, we found a negative impact of the `FreqScale` feature on CPU test score divergence. To recap, Section 3.1.2 discusses the special "8-shape" pattern of the `FreqScale` in Figure 3.2. To show what happens, we pick a model ID, and divide it into two subgroups (`FreqScale=On` and `Off`) and show the CPU test score distribution in each of the subgroups in **Figure 5.2a**, which clearly shows enabling this feature will give CPU boosts.

The negative implication is shown in **Figure 5.2b** where we now break the same model ID's population based on the BIOS feature. The figure only shows two different BIOS subgroups for readability. Here we can see that *each BIOS subgroup exhibits a bimodal distribution* from the impact of the `FreqScale` status, which will lead to inaccurate analysis. In other words, merging the two `FreqScale` statuses may "dilute" the divergence and could thus be detrimental to the highly divergent pair capture.

Figure 5.1: **Divergence pipeline (§5.1).** *Multi-stage divergence measurement based on re-adjusted grouping. The multipliers are then applied for denoising.*

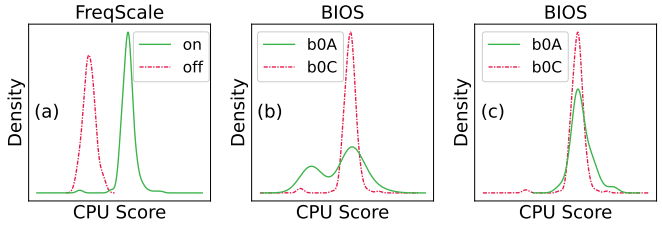

Figure 5.2: **Frequency scaling (§5.1.1).** `FreqScale` *is needed for grouping the CPU tests for more accurate divergence computation. From (b) to (c) shows an improvement.*

For this reason, we must treat this feature differently, but only for CPU tests (memory tests are not impacted by the `FreqScale` feature). Before, we only use model ID as the top-level grouping, but now for every model ID, we must further break the population of the model into two different groups (`FreqScale=On` and `Off`). Only after that, we can do the subgrouping based on other features more accurately. **Figure 5.2c** shows the same analysis in 5.2b, but now we only do so within the population with `FreqScale=On` within this model ID, which gives a cleaner result.

5.1.2 Divergence Cases and Scores

We use three divergence methods, Kullback-Leibler divergence (KLD), Jensen-Shannon divergence (JSD), and Wasserstein distance (WSD) to combine their different advantages as discussed below.

KLD

Kullback-Leibler divergence is a statistical distance measurement that tells how much a modeled distribution is different from its actual, reference probability distribution. Because of the definition of model and actual probability distribution, it is *asymmetric* and thus choosing the model and actual distribution plays a crucial role. It is good for measuring the information gain/loss between

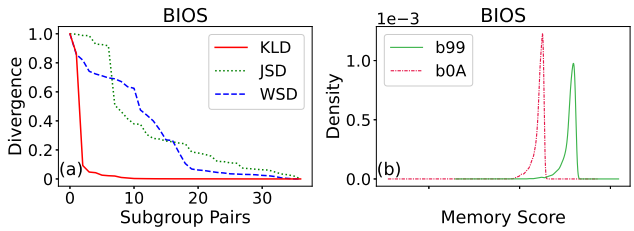

Figure 5.3: **KLD excels at separation (§5.1.2).** *KLD outperforms the others on highly divergent pair separation.*

the two aforementioned distributions. But due to the definition of this divergence measurement, it is undefined when the model distribution is evaluated to zero and could, therefore, be sensitive to the distribution choice. This method is preferred when there is a *clear difference* in role between the two distributions (*e.g.*, comparing a "newer" distribution to an "older" one).

Figure 5.3a shows the need for KLD, compared to JSD and WSD that we will describe later. Here the x-axis represents all the subgroup pairs (*A*-vs-*B*, *A*-vs-*C*, and so on) for the BIOS feature sorted based on their KLD, JSD, and WSD divergences in the memory scores (*normalized* using min-max scaling in the y-axis from 0 to 1). We can clearly see the sharp differences in the score distribution between these three measurements. KLD detects two subgroup pairs with extremely high divergence scores (between 130 and 160) while the 3rd pair's score is only 14 (a sharp drop in the red line).

Figure 5.3b dissects the highest KLD divergence case. The two lines represent the probability distribution function (**PDF**) of the memory scores of two subgroups with different BIOS values (`b0A` and `b99`, with the prefix "b" represents BIOS) within the same model, which clearly shows some separations. COMPANYX confirms the test result distribution drift and found that the performance downgrade is for expanding the option pool and cost models.

JSD

Jensen-Shannon divergence is the enhanced, smoothed, and *symmetric* version of KLD. It leverages the average between the two distributions and, as a result, unlike KLD, it is always defined. And because of its symmetric nature, the result is *not sensitive* to the actual/model distribution

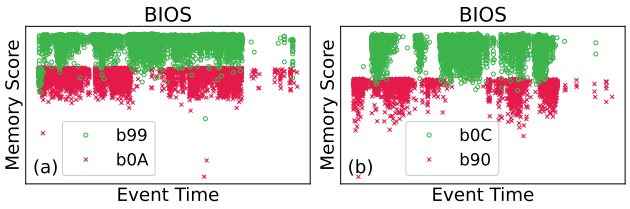

Figure 5.4: **JSD yields stable results (§5.1.2).** *JSD yields stable divergence value on similar pairs. The two pairs displayed here share almost identical JSD values yet have drastically different KLD scores.*

choice. These two characteristics together make JSD a *more stable* statistical distance measurement. If the difference between the two distributions is relatively vague, then JSD may be a better choice since it is symmetric.

To motivate JSD, let us look at **Figure 5.4** where we compare **(a)** `b90` vs. `b0C` subgroups and **(b)** `b0A` vs. `b99` subgroups. The x-axis represents the logging timestamp and the y-axis represents the memory scores. Both comparisons in (a) and (b) show similar behavior that these BIOS versions deliver different ranges with minimal overlap. The only difference is that the former (5.4a) is less evenly sampled temporally as compared to the latter (5.4b).

model: groupA vs. groupB	**KLD**	**JSD**
`mE8: b0C vs. b90`	3.1 (8^{th})	3.16 (2^{th})
`mE8: b0A vs. b99`	156.9 (2^{nd})	3.13 (3^{rd})

The table above shows all the KLD and JSD scores of these two pairs of comparisons (the two rows). Interestingly KLD marks *only* the second row with a high score (with the 2nd rank, as shown in the parentheses) while the first row exhibits a low KLD score (and ranked 8th). In other words, we would miss the first case if we only use KLD. With JSD however, both cases are marked equally strong. In terms of performance stability, JSD surpasses KLD as for the former, similar pattern yields consistent result.

WSD

Wasserstein distance is usually called the earth mover distance as it computes the minimal level of distance required for "moving" from one distribution to another. Like JSD, it is also symmetric.

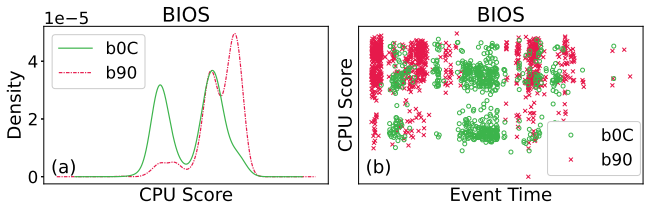

Figure 5.5: **WSD advantage (§5.1.2).** *WSD catches the case that is overlooked by the other two methods.*

Its biggest advantage is that it can be applied on the *raw data* (observed value in the empirical distribution) instead of the estimated density function. The distance-focused nature surpasses the other two methods on outlier detection yet it also causes WSD to be more sensitive to the extreme outliers.

Figure 5.5 motivates our use of WSD. Here, we have two BIOS versions (two lines) that show interesting results. In Figure 5.5a, we have the PDF (density) graph of their CPU scores. BIOS `b0C` is clearly bimodal (green dashed line) and `b90` is also multimodal (red solid line), with only one of the modes highly overlapping with the former. Figure 5.5b shows the raw data observed over time. However, if we see the resulting KLD, JSD and WSD scores in the table below, KLD marks this comparison at rank 18, JSD at rank 9, but only WSD is able to mark it relatively high (at rank 3). The dissimilarity between the pair is severely underrepresented when solely measured by KLD and JSD.

group A vs. group B	**KLD**	**JSD**	**WSD**
`mE8`: `b90` vs. `b0C`	1.10 (18^{th})	2.07 (9^{th})	4.47 (3^{rd})

5.1.3 Denoising with Multipliers

Although a handful of samples are sufficient to estimate any divergences, the population size of a subgroup can still play a significant role in the divergence measurement. The more observations are used for the estimation, the "closer" the model distribution would be to the actual distribution. *Divergence measurements involving more sufficiently sampled subgroups are considered more accurate and thus should be prioritized while denoising is needed for the insufficiently sampled ones.*

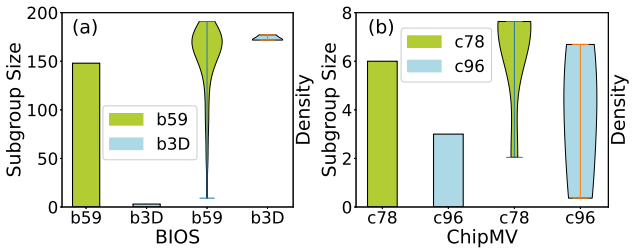

Figure 5.6: **Size matters (multipliers) (§5.1.3).** *Large size difference between the pair should be penalized due to potential inaccuracy. Pairs where both subgroups are insufficiently sampled should be deprioritized as well.*

Therefore, we do two stages of score adjustment based on (a) pairwise size and (b) groupwise size.

Figure 5.6a motivates the first case. Here we compare the sizes of subgroups A and B, but B's size is much smaller than A's size (148 vs. 3). Let us suppose their A-vs-B divergence score is high (*e.g.*, say "105"), but it is not a valid result based on the population—the actual distribution of B may be similar to A's and the high divergence is a result of insufficient sampling, in which case we would like to decrease the confidence of such cases. Hence, we introduce the first "multiplier" (weighting) with a simple formula: `gMultiplier1`$=$`gSmall`$/$`gLarge`.

Here, `gSmall` is the size of the smaller subgroup of the pair and `gLarge` is that of the larger one. Thus in this particular example, the multiplier is very small ($3/148$), which is then applied to the divergence score, resulting in small value as compared to the pairs that have roughly even subgroup sizes.

Figure 5.6b motivates the second case where both sizes of subgroups A and B are small relative to other subgroups such as C. Here, we leverage the median subgroup size within one model, denote `gMedian`, to address the issue. The median is used for being outlier-resistant. For this multiplier we want to penalize the small pairs and boost the large pairs as a way for denoising, with the assumption that the pair with huge size difference has already been taken care of by the aforementioned multiplier. This second "multiplier" is calculated based on the same `gSmall` and `gLarge` along with `gMedian` and a configurable adjustment factor w:

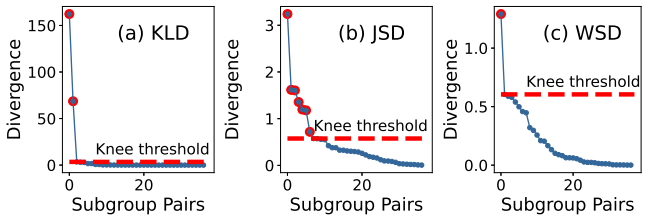

Figure 5.7: **Recommendations (cutoffs) (§5.1.4).** *Each dot represents a subgroup pair's adjusted divergence value. The knee detection functions similarly as in 4.1.1.*

$$\texttt{gMultiplier2} = \begin{cases} 1 - w \text{ if } \texttt{gLarge} < \texttt{gMedian} \\ 1 + w \text{ if } \texttt{gSmall} > \texttt{gMedian} \\ 1 \text{ otherwise} \end{cases}$$

Thus, in this case, regardless of the *A*-vs-*B*'s divergence score, their score will be lowered if the sizes of both are below median, increased if they are above, and remain unchanged otherwise.

5.1.4 Divergence Recommendations

To illustrate our divergence recommendations, **Figure 5.7** shows the divergence scores when using `BIOS` for subgrouping for memory tests (on the old dataset). Each dot in the x-axis represents a subgroup pair and the y-axis represents the divergence score (*e.g.*, the divergence result of `BIOS` version *A* vs. *B* in a model ID *M*). There are 37 pairs in the x-axis. As discussed, the three figures display the different natures of the KLD, JLD, and WSD results. We then apply the knee threshold (similar to Section 4.1.1) to decide which pairs are "high." In the figure, for KLD, JSD, and WSD, there are 2, 7, and 1 pair(s) classified as highly divergent pairs among all 37 pairs. Other than `BIOS`, we also have 15 more sets of graphs (not shown for space) for 3 other features on both datasets.

Table 5.1 shows the overall number of recommendations that the tool reports to the domain experts. The rows are the different features we use for subgrouping (`BIOS`, `UUT`, etc.). The "Total" column represents the total number of subgroup pairs created (*AvB*, *AvC*, and so on) using the

Subgrouping Feature	Total Pairs	#Pairs above threshold KSD	JSD	WSD
BIOS	374	33	24	34
Kernel	2744	62	125	86
CMV	304	15	44	35
UUT	7068	112	24	78

Table 5.1: **Divergence recommendations (§5.1.4).** *Aggregated across the CPU and memory tests in both old and ne*

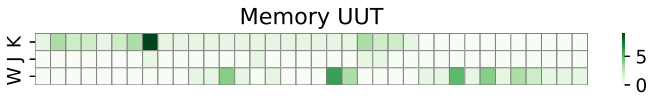

Figure 5.8: **Heatmap (§5.1.4).** *Each cell represents a subgroup defined by* `UUT` *and the color indicates the total number of above-the-knee occurrence.*

corresponding feature across all the models. The rest of the columns show the number of subgroup pairs that are above the knee thresholds based on their KLD, JSD, and WSD scores. Overall, each divergence method only reports 1.7-25% of the total pairs (the median is 8%), which are manageable. When reporting to the domain experts, we also provide the sorted divergence scores so the domain experts can prioritize/triage the cases.

Our tool also provides other types of visualization such as the heatmap in **Figure 5.8**. For space, we only show one subgrouping feature, `UUT`, for memory tests on the old dataset (other than this, there are 15 more figure sets). The x-axis represents the subgroups (*A*, *B*, *C*, and so on) that appear above the knee thresholds. The heatmap color represents the count of how many times the subgroup appears in highly divergence pairs, regardless of their model IDs. This way the domain experts can see if a feature value is causing a divergence in all models. For example, the darkest cell in Figure 5.8 shows a count of 8. This is because there is a specific unit under test (UUT) whose memory test is in a very tight range compared to the other 8 memory units.

CHAPTER 6

OUTLIERS

6.1 Outliers

Lastly, we describe our outlier analysis pipeline and how we compose several statistical techniques to denoise the results and provide more accurate analysis. Unlike our variance and divergence pipelines, the outlier pipeline focuses on *asset-level* analysis.

6.1.1 Outlier Assets

For clarity, we begin by showing sample cases outputted by our pipeline. **Figure 6.1** shows **(a)** a confirmed performance degradation outlier is displayed where the CPU score dropped by 47% *without* trackable configuration change and **(b)** another case with noticeable performance change (increase), which is caused by `FreqScale` status being turned on. These two cases can be easily caught by existing techniques such as DBSCAN [13]. However, using solely off-the-shelf statistical techniques will lead to high false positives, which will be overly costly in the production environment. Thus, we combine and adapt the current techniques for removing the noises.

6.1.2 Outliner Denoising Pipeline

Figure 6.2 shows the design of the pipeline, which is a composition of various statistical techniques, targeting to reduce the false positive rate. First, in **Figure 6.2a**, we only take assets with at least 50 observations to allow the clustering to work properly, and then, as a standard preprocessing step of any clustering method, the min-max scaling is applied to `EventTime` and `TestScore`, the two fields that will be used for the remaining of the clustering steps.

Now we move to the top part of the pipeline, shown in **Figure 6.2b-d**, where we use the combination of DBSCAN, binary clustering and cluster centroid distance check. To motivate these

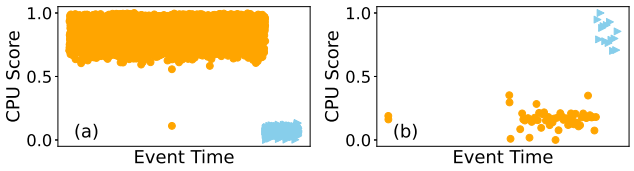

Figure 6.1: **Positive outlier cluster examples (§6.1.1).** *The outlier cluster indicates performance change.*

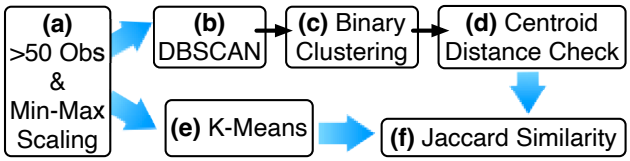

Figure 6.2: **Outlier analysis pipeline (§6.1.2).** *The top part of the pipeline identifies outlier cluster, which is then validated by the bottom part.*

steps, let's use the case in **Figure 6.3a** that exhibits a sparse observation, yielding three clusters by the DBSCAN. But since our goal is to detect performance outliers, two clusters would be sufficient. Note that if DBSCAN only sees one cluster, the case is considered a good case and thus excluded.

After we perform binary clustering, the same case will result in **Figure 6.3b** where the binary clustering sets the cluster with the largest average `EventTime` as cluster #2 and all the other clusters are merged together as cluster #1. But a binary clustering does not directly lead to outlier indication. If either cluster only contains single observation, the case is not considered either.

To further denoise, centroid distance check is then performed. To motivate this with better clarity, **Figure 6.3c** shows a different case study with more observations. Here, we compute the centroid of the first cluster (denoted by the red cross, c_1) and the fluctuation distance (denoted by the green band, d) with the latter defined by two standard deviations of this cluster's test score. Then the centroid of the second cluster is calculated (denoted by the purple cross, c_2). If c_2 is within the fluctuation range of c_1 ($c_1 - d \leq c_2 \leq c_1 + d$), then the performance change is considered normal and this asset will not be recommended to be checked.

Even though the aforementioned steps (the top part of the pipeline in Figure 6.2) can eliminate most of the noises, false positives are still observed when the borderline between the two clusters is vague. For this reason, we augment the pipeline with K-means (Figure 6.2e). To motivate this,

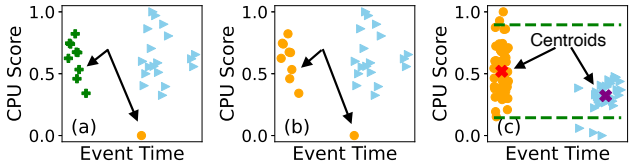

Figure 6.3: **DBSCAN and centroid distance check (§6.1.2).** *(a) and (b) shows how the clustering result (dots in different color) from DBSCAN is transformed into binary. (c) displays the fluctuation range and how c_2 falls within the range.*

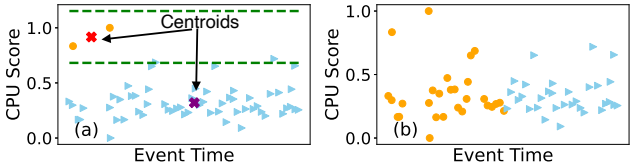

Figure 6.4: **Similarity Stage (§6.1.2).** *Jaccard similarity is applied to the result from both clustering methods to eliminate borderline cases. Cluster is indicated by color.*

let's use **Figure 6.4a** as an example where the binary clustering result from DBSCAN passes the centroid distance check since c_2 falls out of the fluctuation range, however this asset is actually not displaying any abnormal performance change. Therefore, we also apply K-means [41] on the same case, which leads to the clustering result in **Figure 6.4b**, where the centroids of the two cluster share similar y values.

The outputs of the DBSCAN and K-means algorithms are binary strings (*e.g.*, {`0,1,0,1,0,...`}), representing the clustering classification of each observation of the asset. Thus, we use Jaccard similarity [25] (step **Figure 6.2f**) to analyze the outputs of the top and bottom steps of the pipeline. Only when the two results converge (that there is an outlier cluster), the tool would emit recommendations to the domain experts. For example, this case study will be rejected because the clustering results in Figures 6.4a and 6.4b do not converge as decided by the Jaccard similarity.

6.1.3 Recommendations

Out of over 2 million assets, we only take 3,078 of them as they have at least 50 observations. Our outlier pipeline only recommends 28 assets to be further investigated (14 for memory and 14 for CPU) because of the highly efficient denoising process. For example, for CPU, 1,363 assets are eliminated by the binary clustering, and cluster centroid distance check can remove 1,639 assets.

Among the remaining assets, 62 of them are excluded during the Jaccard similarity comparison stage.

CHAPTER 7

KEY OBSERVATIONS

7.1 Key Observations

In this paper, we take a performance benchmarking dataset from FREP, and conduct our investigations via a single-blind research approach. Given the heterogeneity of the datasets, we iterated upon various methods to detect performance outliers and degradation within a dynamic fleet. We present the following key observations as a result of our work on methodology evolution.

Observation 1: *Performance benchmarking at scale is a hard problem.* Performance benchmarking at scale requires that the capacity is continuously cycled through the benchmarks thereby introducing a cost of fleet monitoring. In addition, the collection of features is a complex choice, as collecting too many features increases the state space, and collecting too few can lead to unexplained behaviors. Optimizing for the right feature set requires consideration of features across hardware, software, and testing environments to build a good evaluation system. We also notice that data extracted from a live fleet can have unpredictable sampling density and large variance in data counts across the entire population with some assets benchmarked only once while some assessed numerous times because of either machine states or the traversal across them of which is an artifact of a large fleet. In addition, in certain virtualized environments, the benchmarks could be running on concurrent systems and result in corrupted scores, so isolation is a key requirement. All these combined together make performance benchmarking at scale a hard problem.

Observation 2: *Understanding large-scale datasets requires efforts in state space reduction, correlation techniques along with manual fine-tuning.* We observed from our studies that while we started with 20+ features, certain features have higher entropy and the others do not display the same trend. Ranking features along with leveraging methodologies to achieve sound selection plays a key role in benchmarking. In our study, we continuously iterated to rank features based on domain expertise as well as data at hand using variance, divergence, and outlier methods to

understand benchmark scores and their sensitivity to features. Our feature ranking evolved based on the manual inputs from domain experts at COMPANYX, and also on the understanding of the nuanced state transitions of assets and their relative importance to the production fleet. As a result, we believe that performance outlier detection requires both the automation for efficiency and the manual fine-tuning to arrive at meaningful results at production scale.

Observation 3: *Features have different weights to performance outlier detection (with* `ModelID ¿ BIOS ¿ Kernel`*).* Based on the production use cases and feature ranking discussion, we found that the model architecture has the highest impact on performance scores. While this is an obvious result, the significance of it is higher than the firmware or software related features which can also sway the scores. BIOS version has the second most significant impact on CPU and memory performance and a plethora of BIOS options can affect the enumeration and configuration of the CPU. If left untracked, the lack of BIOS input could have severe performance degradation remain unnoticed. Kernel being the next impactful feature also adds value for future data collection.

Observation 4: *Periodic performance benchmarking tracks fleet regressions and can enhance fleet stability.* In a live fleet, BIOS versions and kernel versions are rolled out continuously. Our methods in this paper, as highlighted by our case studies, have proven effective in identifying regressions due to BIOS (and kernel) configuration variation. The methods presented in this paper have been proven to be useful in our offline evaluation cycle, and we will explore online integration at COMPANYXas the next step. Our results show that periodic performance benchmarking and outlier detection across BIOS release, kernel release, and hardware configuration changes can enhance overall fleet stability.

Observation 5: *Hardware performance degradation is real.* We observe that hardware performance does degrade over time, and is often unexplainable by any of the top features in our study. This leads us to conclude that repeated low performance benchmark score without any changes in top ranking features signals potential hardware issue. Hardware does gradually degrade, and silently sliding down from previously assessed high performance level. We present a couple of as-

set examples which show degradation behavior. Periodic performance benchmarking is required to catch these hardware failures in time to reduce the risk of deploying application laggard hardware in the future.

CHAPTER 8

RELATED WORK

8.1 Related Work

In this section we will discuss related works in the fields and the contribution our work brings in.

Large-scale studies: Large-scale fleet management has gaining its attention in the recent decade due to its increasing importance and there have been a lot of studies on understanding machine performance and failures in large-scale deployment. As a first step, many empirical studies are conducted on large-scale systems on performance [1, 8, 27], general failure and crash recovery analysis [7, 9, 18, 19, 53, 68, 69], and hardware errors [20, 33]. Google published traces from their large-scale production cluster called Borg [64] aiming to facilitate the scheduling pattern study [60]. Besides compute cluster, the performance and reliability of enterprise storage system at large-scale is also well analyzed at NetApp and Google [21, 42, 43, 55]. Microsoft conducted an empirical analysis focusing on hardware failures within CPU and disk subsystems [47]. Meta took a step further via analyzing, predicting, and performing root cause investigation and hardware remediation at scale [35, 36, 37]. *We continue the importance of this area of study and have the focus on leveraging the CPU and memory benchmarks for the cluster management, which we believe have not been reported at such a large scale.*

Dataset scale: In terms of the scale of the dataset, many large scale datasets of various types have been well analyzed by previous studies. Empirical analysis are carried out using incident records [73] and tickets [54, 65, 68] from the platforms that are based on over 100k nodes. Logs and error messages are also common sources for system failure studies focusing on large fleets that could involve millions of servers [26, 34, 47, 67]. *Our large-scale dataset is at the same level. We have a dataset that contains over millions of assets in total and will be made public to academic collaborators (pending legal approval) to facilitate further studies.*

Analysis techniques: For analyzing and further utilizing the dataset, there are a lot of previous works that use techniques ranging from statistical techniques such as variance, hierarchical clustering, and optimal experiment design [24, 31, 63], information theory-based tools [15, 70], to machine learning tools including OCSVM [12], transfer learning [70, 71], gradient boosted tree [32], deep learning model [11, 32], Causal Bayesian Networks [17], and negative-unlabeled learning [10], and even extend the application of NLP to data center management [4, 72]. Particularly, for anomaly detection, clustering techniques such as K-means [39], Extreme Studentized Deviate [22, 62], k-NN [40], SVDD [23], and neural network [57, 58, 61] have been explored. Some studies change the perspective of model building by leveraging user perception rather than complex system metrics to unearth anomalies. *Aiming at the same target, in our work, we show that multi-layered recommendation system (variance, divergence, outlier detection) would be beneficial, and we learn that using off-the-shelf statistical techniques as-is does not suffice the low false positive requirement for the production cluster management. Instead, a customized pipeline is crucial for denoising, given the complex nature of the dataset with a diverse feature set.*

Benchmarks: When it comes to benchmark choices, in particular, CPU and memory benchmarks, Stream [46] and CoreMark-PRO [3] have been recognizably popular for many practitioners. The former is widely used for measuring peak memory bandwidth [6, 14, 44, 45, 48, 59, 66]. For the latter, while CoreMark [16] itself is already an outstanding and widely used benchmark for the CPU pipeline [52], its next generation, the CoreMark-Pro Suite, stems from it and gained its popularity for the wider range of coverage. Its support for stress testing the entire processor makes it a benchmark suitable for various types of compute systems [49, 50]. *We believe that we are the largest dataset with respect to these two benchmarks.*

CHAPTER 9

SUMMARY

9.1 Conclusion

Stable and predictable performance of large-scale fleet is critical for providing reliable services. The performance of a server fluctuates over time and thus identifying underperforming servers is challenging as it requires minimizing the testing overhead as well as the false positive rate. Aiming to pinpoint and remove underperforming servers, FREP is a rigorous and unique customization of off-the-shelf statistical techniques and is applied offline to a dataset containing over 4 million benchmark results from over millions of production servers in COMPANYX. Results from FREP verifies the plan for its deployment in datacenters to facilitate real-time degradation identification.

REFERENCES

[1] Performance Analysis of Alibaba Large-Scale Data Center. https://www.alibabacloud.com/blog/performance-analysis-of-alibaba-large%25-scale-data-center_594676, 2019.

[2] EVT, DVT, and PVT: Product Development Stages Explained. https://www.onlogic.com/company/io-hub/evt-dvt-and-pvt-product-development-stages-explained/, 2022.

[3] CoreMark-PRO An EEMBC Benchmark Suite. https://www.eembc.org/coremark-pro/, 2023.

[4] Christophe Bertero, Matthieu Roy, Carla Sauvanaud, and Gilles Tredan. Experience Report: Log Mining Using Natural Language Processing and Application to Anomaly Detection. In *Proceedings of the IEEE International Symposium on Software Reliability (ISSRE)*, 2017.

[5] Ben W. Bolch. More on unbiased estimation of the standard deviation. *The American Statistician*, 22:27, 1968.

[6] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. Forestgomp: an efficient openmp environment for numa architectures. *International Journal of Parallel Programming*, 38:418–439, 2010.

[7] Ignacio Cano, Srinivas Aiyar, and Arvind Krishnamurthy. Charaterizing private clouds: A large-scale empirical analysis of enterprise clusters. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, 2016.

[8] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[9] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella, and Nematollah Bidokhti. How Bad Can a Bug Get? An Empirical Analysis of Software Failures in the OpenStack Cloud Computing Platform. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2019.

[10] Yi Ding, Avinash Rao, Hyebin Song, Rebecca Willett, and Henry Hoffmann. NURD: Negative-Unlabeled Learning for Online Datacenter Straggler Prediction. In *The Conference on Machine Learning and System*, 2022.

[11] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In *Proceedings of the 2017 ACM Conference on Computer and Communications Security (CCS)*, 2017.

[12] Sarah M Erfani, Sutharshan Rajasegarar, Shanika Karunasekera, and Christopher Leckie. High-dimensional and large-scale anomaly detection using a linear one-class svm with deep learning. *Pattern Recognition*, 58:121–134, 2016.

[13] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Knowledge Discovery and Data Mining*, 1996.

[14] Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009, 2009.

[15] Song Fu. Performance Metric Selection for Autonomic Anomaly Detection on Cloud Computing Systems. In *IEEE Global Communications Conference (GLOBECOM)*, 2011.

[16] Shay Gal-On and Markus Levy. Exploring CoreMark - A Benchmark Maximizing Simplicity and Efficacy. https://www.eembc.org/techlit/articles/coremark-whitepaper.pdf.

[17] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[18] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. An Empirical Study on Crash Recovery Bugs in Large-Scale Distributed Systems. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2018.

[19] Peter Garraghan, Paul Townend, and Jie Xu. An Empirical Failure-Analysis of a Large-Scale Cloud Computing Environment. In *The 15th IEEE International Symposium on High-Assurance Systems Engineering (HASE)*, 2014.

[20] Jim Gray and Catharine Van Ingen. Empirical Measurements of Disk Failure Rates and Error Rates. Microsoft Research Technical Report MSR-TR-2005-96, 2005.

[21] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[22] Jordan Hochenbaum, Owen S Vallis, and Arun Kejariwal. Automatic anomaly detection in the cloud via statistical learning. *arXiv preprint arXiv:1704.07706*, 2017.

[23] Chengqiang Huang, Geyong Min, Yulei Wu, Yiming Ying, Ke Pei, and Zuochang Xiang. Time Series Anomaly Detection for Trustworthy Services in Cloud Computing Systems. In *IEEE Transactions on Big Data*, 2017.

[24] Mihailo Isakov, Eliakin del Rosario, Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert B. Ross, and Michel A. Kinsy. HPC I/O Throughput Bottleneck Analysis with Explainable Local Models. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[25] Paul Jaccard. Comparative study of floral distribution in a portion of the alps and jura. *Bulletin of the Natural History Society of Vaud*, 37:547–579, 1901.

[26] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are Disks the Dominant Contributor for Storage Failures? A Comprehensive Study of Storage Subsystem Failure Characteristics. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST)*, 2008.

[27] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[28] Leonid Kantorovich. Mathematical methods of organizing and planning production. *Management Science*, 6(4):366–422, 1960.

[29] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. Estimating mutual information. *Phys. Rev. E*, 69:066138, Jun 2004.

[30] Solomon Kullback and Richard Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79 – 86, 1951.

[31] Jaewon Lee, Changkyu Kim, Kun Lin, Liqun Cheng, Rama Govindaraju, and Jangwoo Kim. WSMeter: A Performance Evaluation Methodology for Google's Production Warehouse-Scale Computers. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[32] Sebastien Levy, Randolph Yao, Youjiang Wu, Yingnong Dang, Peng Huang, Zheng Mu, Pu Zhao, Tarun Ramani, Naga Govindaraju, Xukun Li, Qingwei Lin, Gil Lapid Shafriri, and Murali Chintalapati. Predictive and Adaptive Failure Mitigation to Avert Production Cloud VM Interruptions. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[33] Xin Li, Michael C. Huang, and Kai Shen. An Empirical Study of Memory Hardware Errors in A Server Farm. In *The 3rd Workshop on Hot Topics in System Dependability (HotDep)*, 2007.

[34] Yinglung Liang, Yanyong Zhang, Morris Jette, Anand Sivasubramaniam, and Ramendra Sahoo. BlueGene/L Failure Analysis and Prediction Models. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2006.

[35] Fan Fred Lin, Keyur Muzumdar, Nikolay Pavlovich Laptev, Mihai-Valentin Curelea, Seunghak Lee, and Sriram Sankar. Fast Dimensional Analysis for Root Cause Investigation in a Large-Scale Service Environment. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, 2020.

[36] Fred Lin, Matt Beadon, Harish Dattatraya Dixit, Gautham Vunnam, Amol Desai, and Sriram Sankar. Hardware Remediation At Scale. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2018.

[37] Fred Lin, Antonio Davoli, Imran Akbar, Sukumar Kalmanje, Leandro Silva, John Stamford, Yanai Golany, Jim Piazza, and Sriram Sankar. Predicting Remediations for Hardware Failures in Large-Scale Datacenters. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2020.

[38] Jianhua Lin. Divergence measures based on the shannon entropy. *IEEE Transactions on Information Theory*, 37(1):145–151, 1991.

[39] Jieyu Lin, Qi Zhang, Hadi Bannazadeh, and Alberto Leon-Garcia. Automated Anomaly Detection and Root Cause Analysis in Virtualized Cloud Infrastructures. In *IEEE/IFIP Network Operations and Management Symposium*, 2016.

[40] Zhaoli Liu, Tao Qin, Xiaohong Guan, Hezhi Jiang, and Chenxu Wang. An Integrated Method for Anomaly Detection From Massive System Logs. In *IEEE Access*, 2018.

[41] Stuart P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[42] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. A Study of SSD Reliability in Large Scale Enterprise Storage Deployments. In *Proceedings of the 18th USENIX Symposium on File and Storage Technologies (FAST)*, 2020.

[43] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. Reliability of SSDs in Enterprise Storage Systems: A Large-Scale Field Study. In *ACM Transactions on Storage (TOS)*, 2021.

[44] Aniruddha Marathe, Rachel Harris, David K. Lowenthal, Bronis R. de Supinski, Barry Rountree, Martin Schulz, and Xin Yuan. A Comparative Study of High-Performance Computing on the Cloud. In *Proceedings of the 22nd IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2013.

[45] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming Performance Variability. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[46] John D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. https://web.archive.org/web/20200205055303/http://www.cs.virginia.edu/~mccalpin/papers/bandwidth/sigmetrics.html, 2006.

[47] Edmund B. Nightingale, John R Douceur, and Vince Orgovan. Cycles, Cells and Platters: An empirical analysis of hardware failures on a million consumer PCs. In *Proceedings of the 2011 EuroSys Conference (EuroSys)*, 2011.

[48] Nick L. Petroni, Jr. Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th conference on USENIX Security Symposium*, 2004.

[49] Ivan Porres, Tanwir Ahmad, Hergys Rexha, Sébastien Lafond, and Dragos Truscan. Automatic exploratory performance testing using a discriminator neural network. In *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2020.

[50] Ivan Porres, Hergys Rexha, and Sébastien Lafond. Online GANs for Automatic Performance Testing. In *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2021.

[51] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. Finding a "Kneedle" in a Haystack: Detecting Knee Points in System Behavior. In *2011 31st International Conference on Distributed Computing Systems Workshops*, pages 166–171, 2011.

[52] Romain Saussard, Boubker Bouzid, Marius Vasiliu, and Roger Reynaud. A Robust Methodology for Performance Analysis on Hybrid Embedded Multicore Architectures. In *Proceedings of the 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, 2016.

[53] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2006.

[54] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST)*, 2007.

[55] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[56] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[57] John Sipple. Interpretable, Multidimensional, Multimodal Anomaly Detection with Negative Sampling for Detection of Device Failure. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*, 2020.

[58] Ya Su, Youjian Zhao, Chenhao Niu, Rong Liu, Wei Sun, and Dan Pei. Robust Anomaly Detection for Multivariate Time Series through Stochastic Recurrent Neural Network. In *Proceedings of the 25nd SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2019.

[59] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[60] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the 2020 EuroSys Conference (EuroSys)*, 2020.

[61] Mineto Tsukada, Masaaki Kondo, and Hiroki Matsutani. A Neural Network-Based On-Device Learning Anomaly Detector for Edge Devices. In *IEEE Transactions on Computers (TC)*, 2020.

[62] Owen Vallis, Jordan Hochenbaum, and Arun Kejariwal. A Novel Technique for Long-Term Anomaly Detection in the Cloud. In *The 6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2014.

[63] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[64] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 2015 EuroSys Conference (EuroSys)*, 2015.

[65] Kashi Vishwanath and Nachi Nagappan. Characterizing Cloud Computing Hardware Reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[66] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. In *Communications of the ACM (CACM)*, 2009.

[67] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting Large-Scale System Problem Detection by Mining Console Logs. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[68] Praveen Yalagandula, Suman Nath, Haifeng Yu, Phillip B. Gibbons, and Srinivasan Seshan. Beyond Availability: Towards a Deeper Understanding of Machine Failure Characteristics in Large Distributed Systems. In *Proceedings of the Workshop on Real, Large Distributed Systems (WORLDS)*, 2004.

[69] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[70] Ji Zhang, Ke Zhou, Ping Huang, Xubin He, Ming Xie, Bin Cheng, Yongguang Ji, and Yinhu Wang. Minority Disk Failure Prediction Based on Transfer Learning in Large Data Centers of Heterogeneous Disk Systems. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2020.

[71] Xu Zhang, Qingwei Lin, Yong Xu, Si Qin, Hongyu Zhang, Bo Qiao, Yingnong Dang, Xinsheng Yang, Qian Cheng, Murali Chintalapati, Youjiang Wu, Ken Hsieh, Kaixin Sui, Xin Meng, Yaohai Xu, Wenchi Zhang, Furao Shen, and Dongmei Zhang. Cross-dataset Time Series Anomaly Detection for Cloud Systems. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[72] Giulio Zhou and Martin Maas. Learning on Distributed Traces for Data Center Storage Systems. In *The Conference on Machine Learning and System*, 2021.

[73] Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Haibo Lin, Haoxiang Lin, and Tingting Qin. An Empirical Study on Quality Issues of Production Big Data Platform. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.