International Journal of Emerging Technology in Computer Science & Electronics (IJETCSE)
ISSN: 0976-1353 Volume 9 Issue 3 –SEPTEMBER 2014.

Implementation of Cloud Diag by Combining Statistical Techniques and Fast Matrix Recovery Algorithm to Identify the Fine Grained Unsupervised Performance Problems for Producing Efficiently in Cloud Computing Systems

Muntha V Charishma*1, A.Ravi#2 and D. Venkata Subbaiah3

*Student, Priyadarshini College of Engineering and Technology, Nellore, India

#Assistant Professor (CSE), Priyadarshini College of Engineering and Technology, Nellore, India

3HOD (CSE), Priyadarshini College of Engineering and Technology, Nellore, India

1charishma.583@gmail.com

2510ravi@gmail.com

3dvsmtech_2005@yahoo.com

***Abstract*— Mixing a mathematical strategy and fast matrix restoration criteria, Cloud Diag can effectively determine fine-grained causes of the efficiency issues, which does not require any domain-specific knowledge to the focus on system. CloudDiag has been used in a realistic manufacturing reasoning handling systems to identify efficiency issues. We illustrate the potency of CloudDiag in three real-world situation research. Building on end-to-end request-flow searching within and across elements, methods are described for determining and position changes in the flow and/or moment of demand handling. The execution of these methods in a device called Spectroscope is analyzed. Six situation researches are provided of using Spectroscope to identify efficiency changes in a allocated storage space service due to rule changes, configuration modifications, and element degradations, indicating the value and efficiency of evaluating demand flows. Initial encounters of using Spectroscope to identify efficiency changes within choose Google services are also performed.**

I. INTRODUCTION

CloudDiag regularly gathers the end-to-end searching information (In particular, performance time of method invocations) from each physical node in the reasoning. It then utilizes personalized Map-Reduce criteria to proactively evaluate the searching information. Specifically, it puts together the searching information of each user demand, and categorizes the searching information into different groups according to call plants of the requests. When the reasoning program is struggling efficiency deterioration (e.g., frequent reaction time of customer demands is bigger than a threshold), a reasoning owner can accessibility CloudDiag with its web connections to execute a efficiency analysis. With the demand searching information, CloudDiag will execute fast personalized matrix restoration criteria to immediately recognize the method invocations (together with the replications. they locate) which play a role the most to the efficiency abnormality. The whole process needs no domain-specific information to the focus on service. CloudDiag has been efficiently released in identifying efficiency issues for the growth reasoning techniques in Alibaba Cloud Processing. We review three case researches in our real-world efficiency analysis encounters to show the potency of CloudDiag in assisting the providers localize the main causes of efficiency issues. This document produces a new strategy for the suite: evaluating demand flows between two accomplishments to recognize why efficiency has modified between them. Such evaluation allows one performance to provide as a design of appropriate performance; featuring key variations from this design and knowing their efficiency expenses allows for simpler analysis than when only a single performance is used. Though acquiring an performance of appropriate efficiency may not be possible in all cases— e.g., when a designer wants to understand why efficiency has always been poor—there are many situations for which request-flow evaluation is useful. For example, it can help identify efficiency changes as a result of modifications made during software growth (e.g., during frequent regression testing) or from improvements to elements of a implemented program.

II. DIAGNOSING ANOMALIES WITHOUT DOMAIN KNOWLEDGE

In this section, CloudDiag first employs a statistical technique to detect anomalous category that contain latency-anomalous requirements. Then, from anomalous categories, a fast matrix recovery algorithm, namely, RPCA is adopted to identify the inconsistent methods and instances. Details are as follows:

Identifying Anomalous Categories:

We cannot rely only on the response latency of a request to confirm whether a request is inconsistent. Long response latency does not indicate a failure. For example, the response latency of one request reading a file from hard disk is several times longer than that of another reading a file from cache. Ordinary and anomalous replicas exist simultaneously when concert problems happen. For a component, the same service requests may pass through normal instances as well as abnormal instances. The response latency of a request will be influenced by anomalous instances that it passes all the way through. Normal and anomalous requests may share the identical call tree and be grouped into one category; thus, the latency distribution of requests within the same category could be utilized to detect whether it contains latency-anomalous requirements or not. Requests within one category have the same call tree; hence, the response latencies should be close to each other. A kind is considered to be normal if the latencies of requests within the category are clustered in a specific range; on the contrary, a category is considered to be anomalous if the latencies are over dispersed. In this regards, we choose the coefficient of variation (CV) [15] to measure the distribution of a set of data. Let α be the threshold.

Identifying Anomalous Methods:

In an anomalous category of requests, our aim now is to isolate the anomalous method invocations that are responsive for the presentation anomaly of the requests. For such a category of requests, we can create a $m \times n$ matrix M, where n is the number of the invoked methods in the corresponding call tree and m is the number of the requests that bear the same call tree. M_{ij} denotes the full matrix M as: $M = L + E$, where L is a low-rank matrix with noncorrupted columns and E is a sparse matrix with a few nonzero corrupted columns. The matrix M is the input of RPCA. Therefore, the problem of identifying anomalous methods in a category is transferred into the process of recovering a matrix with unknown corrupted latency columns. After obtaining the noncorrupted matrix L and error matrix E, we can identify the corrupted columns (i.e., the anomalous methods) from E. The anomalous methods refer to those columns that are utmost from the true column space. For the ith column in original matrix M and

noncorrupted matrix L, the conservatory of deviation can be measured as:

$$\beta = \cos\theta = \frac{\|M(i) \cdot L(i)\|_1}{\|M(i)\|_2 \|L(i)\|_2},$$

Where θ represents the angle between column $M(i)$ and column $L(i)$. The larger the angle is, the more divergence the Column $L(i)$ is away from the true space. A method is defined to be anomalous if β is smaller than a given threshold. For each anomalous method (i.e., the corrupted column), anomalous replicas are located by checking the entries of the ruined column in Matrix E. With the row and column indices of the corrupted entries, we can get the physical addresses of anomalous replicas from physical paths. Since the same method (running on the same component replica) may be identified to be anomalous in different categories, we calculate the times that it is identified to be anomalous. The larger the number of times is, the more suspicious the method is. CloudDiag can then rank the methods in descending order of the number of times that they are identified to be anomalous, which can direct the operators to localize the primary cause of performance anomaly.

III. PROPOSED MODEL SPECTROSCOPE

To illustrate the utility of comparing request flows, this technique was implemented in a tool called Spectroscope and worn to detect performance problems seen in Ursa Minor and in certain Google services. This section provides an overview of Spectroscope, and the next describes its algorithms.

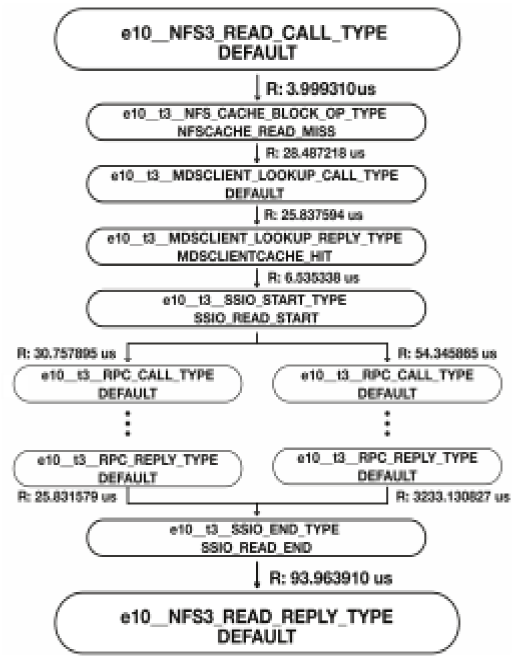

Figure 1-Example request-flow graph.

The graph shows a striped READ in the Ursa Minor distributed storage system. Nodes represent trace points and edges are labeled with the time between successive events. Parallel substructures show concurrent threads of activity. Node labels are constructed by concatenating the machine name (e.g., e10), component name (e.g., NFS3), trace-point name (e.g., READ CALL TYPE), and an optional semantic label (e.g., NFSCACHE READ MISS). Due to space constraints, trace points executed on other components as a result of the NFS server's RPC calls are not shown.

Categorizing request flows:

Even small distributed systems can service hundreds to thousands of requests per second, so comparing all of them individually is not feasible. Instead, exploiting a general expectation that requests that take the same path should incur similar costs, Spectroscope groups identically-structured requests into unique categories and uses them as the basic unit for comparing request flows. For example, requests whose structures are identical because they hit in a NFS server's data and metadata cache will be grouped into the same category, whereas requests that miss in both will be grouped in a dissimilar one. Two requests are deemed structurally identical if their string representations, as determined by a depth first traversal, are identical. For requests with parallel substructures, Spectroscope computes all possible string representations when formative the category in which to bin them. The exponential cost is mitigated by imposing an order on parallel substructures (i.e., by always traversing them in alphabetical order, as determined by their root node names) and by the fact that parallelism is limited in most request flows we have observed. For each category, Spectroscope identifies aggregate statistics, including request count, standard response time, and variance. To identify where time is spent, it also computes average edge latencies and corresponding variances. Spectroscope displays categories in either a graph view, with statistical information overlaid, or within train-schedule visualizations (also known as swim lanes), which more directly show the constituent requests' pattern of activity. Spectroscope uses selection criteria to limit the number of categories developers must examine. For example, when comparing request flows, statistical tests and a ranking scheme are used. The number of categories could be further reduced by using unsupervised clustering algorithms, such as those used in Magpie, to bin similar but not necessarily identical requests into the same category. Initial versions of Spectroscope used off-the-shelf clustering algorithms, but we found the groups they created too coarse-grained and unpredictable. Often, they would group mutations and precursors within the same category, masking their existence. For clustering algorithms to be useful, improvements such as distance metrics that better align with developers' notions of request similarity are needed. Without them, use of clustering algorithms will result in categories composed of seemingly dissimilar requests.

Comparing request flows:

Performance changes can result from a variety of factors, such as internal changes to the system that result in performance regressions, unintended side effects of changes to configuration files or environmental issues. Spectroscope helps diagnose these problems by comparing request flows and identifying the key resulting mutations. When comparing request flows, Spectroscope takes as input request-flow graphs from two periods of activity, which we refer to as a non-problem period and a problem period. It creates categories composed of requests from both periods and uses statistical tests and heuristics to identify which contain structural mutations, response time mutations, or precursors. Categories containing mutations are presented to the developer in a list ranked by expected contribution to the performance change. Note that the periods do not need to be aligned exactly with the performance change (e.g., at Google we often chose day-long periods based on historic average latencies). Visualizations of categories that contain mutations are similar to those described previously, except per period statistical information is shown. The root cause of response-time mutations is localized by showing the edges responsible for the mutation in red. The root cause of structural mutations is localized by providing a ranked list of the candidate precursors, so that the developer can determine how they differ.

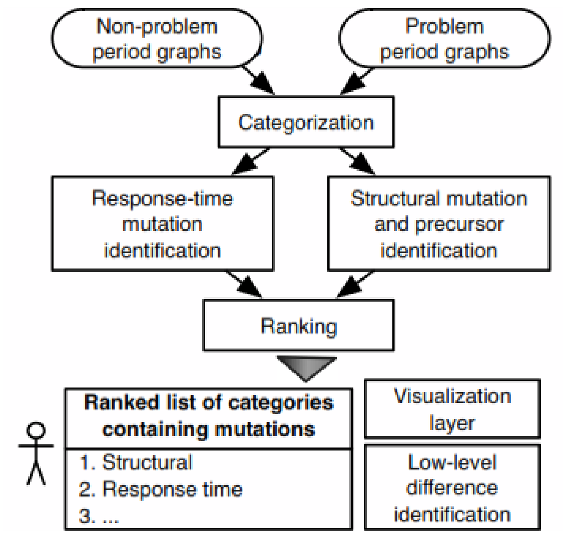
Non-problem period graphs
Problem period graphs
Categorization
Response-time mutation identification
Structural mutation and precursor identification
Ranking
Ranked list of categories containing mutations
1. Structural
2. Response time
3. ...
Visualization layer
Low-level difference identification

Figure 2-Spectroscope's workflow for comparing request flows.

First, Spectroscope groups requests from both periods into categories. Second, it identifies which categories contain mutations or precursors. Third, it ranks mutation categories according to their expected contribution to the performance change. Developers are presented this ranked list. Visualizations of mutations and their precursors can be shown. Also, low-level differences can be identified for them.

IV. URSA MINOR

Ursa Minor separates metadata services from data services, such that clients can access data on storage nodes without moving it all through metadata servers. An Ursa Minor instance (called a "constellation") consists of potentially many NFS servers (for unmodified clients), storage nodes (SNs), metadata servers (MDSs), and end-to-end-trace servers. To access data, clients must first send a request to a metadata server asking for the appropriate permissions and locations of the data on the storage nodes. consumers can then access the storage nodes directly. Ursa Minor has been in active advance since 2004 and comprises about 230,000 lines of code. More than 20 graduate students and staff have contributed to it over its lifetime. More details about its implementation can be found in Abd-El-Malek et al. The components of Ursa Minor are usually run on separate machines within a datacenter. Though Ursa Minor supports a subjective number of components, the experiments and case studies detailed in this paper use a simple five-machine configuration: one NFS server, one metadata server, one trace server, and two storage nodes. One storage node stores data, while the other stores metadata. Not coincidentally, this is the configuration used in the nightly regression tests that uncovered many of the problems described in the case studies.

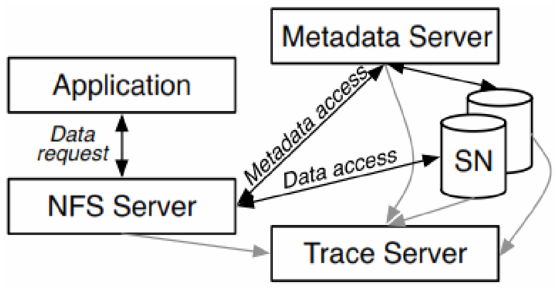

Figure 3-Ursa Minor Architecture:

Ursa Minor can be deployed in many configurations, with an arbitrary number of NFS servers, metadata servers, storage nodes (SNs), and trace servers. Here, a simple five-component configuration is shown.

V. DAPPER & GOOGLE SERVICES

The Google services for which Spectroscope was applied were instrumented using Dapper, which automatically embeds trace points in Google's RPC structure. Like Stardust, Dapper employs request sampling, but uses a sampling rate of less than 0.1%. Spectroscope was implemented as an extension to Dapper's aggregation pipeline, which groups individual requests into categories and was originally written to support Dapper's pre-existing analysis tools. Categories created by the aggregation pipeline only show compressed call graphs with identical children and siblings merged together.

VI. MDS CONFIGURATION CHANGE

After a particular large code check-in, performance of `postmark-large` decayed significantly, from 46tps to 28tps. To diagnose this problem, we used Spectroscope to compare request flows between two runs of `postmark-large`, one from before the check-in and one from after. The results showed many categories that contained structural mutations. Comparing them to their most-likely precursor categories revealed that the storage node utilized by the metadata server had misused Before the check-in, the metadata server wrote metadata only to its dedicated storage node. After the check-in, it issued most writes to the data storage node instead. We also used Spectroscope to identify the low-level parameter differences between a few structural-mutation categories and their corresponding precursor categories. The regression tree found differences in elements of the data distribution scheme (e.g., type of fault tolerance used).

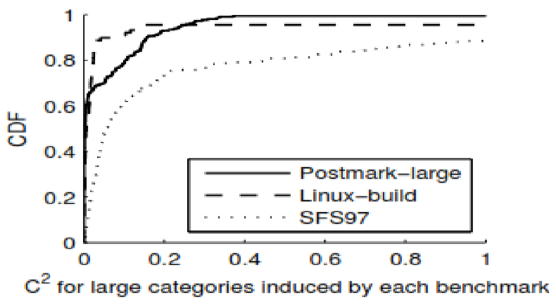

Figure 4- CDF of C^2 for large categories induced by three benchmarks run on Ursa Minor.

At least 88% of the categories induced by each benchmark exhibit low variance (C^2<1). The results for linux-build and SFS are more heavy-tailed than postmark-large, partly due to extra lock contention in the metadata server.

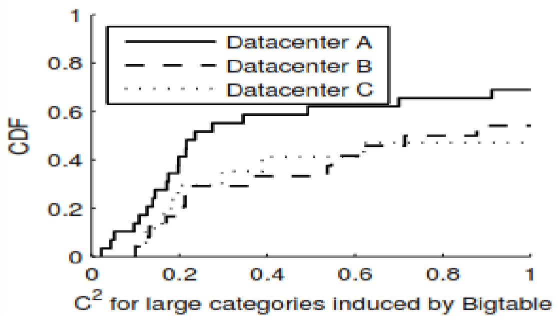

Figure 5: CDF of C^2 for large categories induced by Big table instances in three Google datacenters.

Dapper's instrumentation of big table is sparse; so many paths cannot be disambiguated and have been merged together in the observed categories, resulting in a higher C^2 than otherwise expected. Even so, 47–69% of categories exhibit low variance.

Slowdown due to code changes

This synthetic problem was injected into Ursa Minor to show how request-flow comparison can be used to diagnose slowdowns due to feature additions or regressions and to assess Spectroscope's feeling to changes in response time. Spectroscope was used to compare request flows between two runs of SFS97. Problem period runs included a spin loop injected into the storage nodes' WRITE code path. Any WRITE request that accessed a storage node incurred this extra delay, which manifested in edges of the form $\star \rightarrow$ STORAGE NODE RPC REPLY. Normally, these edges exhibit a latency of 100µs. For the latter two cases, Spectroscope was able to identify the resulting response-time mutations and localize them to the affected edges. Of the categories identified, only 6–7% is false positives and 100% of the 10 highest-ranked ones are relevant. The coverage is 92% and 93%. Variance in response times and the edge latencies in which the delay manifests prevent Spectroscope from properly identifying the affected categories for the 100µs case. It identifies 11 categories that contain requests that traverse the affected edges multiple times as containing response-time mutations, but is unable to assign those edges as the ones responsible for the slowdown.

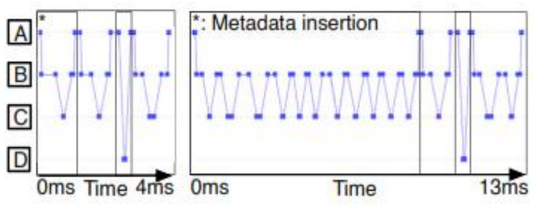

Figure 6: Visualization of create behaviour.

Two train schedule visualizations are shown, the first one a fast early create during postmark-large and the other a slower create issued later in the benchmark. Messages are exchanged between the NFS Server (A), Metadata Server (B), Metadata Storage Node (C), and Data Storage Node (D). The first phase of the create procedure is metadata insertion, which is shown to be responsible for the majority of the delay.

VII. EXPERIENCES AT GOOGLE

This section describes preliminary experiences using request-flow comparison, as implemented in Spectroscope, to diagnose performance problems within select Google services.

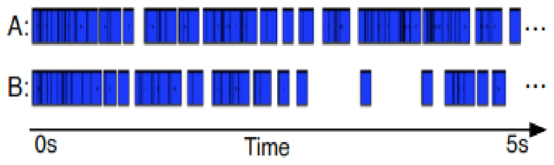

Figure 7: Timeline of inter-arrival times of requests at the NFS Server.

A 5s sample of requests, where each rectangle represents the process time of a request, reveals long periods of inactivity due to not have requests from the client during spiked copy times (B) compared to periods of normal activity (A).

VIII. CONCLUSION

Evaluating demand flows, as taken by end-to-end records, is a highly effective new technique for identifying performance changes between two time times or system editions. Spectroscope's methods for this comparison allow it to perfectly recognize and position strains and recognize their precursors, concentrating attention on the most important variations. Encounters with Spectroscope confirm its effectiveness and effectiveness.

IX. REFERENCES

[1] M. Abd-El-Malek, et al. Ursa Minor: versatile cluster-based storage. Conference on File and Storage Technologies. USENIX Association, 2005. 2, 6
[2] M. K. Aguilera, et al. Performance debugging for distributed systems of black boxes. ACM Symposium on Operating System Principles. ACM, 2003. Software Engineers,December 2010.
[3] P. Barham, et al. Using Magpie for request extraction and workload modelling. Symposium on Operating Systems Design and Implementation. USENIX Association, 2004. 1, 2, 3, 13
[4] C. M. Bishop. Pattern recognition and machine learning, first edition. Springer Science + Business Media, LLC, 2006.
[5] B. M. Cantrill, et al. Dynamic instrumentation of production systems. USENIX Annual Technical Conference. USENIX Association, 2004.
[6] A. Chanda, et al. Whodunit: Transactional profiling for multi-tier applications. EuroSys. ACM, 2007. 1, 2
[7] Chang, et al. Bigtable: a distributed storage system for structured data. Symposium on Operating Systems Design and Implementation. USENIX Association, 2006.
[8] M. Y. Chen, et al. Path-based failure and evolution management. Symposium on Networked Systems Design and Implementation. USENIX Association, 2004. 1, 2, 13
[9] R. Fonseca, et al. Experiences with tracing causality in networked services. Internet Network Management Conference on Research on Enterprise Networking. USENIX Association, 2010. 2, 13
[10] R. Fonseca, et al. X-Trace: a pervasive network tracing framework. C___ _ • _ ýfltworked Systems Design and Implementation. USENIX Association, 2007. 1, 2
[11] http://www.gnu.org/software/gdb/. 1
[12] S. Ghemawat, et al. The Google file system. ACM Symposium on Operating System Principles. ACM, 2003.
[13] S. L. Graham, et al. gprof: a call graph execution profiler. ACM SIGPLAN Symposium on Compiler Construction. Published as SIGPLAN Notices, 17(6):120–126, June 1982.
[14] Graphviz. http://www.graphviz.org. 6
[15] J. Heer, et al. Prefuse: a toolkit for interactive information visualization. Conference on Human Factors in Computing Systems. ACM, 2005.6
[16] J. Hendricks, et al. Improving small file performance in objectbased storage. Technical report CMU-PDL-06-104. Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, May 2006. 10
[17] M. P. Kasick, et al. Black-box problem diagnosis in parallel file systems. Conference on File and Storage Technologies. USENIX Association, 2010. 13
[18] J. Katcher. PostMark: a new file system benchmark. Technical report TR3022. Network Appliance, October 1997. 7
[19] F. J. Massey, Jr. The Kolmogorov-Smirnov test for goodness offit. Journal of the American Statistical Association, 46(253):66–78, 1951. 4
[20] J. C. Mogul. Emergent (Mis)behavior vs. Complex Software Systems. EuroSys. ACM, 2006.
[21] T. Moseley, et al. OptiScope: performance accountability for optimizing compilers. International Symposium on Code Generation and Optimization. IEEE/ACM, 2009. 13
[22] W. Norcott and D. Capps. IoZone filesystem benchmark program, 2002.http://www.iozone.org. 7
[23] X. Pan, et al. Ganesha: black-box fault diagnosis for MapReduce systems. Hot Metrics. ACM, 2009. 13
[24] J. R. Quinlan. Bagging, boosting and C4.5. 13th National Conference on Artificial Intelligence. AAAI Press, 1996. 6
[25] P. Reynolds, et al. Pip: Detecting the unexpected in distributed systems. Symposium on Networked Systems Design and Implementation. USENIX Association, 2006. 1, 13
[26] P. Reynolds, et al. WAP5: Black-box Performance Debugging for Wide-Area Systems. International World Wide Web Conference. ACM Press, 2006. 13
[27] R. R. Sambasivan, et al. Diagnosing performance problems by visualizing and comparing system behaviours. Technical report 10–103. Carnegie Mellon University, February 2010. 2
[28] R. R. Sambasivan, et al. Categorizing and differencing system behavioursb _¨qx@fl_ Ã __ _ s_ýfl•_ ýfl autonomic computing (HotAC). USENIX Association, 2007. 3, 13
[29] SPEC SFS97 (2.0). http://www.spec.org/sfs97. 2, 7
[30] B. H. Sigelman, et al. Dapper, a large-scale distributed systems tracing infrastructure. Technical report dapper 2010-1. Google, April 2010. 1, 2, 13
[31] E. Thereska, et al. Informed data distribution selection in a selfpredicting storage system. International conference on autonomic computing. IEEE, 2006. 13
[32] E. Thereska and G. R. Ganger. IRONModel: robust performance models in the wild. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. ACM, 2008.1,13
[33] E. Thereska, et al. Stardust: Tracking activity in a distributed storage system. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. ACM, 2006. 1, 2, 13
[34] A. Traeger, et al. DARC: Dynamic analysis of root causes of latency distributions. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. ACM, 2008. 13
[35] J. Tucek, et al. Triage: diagnosing production run failures at the user's site. ACM Symposium on Operating System Principles, 2007. 13
[36] E. R. Tufte. The visual display of quantitative information. Graphics Press_ øn€b@ ire, Connecticut, 1983. 3