Automated diagnosis without predictability is a recipe for failure

Raja R. Sambasivan & Gregory R. Ganger
Carnegie Mellon University

Abstract

Automated management is critical to the success of cloud computing, given its scale and complexity. But, most systems do not satisfy one of the key properties required for automation: predictability, which in turn relies upon low variance. Most automation tools are not effective when variance is consistently high. Using automated performance diagnosis as a concrete example, this position paper argues that for automation to become a reality, system builders must treat variance as an important metric and make conscious decisions about where to reduce it. To help with this task, we describe a framework for reasoning about sources of variance in distributed systems and describe an example tool for helping identify them.

1 Introduction

Many in the distributed systems community [12, 20] recognize the need for automated management in cloud infrastructures, such as large distributed systems and datacenters. They predict that the rapidly increasing scale and complexity of these systems will soon exceed the limits of human capability. Automation is the only recourse, lest they become completely unmanageable. In response to this call to arms, many research papers have been published on tools for automating various tasks, especially performance diagnosis [4, 5, 9, 17–19, 23, 24, 27, 28, 31]. Most focus on layering automation on top of existing systems, without regard to whether they exhibit a key property needed for automation—predictability. Most systems do not, especially the most complex ones that need automation the most. This limits the utility of automation tools and the scope of tasks that can be automated.

Our experiences using an automated performance diagnosis tool, Spectroscope [27], to diagnose problems in distributed storage systems, such as Ursa Minor [1] and Bigtable [8], bear out the inadequacy of the predictability assumption. Though Spectroscope has proven useful, it has been unable to reach its full potential due to high variance in performance resulting from poorly structured code, high resource contention, and hardware issues.

The predictability of a distributed system is affected by variance in the metrics used to make determinations about it. As variance increases, predictability decreases as it becomes harder for automation tools to make confident, or worse, correct determinations. Though variance cannot be eliminated completely due to fundamental nondeterminism (e.g., actions of a remote system and bit errors), it can be reduced, improving predictability.

To aid automation, system builders could be encouraged to always minimize variance in key metrics. This policy dovetails nicely with areas in which predictability is paramount. For example, in the early 1990s, the US Postal Service slowed down mail delivery because they decided consistency of delivery times was more important than raw speed. When asked why this tradeoff was made, the postmaster general responded: "I began to hear complaints from mailers and customers about inconsistent first-class mail delivery. . . . We learned how important consistent, reliable delivery is to our customers" [7]. In scientific computing, inter-node variance can drastically limit performance due to frequent synchronization barriers. In real-time systems, it is more important for programs to meet each deadline than run faster on average. Google has recently identified low response-time variance as crucial to achieving high performance in warehouse-scale computing [6].

Of course, in many cases, variance is a side effect of desirable performance enhancements. Caches, a mainstay of most distributed systems, intentionally trade variance for performance. Many scheduling algorithms do the same. Also, reducing variance blindly may lead to synchronized "bad states," which may result in failures or drastic performance problems. For example, Patil et al. describe an emergent property in which load-balancing in GIGA+, a distributed directory service, leads to a large performance dropoff as compute-intensive hash bucket splits on various nodes naturally become synchronized [25].

Some variance is intrinsic to distributed systems and cannot be reduced without wholesale architectural changes. For example, identical components, such as disks from the same vendor, can differ significantly in performance due to fault-masking techniques and manufactur-

ing effects [3,21]. Also, it may be difficult to design complex systems to exhibit low variance because it is hard to predict their precise operating conditions [15, 22].

In practice, there is no easy answer in deciding how to address variance to aid automation. There is, however, a wrong answer—ignoring it, as is too often being done today. Instead, for the highly touted goal of automation to become a reality, system builders must treat variance as a *first-class metric*. They should strive to localize the sources of variance in their systems and make conscious decisions about which ones should be reduced. Such explicit decisions about variance properties will result in more robust systems and will vastly improve the utility of automation tools that rely on low variance to work.

The rest of this paper is organized as follows. Section 2 discusses how automated performance diagnosis tools are affected by high variance. Section 3 identifies three types of variance that can be found in distributed systems and what should be done about them. Section 4 proposes a mechanism system builders can use to identify the sources of variance in their systems. Section 5 discusses open questions and Section 6 concludes.

2 Diagnosis tools & variance

Tools that automate aspects of performance diagnosis each assume a unique model of system behaviour and use deviations from it to predict diagnoses. Most do not identify the root cause directly, but rather automatically localize the source of the problem from any of the the numerous components in the system to just the specific components or functions responsible. Different tools exhibit different failure modes when variance is high, depending on the underlying techniques they use.

Tools that rely on thresholds make predictions when important metrics chosen by experts exceed pre-determined values. Their failure mode is the most unpredictable, as they will return more false positives (inaccurate diagnoses) or false negatives (diagnoses not made when they should have been), depending on the value of the threshold. A low threshold will result in more false positives, whereas increasing it to accommodate the high variance will mask problems, resulting in more false negatives. False positives perhaps represent the worst failure mode, due to the amount of developer effort wasted [2].

To avoid costly false positives, some tools use statistical techniques to avoid predicting when the expected false positive rate exceeds a pre-set one (e.g., 5%). The cost of high variance for them is an increase in false negatives. Some statistical tools use adaptive techniques to increase their confidence before making predictions—e.g., by collecting more data samples. The cost of high variance for them is increased storage/processing cost and an increase in time required before predictions can be made.

Many tools use machine learning to automatically learn the model (e.g., metrics and values) that best predicts performance. The false positive rate and false negative rate are controlled by selecting the model that best trades generality (which usually results in more false negatives) with specificity (which results in more false positives)[1].

2.1 How real tools are affected by variance

Real tools use a combination of the techniques described above to make predictions. This section lists four such tools and how they are affected by variance. Table 1 provides a summary and lists additional tools.

Magpie [5]: This tool uses an unsupervised machine learning algorithm (clustering) to identify anomalous requests in a distributed system. Requests are grouped together based on similarity in request structure, performance metrics, and resource usage. It expects that most requests will fall into one of several "main" clusters of behaviour, so it identifies small ones as anomalies. A threshold is used to decide whether to place a request in the cluster deemed most similar to it, or whether to create a new one. High variance in the values of the features used and use of a low threshold will yield many small clusters, resulting in an increase in false positives. Increasing the threshold will result in more false negatives.

Spectroscope [27]: This tool uses a combination of statistical techniques and thresholds to identify the changes in request processing most responsible for an observed performance change. It relies on the expectation that requests that take the same path through a distributed system's components should incur similar performance costs and that the request topologies observed (e.g., components visited and functions executed by individual requests) should be similar across executions of the same workload. High variance in these metrics will increase the number of false positives and false negatives. Experiments run on Bigtable [8] in three different Google datacenters show that 47–69% of all unique paths observed satisfy the similar paths expectation, leaving much room for improvement [27]. Those paths that do not satisfy it suffer from a lack of enough instrumentation to tease out truly unique paths and high contention with co-located processes.

Peer comparison [18, 19]: These diagnosis tools are intended to be used on tightly coupled distributed systems, such as Hadoop and PVFS. They rely on the expectation that every machine in a given cluster should exhibit the same behaviour. As such, they indict a machine as exhibiting a problem if its performance metrics differ significantly from others. Thresholds are used to determine the degree of difference tolerated. High variance in metric distributions between machines will result in more false positives, or false negatives, depending on the threshold chosen. Re-

[1] This is known as the bias-variance tradeoff.

Tool	FPs / FNs	Tool	FPs / FNs
Magpie [5]	⇑ / ⇑	DARC [31]	- / ⇑
Spectroscope [27]	⇑ / ⇑	Distalyzer [23]	- / ⇑
Peer comp. [19]	⇑ / ⇑	Pinpoint [9]	- / ⇑
NetMedic [17]	- / ⇑	Shen [28]	- / ⇑
Oliner [24]	- / ⇑	Sherlock [4]	- / ⇑

Table 1: How the predictions made by automated performance diagnosis tools are affected by high variance. As a result of high variance, diagnosis tools will yield more false positives (FPs), false negatives (FNs), or both, depending on the techniques they use to make predictions. Note that the choice of failure mode attributed to each tool was made conservatively.

cent trends that result in increased performance variance from same-batch, same-model devices [21] negatively affect this peer-comparison expectation.

Tool described by Oliner [24]: This tool identifies correlations in anomalies across components of a distributed system. To do so, it first calculates an *anomaly score* for discrete time intervals by comparing the distribution of some signal—e.g., average latency—during the interval to the overall distribution. The strength of this calculation is dependent on low variance in the signal. High variance will yield lower scores, resulting in more false negatives. The author himself states this fact: "The [anomaly] signal should usually take values close to the mean of its distribution—this is an obvious consequence of its intended semantics" [24].

3 The three I's of variance

Variance in distributed systems is an important metric that directly affects potential for automated diagnosis. To reduce it, two complementary courses of action are necessary. During the design phase, system builders should make conscious decisions about which areas of the distributed system should be more predictable (exhibit low variance w/regard to important metrics). Since the complexity of distributed systems makes it unlikely they will be able to identify all of the sources of variance during design [3, 15, 22], they must also work to identify sources of variance during development and testing. To help with the latter, this section describes a nomenclature for variance sources that can help system builders reason about them and understand for which ones variance should be reduced.

Intentional variance sources: These are a result of a conscious tradeoff made by system builders. For example, such variance may emanate from a scheduling algorithm that lowers mean response time at the expense of variance. Alternatively, it may result from explicit anti-correlation added to a component to prevent it from entering synchronized, stuck states (e.g., Microreboots [26]). Labeling a source as intentional indicates the system builder will not try to reduce its variance.

Inadvertent variance sources: These are often the result of poorly designed or implemented code; as such, their variance should be reduced or eliminated. For example, such variance sources may include functions that exhibit extraordinarily varied response times because they contain many different control paths (spaghetti code). In [18], Kasick et al. describe how such high variance functions were problematic for an automated diagnosis tool developed for PVFS. Such variance can also emanate from unforeseen interactions between components, or may be the result of extreme contention for a resource (for example, due to poor scheduling decisions made by the datacenter scheduler). The latter suggests that certain performance problems can be diagnosed directly by localizing variance. In fact, while developing Spectroscope [27], we found that high variance was a good predictor of contention.

Intrinsic variance sources: These sources are often a result of fundamental properties of the distributed system or datacenter—for example, the hardware in use. Short of architectural changes, their variance cannot be reduced. Examples include non-flat topologies within a datacenter or disks that exhibit high variance in bandwidth between their inner and outer zones [3].

Variance from intentional and intrinsic sources may be a given, so the quality of predictions made by automation tools in these areas will suffer. However, it is important to guarantee their variance does not impact predictions made for other areas of the system. This may be the case if the data granularity used by an automation tool to make predictions is not high enough to distinguish between a high variance source and surrounding areas. For example, problems in the software stack of a component may go unnoticed if an automation tool does not distinguish it from a high-variance disk. To avoid such scenarios, system builders should help automation tools account for high variance sources directly—for example, by adding markers around them that are used by automation tools to increase their data granularity.

4 VarianceFinder

To illustrate a variance-oriented mindset, this section proposes one potential mechanism, called *VarianceFinder*, for helping system builders identify the main sources of variance in their systems during development and testing. The relatively simple design outlined here focuses on reducing variance in response times for distributed storage systems such as Ursa Minor [1], Bigtable [8], and GFS [13]. However, we believe this basic approach could be extended to include other performance metrics and systems.

VarianceFinder utilizes end-to-end traces (Section 4.1) and follows a two-tiered approach. First, it shows the

variance associated with aspects of the system's overall functionality that should exhibit similar performance (Section 4.2). Second, it allows system builders to select functionality with high variance and identifies the components, functions, or RPCs responsible, allowing them to take appropriate action (Section 4.3). We believe this tiered approach will allow system builders to expend effort where it is most needed.

4.1 End-to-end tracing

To identify sources of variance within a distributed system, a fine-grained instrumentation mechanism is needed. End-to-end tracing satisfies this requirement, as it captures the detailed control flow of individual requests within and across the components of a distributed system with as little as 1% overhead. Many implementations exist, all of which are relatively similar [5, 11, 29, 30]. Figure 1 shows an example *request-flow graph* generated from Stardust [30], the tracing mechanism used in Ursa Minor [1]. Note that nodes in this graph indicate instrumentation points reached by the request, whereas edges are annotated with performance metrics—in this case the latency between executing successive instrumentation points.

4.2 Id'ing functionality & first-tier output

To identify functionality that should exhibit similar performance, VarianceFinder utilizes an informal expectation, common in distributed storage systems, that requests that take the same path through the system should incur similar performance costs. For example, system builders generally expect READ requests whose metadata and data hit in a NFS server's cache to perform similarly, whereas they do not expect this for requests that take different paths because some miss in cache and others hit in it. VarianceFinder groups request-flow graphs that exhibit the same structure—i.e., those that represent identical activities and execute the same trace points—into *categories* and calculates average response times, variances, and squared coefficients of variation (C^2) for each. C^2, which is defined as $(\frac{\sigma}{\mu})^2$, is a normalized measure of variance and captures the intuition that categories whose standard deviation is much greater than the mean are worse offenders than those whose standard deviation is less than or close to the mean. In practice, categories with $C^2 > 1$ are said to have high variance around the mean, whereas those with $C^2 < 1$ exhibit low variance around the mean.

The first-tier output from VarianceFinder consists of the list of categories ranked by C^2 value. System builders can click through highly-ranked categories to see a graph view of the request structure, allowing them to determine whether it is important. For example, a highly-ranked category that contains READ requests likely will be deemed important, whereas one that contains rare requests for the names of mounted volumes likely will not.

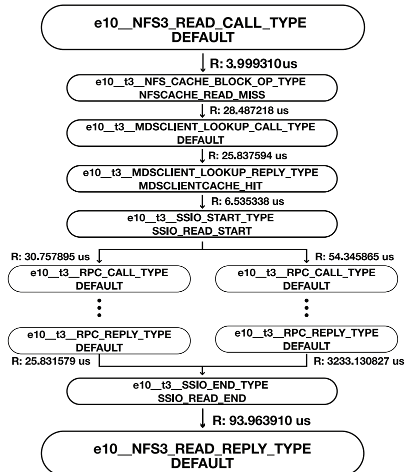

Figure 1: Example request-flow graph. The graph shows a striped READ in the Ursa Minor [1] distributed storage system. Nodes represent instrumentation points and edges are labeled with the time between successive events. Node labels are constructed by concatenating the machine name (e.g., `e10`), component name (e.g., `NFS3`), instrumentation-point name (e.g., `READ_CALL_TYPE`), and an optional semantic label (e.g., `NFSCACHE_READ_MISS`). Due to space constraints, instrumentation points executed on other components as a result of the NFS server's RPC calls are not shown.

4.3 Second-tier output & resulting actions

Once the system builder has selected an important highly-ranked category, he can use VarianceFinder to localize its main sources of variance. This is done by highlighting the highest-variance edges along the critical path of the category's requests. Figure 2 illustrates the overall process. In some cases, an edge may exhibit high variance because of another edge—for example, an edge spanning a queue might display high variance because the component to which it sends data also does so. To help system builders understand these dependencies, clicking on a highlighted edge will reveal other edges that have non-zero covariance with it.

Knowing the edges responsible for the high variance allows the system builder to investigate the relevant areas of the system. Variance from sources that he deems inadvertent should be reduced or eliminated. Alternatively, he might decide that variance from certain sources should not or cannot be reduced because they are intentional or intrinsic. In such cases, he should add tight instrumentation points around the source to serve as markers. Automation tools that use these markers to increase their data granularity—especially those that use end-to-end traces directly [5, 27, 29]—will be able to make better predictions about areas surrounding the high-variance source.

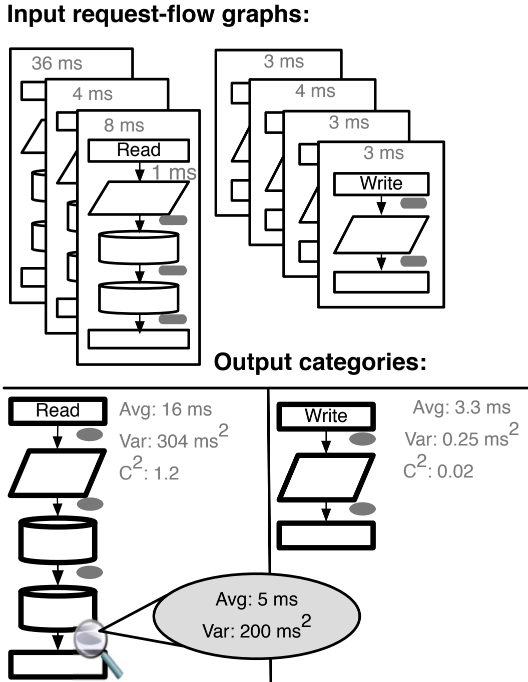

Figure 2: Example of how a VarianceFinder implementation might categorize requests to identify functionality with high variance. VarianceFinder assumes that requests that take the same path through a distributed system should incur similar costs. It groups request-flow graphs that exhibit the same structure into *categories* and calculates statistical metrics for them. Categories are ranked by the squared coefficient of variation (C^2) and high-variance edges along their critical path are automatically highlighted (as indicated by the magnifying glass).

Adding instrumentation can also help reveal previously unknown interesting behaviour. The system builder might decide that an edge exhibits high variance because it encompasses too large of an area of the system, merging many dissimilar behaviours (e.g., cache hits and cache misses). In such cases, extra trace points should be added to disambiguate them.

5 Discussion

This paper argues that variance in key performance metrics needs to be addressed explicitly during design and implementation of distributed systems, if automated diagnosis is to become a reality. But, much additional research is needed to understand how much variance can and should be reduced, the difficulty of doing so, and the resulting reduction in management effort.

To answer the above questions, it is important that we work to identify the breakdown of intentional, inadvertent, and intrinsic variance sources in distributed systems and datacenters. To understand if the effort required to reduce variance is worthwhile, the benefits of better automation must be quantified by how real people utilize and react to automation tools, not via simulated experiments or fault injection. If this tradeoff falls strongly in favour of automation and intrinsic variance sources are the largest contributers, architectural changes to datacenter and hardware design may be necessary. For example, system builders may need to increase the rate at which they adopt and develop strong performance isolation [16] or insulation [32] techniques. Also, hardware manufacturers, such as disk drive vendors, may need to incorporate performance variance as a first-class metric and strive to minimize it.

Similarly, if (currently) intentional sources are the largest contributors, system builders may need to re-visit key design tradeoffs. For example, they may need to consider using datacenter schedulers that emphasize predictability and low variance in job completion times [10] instead of ones that dynamically maximize resource utilization at the cost of predictability and low variance [14].

Finally, automated performance diagnosis is just one of many reasons why low variance is important in distributed systems and datacenters. For example, strong service-level agreements are difficult to support without expectations of low variance. As such, many of the arguments posed in this paper are applicable in a much broader sense.

6 Conclusion

Though automation in large distributed systems is a desirable goal, it cannot be achieved when variance is high. This paper presents a framework for understanding and reducing variance in performance metrics so as to improve the quality of automated performance diagnosis tools. We imagine that there are many other tools and design patterns for reducing variance and enhancing predictability. In the interim, those building automation tools must consider whether the underlying system is predictable enough for their tools to be effective.

Acknowledgements

We thank Michelle Mazurek, Ilari Shafer, and Soila Kavulya for their feedback. We thank the members and companies of the PDL Consortium (including Actifio, APC, EMC, Emulex, Facebook, Fusion-io, Google, Hewlett-Packard Labs, Hitachi, Intel, Microsoft Research, NEC Labs, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, VMWare, and Western Digital) for their interest, insights, feedback, and support. This research was sponsored in part by a Google research award, NSF grant #CNS-1117567, and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC).

References

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: versatile cluster-based storage. *Conference on File and Storage Technologies*, pages 59–72. USENIX Association, 2005.

[2] Anonymous. Personal communications with Facebook and Google engineers, December 2010 & April 2011.

[3] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-stutter fault tolerance. *Hot Topics in Operating Systems*, pages 33–37. IEEE, 2001.

[4] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. *ACM SIGCOMM conference on data communication*, pages 13–24. ACM, 2007.

[5] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. *Symposium on Operating Systems Design and Implementation*, pages 259–272. USENIX Association, 2004.

[6] L. A. Barroso. Warehouse-Scale Computing: entering the teenage decade. *International Symposium on Computer Architecture*. ACM, 2011.

[7] J. Bovard. Slower is better: the new postal service. *Individual Liberty, Free Markets, and Peace*, **146**, 1991. http://www.cato.org/pubs/pas/pa-146.html.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. *Symposium on Operating Systems Design and Implementation*, pages 205–218. USENIX Association, 2006.

[9] M. Y. Chen, A. Accardi, E. Kiciman, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. *Symposium on Networked Systems Design and Implementation*, pages 309–322. USENIX Association, 2004.

[10] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. *European Conference on Computer Systems*, pages 99–112. ACM, 2012.

[11] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: a pervasive network tracing framework. *Symposium on Networked Systems Design and Implementation*. USENIX Association, 2007.

[12] G. R. Ganger, J. D. Strunk, and A. J. Klosterman. *Self-* Storage: brick-based storage with automated administration*. Technical Report CMU–CS–03–178. Carnegie Mellon University, August 2003.

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *ACM Symposium on Operating System Principles*, pages 29–43. ACM, 2003.

[14] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. *Symposium on Networked Systems Design and Implementation*. USENIX Association, 2011.

[15] S. D. Gribble. Robustness in Complex Systems. *Hot Topics in Operating Systems*, pages 21–26. IEEE, 2001.

[16] A. Gulati, A. Merchant, and P. Varman. mClock: handling throughput variability for hypervisor IO scheduling. *Symposium on Operating Systems Design and Implementation*. USENIX Association, 2010.

[17] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. *ACM SIGCOMM conference on data communication*, pages 243–254. ACM, 2009.

[18] M. P. Kasick, R. Gandhi, and P. Narasimhan. Behavior-based problem localization for parallel file systems. *Hot Topics in System Dependability*. USENIX Association, 2010.

[19] M. P. Kasick, J. Tan, R. Gandhi, and P. Narasimhan. Black-box problem diagnosis in parallel file systems. *Conference on File and Storage Technologies*. USENIX Association, 2010.

[20] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, **36**(1):41–50. IEEE, January 2003.

[21] E. Krevat, J. Tucek, and G. R. Ganger. Disks are like snowflakes: no two are alike. *Hot Topics in Operating Systems*. USENIX Association, 2011.

[22] J. C. Mogul. Emergent (Mis)behavior vs. Complex Software Systems. *European Conference on Computer Systems*, pages 293–304. ACM, 2006.

[23] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. *Symposium on Networked Systems Design and Implementation*. USENIX Association, 2012.

[24] A. J. Oliner, A. V. Kulkarni, and A. Aiken. Using correlated surprise to infer shared influence. *International Conference on Dependable Systems and Networks*, pages 191–200. IEEE/ACM, 2010.

[25] S. Patil and G. Gibson. Scale and concurrency of GIGA+: file system directories with millions of files. *Conference on File and Storage Technologies*. USENIX Association, 2011.

[26] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. *Recovery-oriented computing (ROC): motivation, definition, techniques, and case studies*. Technical report – UCB/CSD–02–1175. UC Berkeley, March 2002.

[27] R. R. Sambasivan, A. X. Zheng, M. D. Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. *Symposium on Networked Systems Design and Implementation*, pages 43–56. USENIX Association, 2011.

[28] K. Shen, C. Stewart, C. Li, and X. Li. Reference-driven performance anomaly identification. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 85–96. ACM, 2009.

[29] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. *Dapper, a large-scale distributed systems tracing infrastructure*. Technical report dapper-2010-1. Google, April 2010.

[30] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: Tracking activity in a distributed storage system. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 3–14. ACM, 2006.

[31] A. Traeger, I. Deras, and E. Zadok. DARC: Dynamic analysis of root causes of latency distributions. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. ACM, 2008.

[32] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: performance insulation for shared storage servers. *Conference on File and Storage Technologies*, 2007.