Ganesha: Black-Box Diagnosis of MapReduce Systems

Xinghao Pan, Jiaqi Tan, Soila Kavulya, Rajeev Gandhi, Priya Narasimhan
Electrical & Computer Engineering Department, Carnegie Mellon University
Pittsburgh, PA 15213
xinghao@cmu.edu, jiaqi.tan@alumni.cmu.edu, {spertet, rgandhi, priyan}@andrew.cmu.edu

ABSTRACT

Ganesha aims to diagnose faults transparently (in a black-box manner) in MapReduce systems, by analyzing OS-level metrics. Ganesha's approach is based on peer-symmetry under fault-free conditions, and can diagnose faults that manifest asymmetrically at nodes within a MapReduce system. We evaluate Ganesha by diagnosing Hadoop problems for the `Gridmix` Hadoop benchmark on 10-node and 50-node MapReduce clusters on Amazon's EC2. We also candidly highlight faults that escape Ganesha's diagnosis.

1. INTRODUCTION

Performance problems in distributed systems can be hard to diagnose and to localize to a specific node or a set of nodes. There are many challenges in problem localization (i.e., tracing the problem back to the culprit node) and root-cause analysis (i.e., tracing the problem further to the underlying code-level fault or bug, e.g., memory leak, deadlock). As we show, performance problems can originate at one node in the system and then start to manifest at other nodes as well, due to the inherent communication across components–this can make it hard to discover the original culprit node.

A *black-box* diagnostic approach aims to discover the culprit node by analyzing performance data from the OS or network, without having to instrument the application or to understand its semantics. The most interesting problems to diagnose are not necessarily the outright crash (fail-stop) failures, but rather those that result in a "limping-but-alive" system, i.e., the system continues to operate, but with degraded performance.

We describe Ganesha, our black-box diagnostic approach that we apply to diagnose such performance problems in Hadoop [3], the open-source implementation of MapReduce [1]. Ganesha is based on our hypothesis (borne out by observation) that fault-free nodes in MapReduce behave similarly. Ganesha looks for asymmetric behavior across nodes to perform its diagnosis. Inevitably, this black-box approach will not have coverage—faults that do not result in a consistent asymmetry across nodes will escape Ganesha's diagnosis.

Black-box diagnosis is not new. Other black-box diagnostic techniques [14, 12, 13] determine the root-cause of a problem, given the knowledge that a problem exists in the system (the techniques differ in how they "know" that a problem exists). In a MapReduce system with its potentially long-running jobs, the system might not provide us with quick indications of a job experiencing a problem. Thus, in contrast with other techniques, Ganesha attempts to determine, for itself, whether a problem exists and, if so, traces the problem to the culprit node(s).

In this paper, we describe Ganesha's black-box diagnostic approach, based on our (experimentally substantiated) hypotheses of a MapReduce system's behavior. We evaluate our diagnosis approach on the well-accepted, multi-workload `Gridmix` Hadoop benchmark on a 10-node and a 50-node MapReduce cluster on Amazon's EC2. Our evaluation is performed by injecting and studying real-world problems that have been reported either in the Hadoop issue tracker or the Hadoop users' mailing list. We demonstrate the black-box diagnosis of faults that manifest asymmetrically at "peer" Hadoop slave nodes in the system. We discuss equally our experiences with faults that escape Ganesha's diagnosis, and suggest how Ganesha can be synthesized with white-box metrics extracted by SALSA, our previously developed Hadoop log-analysis tools [9].

2. TARGET SYSTEM: MAPREDUCE

Hadoop [3] is an open-source implementation of Google's MapReduce [1] framework that enables distributed, data-intensive, parallel applications by decomposing a massive job into smaller tasks and a massive data-set into smaller partitions, such that each task processes a different partition in parallel. Hadoop uses the Hadoop Distributed File System (HDFS), an implementation of the Google Filesystem [2], to share data amongst the distributed tasks in the system. The Hadoop framework has a single master node running the NameNode (which provides the HDFS namespace) and JobTracker (which schedules Maps and Reduces on multiple slave nodes) daemon. Each slave node runs a TaskTracker (execution) and a DataNode (HDFS) daemon.

We evaluate Ganesha's problem-diagnosis approach on the `Gridmix` Hadoop benchmark on a 10-node and a 50-node MapReduce cluster on EC2. `Gridmix` is an increasingly well-accepted Hadoop benchmark that is used to validate performance across different Hadoop upgrades, for instance. `Gridmix` models a cluster workload by generating random data and submitting MapReduce jobs that mimic the observed data-access patterns in user jobs. `Gridmix` comprises 5 different jobs, ranging from an interactive workload to a large sort of compressed data. We scaled down the size of the dataset to 2MB of compressed data for the 10-node cluster and 200MB for the 50-node cluster so that the benchmark would complete in 30 minutes. For lack of space, we omit our results with other Hadoop benchmarks (that are simpler than `Gridmix`), such as `RandWriter`, `Sort`, `Nutch` and `Pig`.

3. PROBLEM STATEMENT & APPROACH

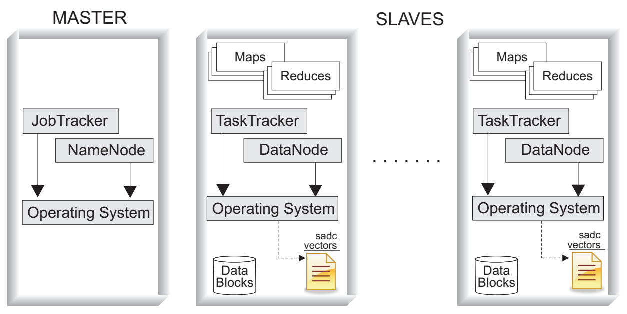

Figure 1: Architecture of Hadoop, showing our instrumentation points.

user	% CPU time in user-space
system	% CPU time in kernel-space
iowait	% CPU time waiting for I/O job
ctxt	Context switches per second
runq-sz	Number of processes waiting to run
plist-sz	Total number of processes and threads
ldavg-1	system load average for the last minute
eth-rxbyt	Network bytes received per second
eth-txbyt	Network bytes transmitted per second
pgpgin	KBytes paged in from disk per second
pgpgout	KBytes paged out to disk per second
fault	Page faults (major+minor) per second
bread	Total bytes read from disk per second
bwrtn	Total bytes written to disk per second

Table 1: Gathered black-box metrics (`sadc`-vector).

We seek to understand whether Ganesha can localize performance problems accurately and non-invasively, and what are the limitations of black-box diagnosis for Hadoop.

Hypotheses. We hypothesize that (1) Hadoop slave nodes exhibit a small number of distinct behaviors, from the perspective of black-box metrics; in a short interval (e.g. 1s) of time, the system's performance tends to be dominated by one of these behaviors; and (2) under fault-free operation, Hadoop slave nodes will exhibit similar behavior (*peer-symmetry*) over moderately long durations. We exploit these hypotheses for Ganesha's fault diagnosis. We make no claims about peer-symmetry or lack thereof under faulty conditions. (That is, we claim that fault-free operation results in peer-symmetry, but that faulty conditions may or may not also result in peer-symmety). Both our hypotheses are grounded in our experimental observations of Hadoop's behavior on `Sort`, `RandWriter`, `Pig`, `Nutch` and `Gridmix` benchmarks for Hadoop clusters of 10 and 50 nodes on EC2.

Goals. Ganesha should run transparently to, and not require any modifications of, both Hadoop and its applications. Ganesha should be usable in production environments, where administrators might not have the luxury of instrumenting applications but could instead leverage other (black-box) data. Ganesha should produce *low false-positive rates*, in the face of a variety of workloads for the system under diagnosis, and more importantly, even if these workloads fluctuate[1], as with `Gridmix`. Ganesha's data-collection should impose minimal instrumentation overheads on the system under diagnosis.

Non-Goals. Ganesha currently aims for (coarse-grained) problem diagnosis by identifying the culprit slave node(s). Clearly, this differs from (fine-grained) root-cause analysis, which would aim to identify the underlying fault or bug, possibly even down to the offending line of code. While Ganesha can be supported online, this paper is intentionally focused on Ganesha's offline analysis for problem diagnosis. We also do not target faults on the master node.

Assumptions. We assume that Hadoop and its applications are the dominant source of activity on every node. We assume that a majority of the Hadoop nodes are problem-free and that all nodes are homogeneous in hardware.

[1]Workload fluctuations can often be mistaken for anomalous behavior, if the system's behavior is characterized in terms of OS metrics alone. Ganesha, however, can discriminate between the two because fault-free peer nodes track each other under workload fluctuations.

4. DIAGNOSTIC APPROACH

For our problem diagnosis, we gather and analyze black-box (i.e., OS-level) performance metrics, without requiring any modifications to Hadoop, its applications or the OS. For black-box data collection, we use `sysstat`'s `sadc` program [5] to periodically gather a number of metrics (14, to be exact, as listed in Table 1) from */proc*, every second. We use the term `sadc`-vector to denote a vector of values of these metrics, all sampled at the same instant of time. We collect the time-series of `sadc`-vectors from each slave node and then perform our analyses to determine whether there is a performance problem in the system, and then to trace the problem back to the culprit Hadoop slave node.

4.1 Approach

MapReduce nodes exhibit a small number of distinct behaviors, from the perspective of black-box metrics. In a short interval (e.g. 1s) of time, the system's performance tends to be dominated by one of these behaviors. Hadoop's performance, over a short interval of time, can thus be classified into K distinct *profiles* corresponding to these distinct behaviors. Effectively, these profiles are a way to classify the observed `sadc`-vectors into K clusters (or centroids). While profiles do not represent semantically meaningful information, they are motivated by our observation that, over a short interval of time, each Hadoop slave node performs specific resource-related activities, e.g., computations (CPU-intensive), transfering data (network-intensive), disk access (I/O-intensive). The profiles, thus, represent a way to capture Hadoop's different behaviors, as manifested simultaneously on all of the 14 metrics. We use n to denote the number of slave nodes.

There are two phases to our approach — training and deployment — as shown in Figure 2. In the training phase, we learn the profiles of Hadoop by analyzing `sadc`-vector samples from slave nodes, gathered over multiple jobs in the fault-free case. In the deployment phase, we determine whether there is a problem for a given job, and if so, which slave node is the culprit. We validate Ganesha's approach by injecting various faults at one of the Hadoop slave nodes, and then determining whether we can indeed diagnose the culprit node correctly. The results of our validation are described in Section 5.1.

Training. We apply machine-learning techniques to learn

[Source] Reported Failure	[Fault Name] Fault Injected
[Hadoop users' mailing list, Sep 13 2007] CPU bottleneck resulted from running master and slave daemons on same machine	[CPUHog] Emulate a CPU-intensive task that consumes 70% CPU utilization
[Hadoop users' mailing list, Sep 26 2007] Excessive messages logged to file during startup	[DiskHog] Sequential disk workload writes 20GB of data to filesystem
[HADOOP-2956] Degrade network connectivity between DataNodes results in long block transfer times	[PacketLoss5/50] Induce $5\%, 50\%$ packet losses by dropping all incoming/outgoing packets with probabilities of $0.05, 0.5$
[HADOOP-1036] Hang at TaskTracker due to an unhandled exception from a task terminating unexpectedly. The offending TaskTracker sends heartbeats although the task has terminated.	[HANG-1036] Revert to older version and trigger bug by throwing NullPointerException
[HADOOP-1152] Reduces at TaskTrackers hang due to a race condition when a file is deleted between a rename and an attempt to call getLength() on it.	[HANG-1152] Emulate the race by flagging a renamed file as being flushed to disk and throwing exceptions in the filesystem code

Table 2: Injected faults, and the reported failures that they emulate. HADOOP-xxxx represents a Hadoop bug database entry.

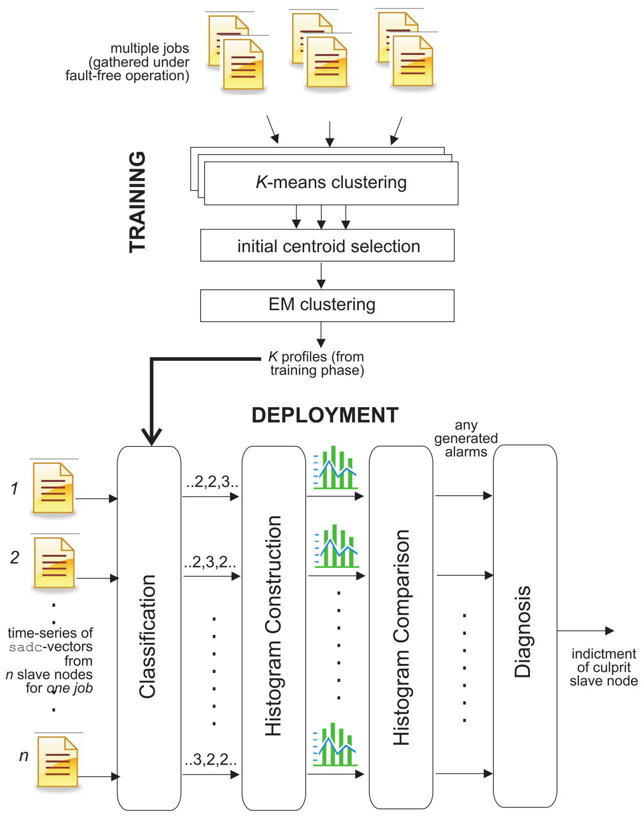
multiple jobs (gathered under fault-free operation)
TRAINING
K-means clustering
initial centroid selection
EM clustering
K profiles (from training phase)
DEPLOYMENT
1
2
time-series of sadc-vectors from *n* slave nodes for one job
n
Classification
..2,2,3..
..2,3,2..
..3,2,2..
Histogram Construction
Histogram Comparison
any generated alarms
Diagnosis
indictment of culprit slave node

Figure 2: Ganesha's approach.

the K profiles that capture Hadoop's behavior. We model our training data (a collection of `sadc`-vector time-series of fault-free experimental runs) as a mixture of K Gaussian distributions. The fault-free training data is used to compute the parameters—means and covariance matrices—of the K Gaussians. We enforce an equal prior over the K Gaussians, since the prior distributions of the K Gaussians may differ over different workloads. We do *not* assume that our training data is labeled, i.e., we do not know, a priori, which of the K Gaussians each gathered `sadc`-vector is associated with. Instead, we use the expectation-maximization (EM) algorithm [6] to learn the values of the unknown parameters in the mixture of Gaussians. Since the convergence time of the EM algorithm depends on the "goodness" of the initial values of these unknown parameters, we use K-means clustering to determine the initial values for the EM algorithm. In fact, we run the K-means clustering multiple times, with different initializations for the K-means clustering, in order to choose the best resulting centroid values (i.e., those with minimum distortion) as the initial values for the EM algorithm. The output of the EM algorithm consists of the means and covariance matrices, (μ_i, Σ_i), respectively of each of the K Gaussians. We chose a value of $K = 7$ in our experiments[2].

Deployment. Our test data consists of `sadc`-vectors collected from the n slave nodes for a single job. At every sampling interval, Ganesha classifies the test `sadc`-vector samples from each slave node into one of the K profiles, i.e., each test `sadc`-vector is mapped to the best Gaussian, (μ_i, Σ_i). If the test `sadc`-vector differs significantly from all of the K Gaussians, it is classified as "unknown". For each of the n slave nodes, we maintain a histogram of all of the Gaussian labels seen so far. Upon receiving a new classified `sadc`-vector for a slave node j, Ganesha incrementally updates the associated histogram, H_j, as follows. The histogram count values of all the labels are multiplied by an exponential decay factor, and 1 is added to the count value of the label that classifies the current `sadc`-vector. From our peer-symmetry hypothesis, slave nodes should exhibit similar behavior over moderately long durations; thus, we expect the histograms to be similar across all of the n slave nodes. If a slave node's histogram differs from the other nodes in a statistical sense, then, Ganesha can indict that "odd-slave-out" as the culprit.

To accomplish this, at each time instant, we perform a pairwise comparison of the histogram, H_j, with the remaining histograms, $H_l, l \neq j$, of the other slave nodes, l. The square root of the Jensen-Shannon divergence, which is a symmetric version of the Kullback-Leibler divergence and is known to be metric[3], is used as the distance measure to compute the pairwise histogram distance between slave nodes. An alarm is raised for a slave node if its pairwise distance is

[2]We discovered that as we increased K, the mean squared error of K-means decreases rapidly for $K < 7$ and slowly for $K > 7$. Our choice of $K = 7$ is guided by the 'elbow' rule-of-thumb: an additional cluster does not add significant information.

[3]A distance between two objects is 'metric' if it exhibits symmetry, triangular inequality, and non-negativity.

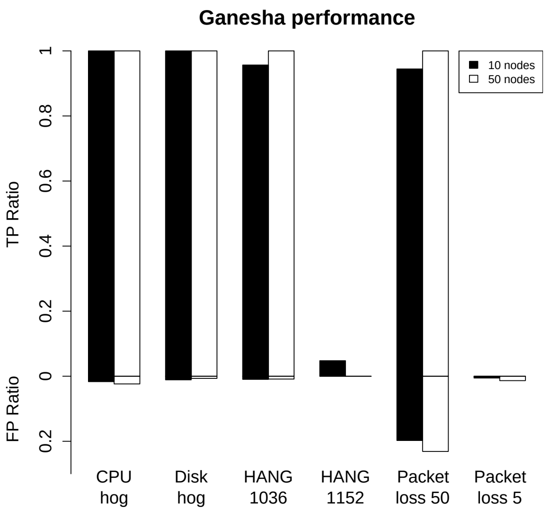

Figure 3: Diagnosis results for Ganesha on faults injected in Hadoop for fault-workload pairs.

more than a threshold value with more than $\frac{n-1}{2}$ slave nodes. An alarm is treated merely as a suspicion; repeated alarms are needed for indicting a node. Thus, Ganesha maintains an exponentially weighted alarm-count raised for each of the slave nodes in the system. Ganesha indicts a node as the culprit if that node's exponentially weighted alarm-count exceeds a predefined threshold value.

5. EXPERIMENTAL VALIDATION

We analyzed system metrics from two (10-slave nodes and 50-slave nodes) clusters running Hadoop 0.18.3. Each node consisted of an AMD Opeteron 1220 dual-core CPU with 4GB of memory, Gigabit Ethernet, and a dedicated 320GB disk for Hadoop, running amd64 Debian/GNU Linux 4.0.

We selected our candidate faults from real-world problems reported by Hadoop users and developers in: (i) the Hadoop issue tracker [4] from February 2007 to February 2009, and (ii) 40 postings from the Hadoop users' mailing list from September to November 2007. We describe our results for the injection of the six specific faults listed in Table 2. For each injected fault, we collected 20 traces on the 10-node cluster and 6 traces on the 50-node cluster, with Hadoop's speculative execution enabled for half of the traces. We demonstrate that Ganesha is able to detect three of the injected faults (*CPUHog*, *DiskHog* and *HANG-1036*), and discuss Ganesha's shortcomings at detecting the other three faults (*PacketLoss5*, *PacketLoss50* and *HANG-1152*).

5.1 Results

We evaluated Ganesha's approach using the true-positive (TP) and false-positive (FP) ratios across all runs for each fault. Figure 3 summarizes our results. A node with an injected fault that is correctly indicted is a true-positive, while a node without an injected fault that is incorrectly indicted is a false-positive. Thus, the true-positive and false-positive ratios are computed as:

$$TP = \frac{\#\text{ faulty nodes correctly indicted}}{\#\text{ nodes with injected faults}}$$

$$FP = \frac{\#\text{ nodes without faults incorrectly indicted}}{\#\text{ nodes without injected faults}}$$

Speculative execution did not have a significant effect on our results. Thus, we present results without discriminating experiments by whether speculative execution was enabled.

Figure 3 demonstrates that Ganesha was able to achieve almost perfect TP ratios of 1.0 and FP ratios of 0.0 for *CPUHog*, *DiskHog*, and *HANG-1036* on all cluster sizes. For *PacketLoss50*, Ganesha achieved high TP and FP ratios. Conversely, both TP and FP ratios for *HANG-1152* and *PacketLoss5* were low. We discuss these results in Section 6.

In addition, we compute the the false-alarm rate to be the proportion of slave nodes indicted in fault-free runs; this turns out to be 0.03 in both the 10-node and 50-node cases. These low false-alarm rates across both cluster sizes suggest that, in the case where nodes are indicted by Ganesha, a fault is truly present in the system, albeit not necessarily at the node(s) indicted by Ganesha.

6. DISCUSSIONS

Here, we candidly discuss some of the injected faults that Ganesha was less effective at diagnosing. Ganesha achieved low true-positive ratios at diagnosing *HANG-1152*, indicating that our view of the faulty node was obscured such that, under the injected fault, the behavior of the faulty node did not deviate significantly from the fault-free nodes. This is due to the relationship between the workload and the injected faults: specifically, the injected fault occurs late in the processing cycle, during the later part of a Reduce. The manifestations of *HANG-1152* are internal to the processing of each Reduce, and are triggered only near the end of each Map-to-Reduce processing cycle, where the jobs would have ended, thus squelching the manifestation of the fault before the fault can cause significant deviation in behavior on the faulty nodes.

This is in contrast to *HANG-1036*, which is a hang triggered at the beginning of Map processing, which we detect with an almost perfect true-positive ratio. This is a limitation of diagnosing using only black-box operating system performance counters, which limits us from utilizing application specific information to obtain finer-grained views. However, we have experienced more success at diagnosing application hangs using white-box finer-grained information from Hadoop's natively-generated logs [9], and we intend to study how white-box and black-box information can be jointly synthesized to diagnose failures more effectively.

Ganesha also achieved a high FP ratio for *PacketLoss50*, reflecting the highly correlated nature of the fault. Nodes that attempted to communicate with the culprit node also exhibited a slowdown in activity as they waited on the culprit node. A non-culprit slave node appears anomalous, as compared to other slave nodes, whenever it communicates with the culprit node. Hence, Ganesha indicts a significant number of non-culprit nodes as well. However, by using white-box metrics extracted from Hadoop's logs, we show in [11] that we can identify the culprit node with high accuracy, even under such correlated faults.

On the other hand, Ganesha achieves low TP and FP ratios for *PacketLoss5*, because TCP is resilient against minor packet losses, but not against major packet losses. Hence, the minor packet losses are masked by TCP's own reliable delivery mechanisms and the faulty node does not deviate significantly from other nodes.

7. RELATED WORK

We compare Ganesha with two sets of related work. First, we discuss other work on performance debugging for MapReduce systems. We then discuss other failure diagnosis techniques, most which target multi-tiered enterprise systems.

Performance debugging for MapReduce. [7] is an integrated tracing framework which requires modifications to code and network protocols. It has been applied to Hadoop to build and visualize request paths. [8] is a framework for collecting, storing and visualizing logs, and allows for plug-in statistical modules for analysis. Ganesha could potentially be implemented as one such module. However, [7, 8] do not automatically diagnose failures in MapReduce systems. Instead, they present information that aids the user in debugging performance problems.

Our prior work, SALSA [9] also performs automated failure diagnosis, but uses white-box metrics as extracted from Hadoop's logs. SALSA and Ganesha are complementary approaches; [11] shows how the two may be integrated to achieve better diagnosis results. Mochi [10] builds on state-machine models extracted by SALSA to build conjoined causal control- and data-flow paths of Hadoop's execution.

Other failure diagnosis work. Ganesha collects black-box performance counters from the OS. It models "normal" behavior as that observed on the majority of slave nodes; that is, Ganesha performs peer comparison for failure diagnosis. Like Ganesha, [12, 13] also collect resource utilization metrics. However, this is done at the thread level in [12]. Further, [12] captures events at the kernel, middleware and application layers. An anomaly is detected when a new request does not match any previously observed cluster of requests. [13] requires additional measurement of whether service-level objectives are violated. Problems are identified by comparing their signatures to previously observed clusters of signatures. However, MapReduce systems allow for arbitrary user code and thus does not lend itself well to the historical comparisons used in [12, 13].

[14, 15] collect only request path information. [14] infers paths from unmodified messaging layer messages; [15] collects messages at the middleware layer. [14] aims to isolate performance bottlenecks rather than detecting anomalies. [15] performs historical and peer comparison of path shapes to detect anomalies. However, paths in MapReduce follow the same Map to Reduce shape (except in straightforward cases of outright crashes or errors). Performance problems are not likely to result in changes in path shape, thus rendering a technique based on path shapes less useful in a MapReduce setting than in multi-tiered enterprise systems targetted in [15].

8. CONCLUSION AND FUTURE WORK

We describe Ganesha, a black-box diagnosis technique that examines OS-level metrics to detect and diagnose faults in MapReduce systems. Ganesha relies on experimental observations of MapReduce behavior to diagnose faults. The two, experimentally-based key hypotheses — peer symmetry of slave nodes and a small number of profiles of MapReduce behavior – drive Ganesha's diagnosis algorithm.

We are currently extending Ganesha to diagnose correlated faults by using data- and control-flow dependencies extracted by our Hadoop log-analysis tools [9]. Also, we are integrating Ganesha with other diagnosis techniques to enable improved diagnosis [11]. Also, Ganesha's learning phase assumes metrics with Gaussian distributions; we plan to investigate if the diagnosis can be improved by using other, possibly non-parametric, forms of clustering. We also expect to run Ganesha online by deploying its algorithms within our ASDF online problem-diagnosis framework [16].

9. REFERENCES

[1] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pp 137–150, San Francisco, CA, Dec 2004.

[2] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *SOSP*, pp 29–43, Lake George, NY, Oct 2003.

[3] Hadoop. `http://hadoop.apache.org/core`.

[4] Apache's JIRA issue tracker, 2006. `https://issues.apache.org/jira`.

[5] S. Godard. SYSSTAT, 2008. `http://pagesperso-orange.fr/sebastien.godard`.

[6] D. R. A. Dempster, N. Laird. Maximum likelihood from incomplete data via the em algorithm. *J. of the Royal Statistical Society*, 39:1,38, 1977.

[7] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. X-Trace: A pervasive network tracing framework. In *NSDI*, Cambridge, MA, Apr 2007.

[8] G. F. Cretu-Ciocarlie, M. Budiu, M. Goldszmidt. Hunting for Problems with Artemis. In *USENIX Workshop on Analysis of System Logs*, San Diego, CA, Dec 2008.

[9] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. SALSA: Analyzing logs as state machines. In *USENIX Workshop on Analysis of System Logs*, San Diego, CA, Dec 2008.

[10] J. Tan, X. Pan, S. Kavulya, R. Gandhi, P. Narasimhan. Mochi: Visual Log-Analysis Based Tools for Debugging Hadoop. HotCloud, San Diego, CA, Jun 2009.

[11] X. Pan, Blind Men and the Elephant: Piecing Together Hadoop for Diagnosis. Masters Thesis, Carnegie Mellon University, 2009. Technical Report: CMU-CS-09-135, Carnegie Mellon University, May 2009

[12] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *OSDI*, San Francisco, CA, Dec 2004.

[13] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP*, pp 105–118, Brighton, U.K., Oct 2005.

[14] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed system of black boxes. In *SOSP*, pp 74–89, Bolton Landing, NY, Oct 2003.

[15] E. Kiciman and A. Fox. Detecting application-level failures in component-based internet services. *IEEE Trans. on Neural Networks*, 16(5):1027–1041, Sep 2005.

[16] K. Bare, M. Kasick, S. Kavulya, E. Marinelli, X. Pan, J. Tan, R. Gandhi, and P. Narasimhan. ASDF: Automated online fingerpointing for Hadoop. Technical Report CMU-PDL-08-104, Carnegie Mellon University, May 2008.