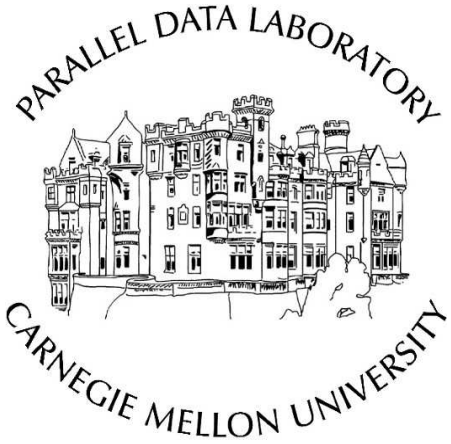

Ganesha: Black-Box Fault Diagnosis for MapReduce Systems

Xinghao Pan, Jiaqi Tan, Soila Kavulya, Rajeev Gandhi, Priya Narasimhan

CMU-PDL-08-112
September 2008

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Ganesha aims to diagnose faults transparently in MapReduce systems, by analyzing OS-level metrics alone. 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. While our training is performed on smaller Hadoop clusters and for specific workloads, our approach allows us to diagnose faults in larger Hadoop clusters and for unencountered workloads. We also candidly highlight faults that escape Ganesha's black-box diagnosis.

Acknowledgements: This work is partially supported by the NSF CAREER Award CCR-0238381, NSF Award CCF-0621508, and the Army Research Office grant number DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to the Center for Computer and Communications Security at Carnegie Mellon University.

Keywords: Hadoop, Fault diagnosis, Clustering

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 [12], the open-source implementation of MapReduce [7]. 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 [2, 4, 6] determine the root-cause of a problem, given the knowledge that there is a problem 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 explore when such a black-box diagnostic approach can and cannot work, based on our hypotheses of MapReduce system behavior. We demonstrate the black-box diagnosis of faults that manifest asymmetrically at "peer" nodes in the system. More interestingly, we can diagnose: (i) different faults by training on fault-free data, (ii) faults in larger MapReduce clusters although we train on smaller MapReduce clusters, (iii) faults for unencountered workloads, although we train on specific workloads. We candidly discuss our experiences with faults (such as those that manifest symmetrically at all nodes, or those that travel around the system) that escape Ganesha's diagnosis, and suggest ways in which we can address them.

2 Target System: MapReduce

Hadoop [12] is an open-source implementation of Google's MapReduce [7] 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 [18], to share data amongst the distributed tasks in the system. HDFS splits and stores files as fixed-size blocks (except for the last block).

Hadoop uses a master-slave architecture, as shown in Figure 1, with a unique master host and multiple slave hosts. The master host typically runs two daemons: (1) the JobTracker that schedules and manages all of the tasks belonging to a running job; and (2) the NameNode that manages the HDFS namespace by providing a filename-to-block mapping, and regulates access to files by clients (i.e., the executing tasks). Each slave host runs two daemons: (1) the TaskTracker that launches tasks on its host, as directed by the JobTracker; the TaskTracker also tracks the progress of each task on its host; and (2) the DataNode that

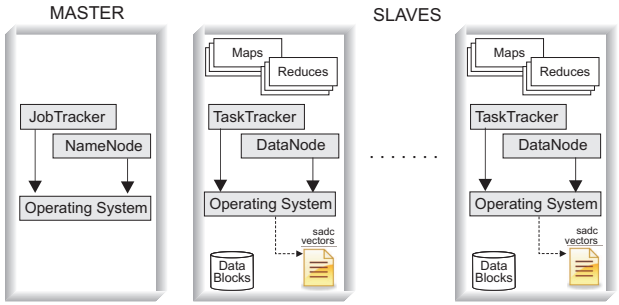

Figure 1: Architecture of Hadoop, showing our instrumentation points.

serves data blocks (on its local disk) to HDFS clients.

We explore fault-diagnosis for three candidate MapReduce workloads, of which the first two are commonly used to benchmark Hadoop:

• *RandWriter*: write 32 GB of random data to disk;
• *Sort*: sort 3 GB of records;
• *Nutch*: open-source distributed web crawler for Hadoop [13] representative of a real-world workload

3 Problem Statement & Approach

We seek to understand whether Ganesha can localize performance problems accurately and non-invasively, and whether Ganesha can assist us in understanding the limitations of black-box diagnosis for MapReduce systems.

Hypotheses. We hypothesize that 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. We also hypothesize that, under fault-free operation, MapReduce slave nodes will exhibit similar behavior over moderately long durations. We exploit this *peer-symmetry* for Ganesha's fault diagnosis. We make no claims about the symmetry or lack thereof under faulty conditions.

Goals. Ganesha should run transparently to, and not require any modifications of, both the hosted applications and any middleware that they might use. 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 in the case of Nutch. 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.

[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 in workload fluctuations.

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).

[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 wrote 20GB of data to filesystem
[HADOOP-2956] Degraded network connectivity between DataNodes results in long block transfer times	[PacketLoss] 50% packet loss
[HADOOP-1036] Infinite loop at slave node due to an unhandled exception from a Hadoop subtask that terminates unexpectedly. The offending slave node sends heartbeats although the subtask has terminated.	[HADOOP-1036] Revert to older version and trigger bug by throwing NullPointerException

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

Assumptions. We assume that the target MapReduce system is the dominant source of activity on every node. We assume that a majority of the MapReduce nodes are problem-free and that all nodes are homogeneous in hardware. We also assume that MapReduce's speculative execution is disabled.

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 to collect these metrics. For black-box data collection, we use `sysstat`'s `sadc` program [14] to periodically gather a number of metrics (14, to be exact, as listed in Table 1) from */proc*, at a sampling interval of one second. We use the term `sadc`-vector to denote a vector containing samples of these 14 metrics, all extracted at the same instant of time. We collect the time-series of `sadc`-vector samples 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 slave node.

4.1 Approach

From our hypothesis, Hadoop's performance, over a short interval of time, can be classified into K distinct *profiles*. 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 ob-

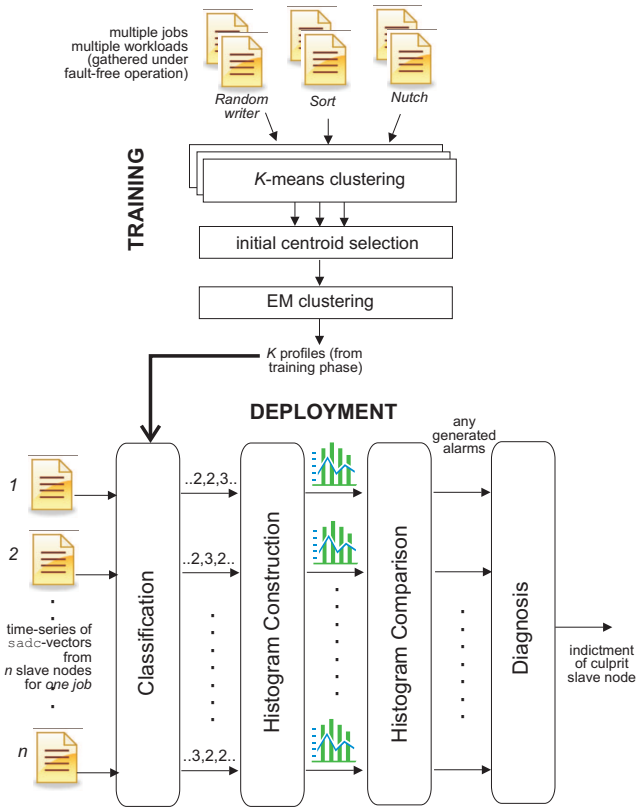

Figure 2: Ganesha's approach.

servation 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 and multiple workloads 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. Note that we train on multiple workloads, but can test with any given workload—this is a fairly important aspect of our technique. 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 5.2.

Preprocessing First, prior to using received trace data, all nodes must have an equal number of samples spaced at equal intervals. While we collected data at a rate of 1 sample per second, this collection was best-effort and samples could have been missed. We performed a Locally Weighted Linear Regression to interpolate data before processing (Figure 3).

Next, we normalized the data by taking the logarithms of each value (Figure 4). We observed that samples with large values in particular metrics corresponded to high levels of activity with respect to the metrics, and small values corresponded to low levels of activity. The sample values also tended to have higher variance during high levels of activity as compared to the variance of sample values during low levels of activity. Since the metric values are larger during periods of high levels of activity, by taking the

logarithm of the metrics, we are able to greatly reduce the variance during periods of high activity. Hence, the logarithm of the metrics during periods of both high and low activities have similar variances.

Finally, we normalized the logarithms of the values by the standard deviations of each metric as observed in the training data. In *K*-means clustering, the Euclidean distance is used as the distance measure between data points. By normalizing all values by the standard deviations, we prevent the scenario where a few metrics of large variance dominate the *K*-means process. We can thus be more confident that the resultant *K*-means clusters are truly representative of the distinct behaviors, rather than a separation of samples based on the few dominant metrics.

Training. We apply machine-learning techniques to learn the *K* profiles that capture Hadoop's behavior (Figure 6). We model our training data (a collection of `sadc`-vector time-series of fault-free experimental runs of *Sort*, *RandWriter* and *Nutch*) 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 [1] 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 $K = 6$ in our experiments.

Deployment. Our test data consists of `sadc`-vectors collected from the *n* slave nodes, under a single job of a given workload. 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) (Figure 5). If the test `sadc`-vector differs significantly from all of the *K* Gaussians, it is classified as "unknown". Next, in the deployed phase (Figure 7), Ganesha examines these classifications of each of the data points. 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 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 [8] and is known to be metric[2], 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 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 threhold value.

[2] A distance between two objects is "metric" if it has the properties of symmetry, triangular inequality, and non-negativity.

function FILLDATA(*sampleTimes, data*)
  initialize $completeSampleTimes \leftarrow \mathbf{0}$
  initialize $completeData \leftarrow \mathbf{0}$
  for all t from $\min_i sampleTimes_i$ to $\max_i sampleTimes_i$ **do**
    if $t \notin sampleTimes$ **then**
      $h \leftarrow$ 3rd smallest value in $\{abs(s-t) : s \in sampleTimes\}$
      for all $t' \in sampleTimes$ **do**
        $weight(t') \leftarrow \exp\left(-\frac{(t'-t)^2}{2h^2}\right)$
      end for
      $d \leftarrow LWLR(t, weights, sampleTimes, data)$
    else
      $d \leftarrow data_i$, where $t = sampleTimes_i$
    end if
    $completeSampleTimes_t \leftarrow t$
    $completeData_t \leftarrow d$
  end for
  return (*completeSampleTimes, completeData*)
end function

Figure 3: Fill in missing data points using locally weighted linear regression. Note: $LWLR(t, weights, \vec{X}, \vec{Y})$ performs locally weighted linear regression using the training set $\{(x_i, y_i)\}$ where each data point (x_i, y_i) has weight $weight_i$. The function returns the predicted value for y when $x = t$.

function NORMALIZEDATA(*data*)
  for all d in *data* **do**
    $d \leftarrow \log(d+1)/\sigma$
  end for
  return *data*
end function

Figure 4: Normalize a given set of data

function CLASSIFYDATA(*data*, μ, Σ)
  for i from 1 to $size(\mu)$ **do**
    $w_i \leftarrow \frac{1}{\sqrt{|\Sigma_i|}} \exp\left\{-\frac{1}{2}(data-\mu_i)^T \Sigma_i^{-1} (data-\mu_i)\right\}$
    $(U, \Lambda) = eig(\Sigma_i)$ such that $\Sigma_i = U\Lambda U^T = U\Lambda^{1/2}(U\Lambda^{1/2})^T$
    $dist_i \leftarrow (U\Lambda^{1/2})^{-1}(data-\mu_i)$
  end for
  $label \leftarrow \max_i w_i$
  if $dist_{label} >$ fixed threshold **then**
    $label \leftarrow size(\mu)+1$
  end if
  return *label*
end function

Figure 5: Classify data point to a cluster (or identify as unknown behavior)

function TRAINING(sadc_*data*, K) ▷ sadc_*data* is a set of sadc-vectors, i.e. each sadc_$data_i$ is a sadc-vector
  $M \leftarrow size(\texttt{sadc\_}data)$
  $\sigma \leftarrow standardDeviation(\log(\texttt{sadc\_}data + 1))$
  $trainingData \leftarrow normalizeData(\texttt{sadc\_}data, \sigma)$
  for i from 1 to 5 **do**
    $(centriods_i, labels_i) \leftarrow kMeans(trainingData, K)$
    $distortion_i \leftarrow \sum_{j=1}^{M} abs(trainingData_j - centriods_i(labels_i(j)))$
  end for
  $index \leftarrow \arg\min_i distortion_i$
  $\tilde{\mu} \leftarrow centriods_{index}$
  for i from 1 to K **do**
    $\tilde{\Sigma}(i) \leftarrow covariance(\{trainingData_j : labels_{index}(j) = i\})$
  end for
  $(\mu, \Sigma) \leftarrow EM\text{-}GMM(trainingData, K, \tilde{\mu}, \tilde{\Sigma})$
  return (σ, μ, Σ)
end function

Figure 6: Training phase

procedure GANESHA(σ, μ, Σ, λ, $\{sampleTimes_i, \texttt{sadc\_data}_i\}^N$, *threshold*)
  for i from 1 to N **do**
    initialize $H_i \leftarrow \mathbf{0}$
    $data_i \leftarrow normalizeData(\texttt{sadc\_data}_i, \sigma)$
    $(sampleTimes_i, data_i) \leftarrow fillData(sampleTimes_i, data_i)$
  end for
  for all time t **do**
    for i from 1 to N **do**
      $label_i \leftarrow classifyData(data_i(t), \mu, \Sigma)$
      $H_i \leftarrow \lambda H_i$
      $H_i(label_i) \leftarrow H_i(label_i) + 1$
    end for
    for all node pair i, j **do**
      $distMatrix(i, j) \leftarrow \sqrt{JSD(H_i, H_j)}$
    end for
    for all node i **do**
      if $count_j(distMatrix(i, j) > threshold) > \frac{1}{2}N$ **then**
        raise alarm at node i
        if 20 consecutive alarms raised **then**
          indict node i
        end if
      end if
    end for
  end for
end procedure

Figure 7: Deployed phase. Note: $JSD(H_i, H_j)$ is the Jensen-Shannon divergence between the histograms at nodes i and j.

		small-cluster						small-cluster cross-validated					
Injected Fault	Fault Manifestation	*RandWriter*		*Sort*		*Nutch*		*RandWriter*		*Sort*		*Nutch*	
		TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
CPUHog	Static asymmetric	1.0	0	1.0	0	1.0	0	1.0	0	1.0	0.03	1.0	0
DiskHog	Static asymmetric	1.0	0.03	1.0	0.03	1.0	0	1.0	0	0.9	0.2	1.0	0
PacketLoss	Traveling asymmetric	0.7	0.03	0.6	0.2	0.7	0.33	0.6	0	1.0	0.48	0.7	0.4
HADOOP-1036	Symmetric	1.0	0	0	0	0	0	0.9	0	0	0	0	0
		large-cluster						large-cluster cross-validated					
CPUHog	Static asymmetric	0.9	0.03	1.0	0	1.0	0.04	1.0	0.05	1.0	0.11	1.0	0.04
DiskHog	Static asymmetric	1.0	0.06	1.0	0	1.0	0.01	1.0	0.05	1.0	0.07	1.0	0.01
PacketLoss	Traveling asymmetric	0.5	0.06	0.9	0.53	0.7	0.06	0.5	0.04	1.0	0.63	0.8	0.09
HADOOP-1036	Symmetric	1.0	0.02	0	0	0	0	1.0	0.04	0.4	0.06	0	0

Table 3: Diagnosis results for Ganesha on faults injected in Hadoop for fault-workload pairs; TP = true-positive ratio, FP = false-positive ratio. Traveling asymmetric and symmetric fault-manifestations are grayed-out because they are outside Ganesha's current diagnosis.

5 Experimental Validation

We analyzed system metrics from two Hadoop 0.12.3 clusters[3]: `small-cluster` (6-node: 5-slave, 1-master) and `large-cluster` (16-node: 15-slave, 1-master). 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 [11] from October 1, 2006 to December 1, 2007, and (ii) 40 postings from the Hadoop users' mailing list from September to November 2007. We describe our results for the injection of the four specific faults listed in Table 2. We intentionally chose these four faults for discussion in this paper to show where Ganesha works (*CPUHog* and *DiskHog*), where it does not (*PacketLoss* and *HADOOP-1036*), and to describe why. We describe our goals for experimentation.

[Goal #1] What kinds of faults escape Ganesha's diagnosis? We sought to explore what kinds of faults manifested in a way that escaped our diagnosis, namely, (i) faults that manifested symmetrically across all nodes (violating our hypothesis) or (ii) faults that manifested asymmetrically, but where the fault-manifestation traveled across the system.

[Goal #2] Can we train on fault-free data alone and diagnose a variety of faults on Hadoop clusters? We performed our training on fault-free runs. We then deployed Ganesha on the fault-injected runs. The goal was to study Ganesha's ability to diagnose faults without needing to train on them.

[Goal #3] Can we train on smaller Hadoop clusters and diagnose faults on larger Hadoop clusters? We performed our training on fault-free runs on the `small-cluster` for each of the three workloads (30 runs total). We then deployed Ganesha on 10 fault-injected runs for each fault-workload pair on `large-cluster`. The goal was to study Ganesha's ability to diagnose faults on a cluster of 16 nodes, when training was performed on a 6-node Hadoop cluster.

[Goal #4] Can we train on any two workloads, and then diagnose faults for the third workload? We termed this phase of evaluation *cross-validation*. We performed our training on 10 fault-free `small-cluster` runs of only two of the three workloads (20 runs total), and then deployed Ganesha for 10 fault-injected runs for the third workload on the `small-cluster`. The goal was to assess Ganesha's ability to diagnose faults for unencountered (untrained) workloads.

We then combined goals #2, #3 and #4 to extrapolate our diagnosis to larger clusters and to unencoun-

[3]We recognize that 16 nodes is, by no means, a realistically large Hadoop cluster. We use the adjectives "large" and "small" simply to denote the clusters of two different sizes.

tered workloads, both at once, while avoiding the need to gather fault-induced training data.

5.1 [Goal #1] Fault-Manifestation Types

With the fault injected only on a single node, we were able to observe interesting fault manifestations.
Static, asymmetric manifestation: The culprit node behaves differently from other nodes, and this manifestation does not travel to other nodes. Examples are *CPUHog* and *DiskHog*. These faults are detectable and diagnosable correctly by Ganesha, based on our hypotheses of similar slave-node behavior.
Traveling, asymmetric manifestation: Nodes affected by the fault behave asymmetrically, but the asymmetry travels to nodes other than the culprit. An example is *PacketLoss*, where nodes that attempt to communicate with the culprit node also exhibit slowdown in activity as they wait on the culprit node. These faults are detectable, but not diagnosable, by Ganesha.
Symmetric manifestation: All nodes are affected by the fault, leading to symmetric (faulty) behavior. An example is *HADOOP-1036*. These faults are not detectable or diagnosable by Ganesha. However, this fault was detectable for the *RandWriter* workload, because the workload is such that each node independently writes random data to disk, so that only the culprit node halted processing, while the remaining nodes did not.

Thus, there are different kinds of fault manifestations, from a black-box viewpoint. Based on our peer-similarity hypothesis, only some of them are detectable and diagnosable by Ganesha's black-box approach.

5.2 Results

We evaluated Ganesha's approach using the true-positive (TP) and false-positive (FP) ratios [9] across all runs for each fault-workload pair. Table 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:

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

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

In addition, we compute the the false-alarm rate to be the proportion of slave nodes indicted in fault-free runs. Table 4 summarizes these results. The low false-alarm rates 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.

[Goal #2] Table 3 demonstrates that we can, indeed, train on fault-free data alone to diagnose faults. We achieved high TP ratios and low FP ratios for faults with static asymmetric manifestations.

[Goal #3] Ganesha was successful at diagnosing faults with static asymmetric manifestations, achieving TP ratio ≤ 0.9 across all workloads with very low FP ratios, in the `small-cluster` (non cross-validated) case. This applied also to the `large-cluster` case, demonstrating Ganesha's ability to extrapolate from behaviors learned from a small cluster to diagnose faults on a larger cluster.

[Goal #4] Ganesha was able to extrapolate its diagnosis across workloads; TP ratios remained high and FP ratios remained very low, in moving from diagnosing using non-cross-validated to the cross-validated cases. This extrapolation across workloads was effective even when extrapolating to the size of the cluster, as the TP ratios remained ≤ 0.9, and FP ratios remained low across all workloads, going from the `small-cluster` cross-validated to the `large-cluster` cross-validated cases.

	Non cross-validated		Cross-validated	
Workload	6-node	16-node	6-node	16-node
RandWriter	0	0.05	0	0.04
Sort	0	0.02	0	0
Nutch	0	0.05	0	0.04

Table 4: False alarm rates on fault-free runs.

6 Related Work

Diagnosing faults in distributed systems involves: (i) collecting data about the system, (ii) localizing faults to individual requests or nodes, and (iii) identifying root-causes of these problems. We compare Ganesha's approach with recent work. We note that Ganesha targets systems with long-lived jobs, as compared to work on Internet services with many short-lived jobs [4, 6, 5, 2, 15].

Instrumentation sources. Both Ganesha and Magpie [4] use black-box system metrics. Magpie uses expert-input to associate resource-usage with individual user requests, while Ganesha does not need expert-input as we extract coarse-grained aggregate observations. Pip [17], X-trace [10] and Pinpoint [15] extract white-box metrics about individual request paths through systems by tagging messages between components. [2] infers request paths from unmodified messaging layer messages. While X-trace modifies the messaging layer in the system, Ganesha is transparent to the system.

Fault localization. Pip identifies failed requests via violations of programmer-inserted expectations. Magpie clusters fine-grained resource-usage profiles of requests, and can identify anomalous ones with large numbers of observed requests. [6] uses externally-supplied violations of *a priori* performance thresholds to identify failures, while in many Internet-service systems dealt with by current techniques, failed requests are easily detected at egress points [5, 2, 15]. Current techniques do not identify problems before requests fail. This is difficult because problems can occur at many points. Ganesha detects and localizes problems in such systems.

Root-cause analysis. Given knowledge of failed requests, [6, 5] perform root-cause analysis on requests known to have failed by using clustering and decision trees respectively. [15, 2] identify components along request paths that contribute to failures or slowdowns. These techniques uncover root-causes of a failure given known failed requests, while Ganesha identifies problematic requests in systems where this is hard.

In addition, X-trace has been applied to MapReduce systems to build and visualize request paths [16], but not yet to automatically detect problems. Ganesha automatically identifies nodes on which problems occurred.

7 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 peer-symmetry to diagnose faults. Ganesha is able to extrapolate its fault diagnosis to larger MapReduce clusters and to unseen workloads.

We propose to diagnose faults with traveling asymmetric manifestations by identifying fault propagations using data- and control-flow dependencies extracted using white-box information that we previously explored [19]. Also, we intend to extend Ganesha to include white-box metrics, which may enable extended diagnosis. Also, Ganesha's learning phase of Ganesha 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 our diagnosis online by deploying Ganesha as a module in our ASDF online problem-diagnosis framework [3].

Symmetric Ganesha was not able to detect symmetric failures, achieving TP rates of nearly 0, as we rely

on detecting nodes with behaviors significantly different from other nodes, so that such symmetric manifestations escape our diagnosis. However, the symmetric failure was detected on the *RandWriter* workload, because the workload is a special case in which each node independently wrote random data to disk, so that only the node with the injected failure stopped processing, while the remaining nodes did not depend on it.

References

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

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed system of black boxes. In *ACM Symposium on Operating Systems Principles*, pages 74–89, Bolton Landing, NY, Oct 2003.

[3] 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 PDL, May 2008.

[4] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec 2004.

[5] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *International Conference on Autonomic Computing*, pages 36–43, New York, NY, May 2004.

[6] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *ACM Symposium on Operating Systems Principles*, pages 105–118, Brighton, United Kingdom, Oct 2005.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 137–150, San Francisco, CA, Dec 2004.

[8] D. M. Endres and J. E. Schindelin. A new metric for probability distributions. *Information Theory, IEEE Transactions on*, 49(7):1858–1860, 2003.

[9] T. Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27:861–874, 2006.

[10] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. X-Trace: A pervasive network tracing framework. In *USENIX Symposium on Networked Systems Design and Implementation*, Cambridge, MA, Apr 2007.

[11] The Apache Software Foundation. Apache's JIRA issue tracker, 2006. https://issues.apache.org/jira.

[12] The Apache Software Foundation. Hadoop, 2007. http://hadoop.apache.org/core.

[13] The Apache Software Foundation. Nutch, 2007. http://lucene.apache.org/nutch.

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

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

[16] A. Konwinski, M. Zaharia, R. Katz, and I. Stoica. X-tracing Hadoop. *Hadoop Summit*, Mar 2008.

[17] P. Reynolds, C. Killian, J. Wiener, J. Mogul, M. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *USENIX Symposium on Networked Systems Design and Implementation*, San Jose, CA, May 2006.

[18] H. Gobioff S. Ghemawat and S. Leung. The Google file system. In *ACM Symposium on Operating Systems Principles*, pages 29 – 43, Lake George, NY, Oct 2003.

[19] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. SALSA: Analyzing Logs As State Machines. Technical Report CMU-PDL-08-111, Carnegie Mellon University PDL, Sep 2008.