Blind Men and the Elephant: Piecing Together Hadoop for Diagnosis

Xinghao Pan, Jiaqi Tan
DSO National Laboratories
Singapore
{pxinghao,tjiaqi}@dso.org.sg

Soila Kalvulya, Rajeev Gandhi, Priya Narasimhan
Electrical and Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA, U.S.A.
{spertet,rgandhi}@ece.cmu.edu, priya@cs.cmu.edu

***Abstract*—Google's MapReduce framework enables distributed, data-intensive, parallel applications by decomposing a massive job into smaller (Map and Reduce) tasks and a massive data-set into smaller partitions, such that each task processes a different partition in parallel. However, performance problems in a distributed MapReduce system can be hard to diagnose and to localize to a specific node or a set of nodes. On the other hand, the structure of large number of nodes performing similar tasks naturally affords us opportunities for observing the system from multiple viewpoints.**

We present a "Blind Men and the Elephant" (Blimey) framework in which we exploit this structure, and demonstrate how problems in a MapReduce system can be diagnosed by corroborating the multiple viewpoints. More specifically, we present algorithms within the Blimey framework based on OS-level performance counters, on white-box metrics extracted from logs, and on application-level heartbeats. We show that our Blimey algorithms are able to capture a variety of faults including resource hogs and application hangs, and to localize the fault to subsets of slave nodes in the MapReduce system.

In addition, we discuss how the diagnostic algorithms' outcomes can be further synthesized in a repeated application of the Blimey approach. We present a simple supervised learning technique which allows us to identify a fault if it has been previously observed.

***Keywords*-MapReduce, Hadoop, Failure Diagnosis**

I. Introduction

Problem diagnosis is the science of automatically discovering if, what, when, where, why and how problems occur in systems and programs. In general, however, answering these questions may not be easy. At times, it is not entirely clear what constitues a problem or if one even exists. As systems today become increasingly large and complex, programmers and sysadmins have more trouble reasoning about their systems. The vast amounts of data can also easily overwhelm a human debugger.

MapReduce (MR) [1] is a programming framework and implementation introduced by Google for data-intensive cloud computing on commodity clusters. Hadoop [2], an open-source Java implementation of MapReduce, is used by Yahoo! and Facebook. Debugging the performance of Hadoop programs is difficult because of their scale and distributed nature. For example, Yahoo! Search Webmap is a large production Hadoop application that runs on a 10,000+ core Linux cluster and produces data that is used in Yahoo! Web search queries; the cluster's raw disk is 5+ petabytes in size [3]. Hadoop can be certainly debugged by examining the local (node-specific) logs of its execution. These logs can be overwhelmingly large to analyze manually, e.g., a fairly simple *Sort* workload running for 850 seconds on a 5-node Hadoop cluster generates logs at each node, with a representative node's log being 6.9MB in size and containing 42,487 lines of logged statements. Furthermore, to reason about system-wide, cross-node problems, the logs from distinct nodes must be collectively and manually analyzed. Hadoop provides a simple web-based user interface that reveals key statistics about job execution. However, the user interface can be cumbersome to navigate when debugging a performance problem in a large MapReduce system, not to mention the fact that some kinds of problems might completely escape (i.e., not be visible in) this interface.

In this paper, we propose to perform problem diagnosis for Hadoop systems by corroborating and synthesizing multiple distinct viewpoints of Hadoop's behavior. Hadoop, as a large distributed system, provides us with multiple sources (e.g. OS-level performance counters, tasks' durations as inferred from application logs) and multiple locations (the master node and large number of slave nodes) at which the instrumentation may be performed. We corroborate the instrumentated data from the different locations, and further synthesize this corroboration to piece together a picture of Hadoop's behavior for the purpose of problem diagnosis.

II. Problem statement

In contrast to traditional web enterprise systems, MapReduce systems are composed of large numbers of machines performing similar (though not necessarily identical) tasks. We seek to exploit this structure to diagnose performance problems in the MapReduce system.

There are two high-level goals in this paper. Firstly, we seek to indict faulty slave nodes for a variety for faults. Hadoop offers multiple locations at which we can instrument and observe the system. Corroborating the instrumentated data in the Blimey framework will allow us to indict faulty slave nodes. We also show that by synthesizing the outcomes of diagnostic algorithms, we can improve the localization of the fault, and furthermore identify the fault if it were previously seen.

Furthermore, in our diagnosis, we primarily target performance problems that result in a Hadoop job taking longer to complete than expected for a variety of reasons, including external/environmental factors on the node (e.g., a non-Hadoop process consuming system resources to the detriment of the Hadoop job), reasons not specific to user MapReduce code (e.g., bugs in Hadoop), or interactions between user MapReduce code and the Hadoop infrastructure (e.g., bugs in Hadoop that are triggered by user code). We intentionally do not target faults due to bugs in user-written MapReduce application code. We seek to have our diagnosis approach work in production environments, requiring no modifications to existing MapReduce-application code or existing Hadoop infrastructure.

However, we do not aim to have fine-grained diagnosis, that is, our diagnosis will not identify the root-cause, nor pinpoint exactly the offending line of code at which the fault originated. We also do not aim to have complete coverage of either faults or all possible instrumentation sources. While we keep these ultimate goals of problem diagnosis in mind, they are not the primary focus of this paper.

We make the assumption that MapReduce applications and infrastructure are the dominant sources of activity on every node. We also assume that a majority of the MapReduce nodes are problem-free and homogeneous in hardware.

III. Overview

A. Background: MapReduce and Hadoop

Hadoop [2] 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 (Map and Reduce) tasks and a massive data-set into smaller partitions, such that each task processes a different partition in parallel. A Hadoop job consists of a group of Map and Reduce tasks performing some data-intensive computation. Hadoop uses the Hadoop Distributed File System (HDFS) 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 with a Hadoop cluster having a unique master node and multiple slave nodes.

The master node 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 (the executing tasks). Each slave node runs two daemons: (1) the TaskTracker that launches tasks locally on its host, as directed by the JobTracker, and then tracks the progress of each of these tasks; and (2) the DataNode that serves data blocks (on its local disk) to HDFS clients. Hadoop provides fault-tolerance by using periodic keep-alive heartbeats from slave daemons (from TaskTrackers to the JobTracker and DataNodes to the NameNode). Each Hadoop daemon generates logs that record the local execution of the daemons as well as MapReduce application tasks and local accesses to data.

B. Synopsis of Blimey's Approach

There can be multiple perspectives of Hadoop's or a MapReduce application's behavior, e.g., from the operating-system's viewpoint, from the application's viewpoint, from the network's viewpoint, etc. In the mythological story of *The Blind Men and the Elephant* , each blind man arrives at a different conclusion based on his limited perspective of the elephant. It is only by corroborating all the blind men's perspectives that one can reconstruct a complete picture of the elephant. MapReduce, as a large distributed system, affords us many opportunities or instrumentation points to observe the system's behavior. Each view can be thought to correspond to a "blind man", and the MapReduce system itself to the elephant. By mediating across and synthesizing the different views, our approach, Blimey, acts as the wise man and is able to diagnose problems in MapReduce. In particular, we apply the Blimey approach at two different levels: instrumentation points and diagnostic algorithms.

- We present a number of diagnostic algorithms, each of which corroborates the views from different instrumentation sources at each node in the system.
- The diagnostic algorithms then provide a secondary perspective into the MapReduce system. Treating the different diagnostic algorithms as the blind men, we synthesize the algorithms' outcomes to identify the fault.

We produce two levels of diagnostic outcomes. First, a collection of diagnostic algorithms (Section IV-C) each identifies a set of slave nodes in the cluster that possibly increased the runtime of the job. Second, the synthesis of the outputs of these algorithms (Section V), given prior labelled training data, produces for each node, the most likely fault from a class of previously seen faults in the labelled training data (possibly no fault) present in the cluster, and whether that node suffers from the identifed fault.

[Blind Men #1] Views from instrumentation points. Large distributed systems have different instrumentation points from which the behavior and properties of the system can be simultaneously observed. Often, these instrumentation points can serve as somewhat redudnant or corroborating (albeit distinct) views of the system. We exploit the parallel distributed nature of MapReduce systems by applying the Blimey framework to three distinct types of instrumentation views of the system: (white-box) heartbeat-related metrics, extracted from the Hadoop logs; (white-box) execution-state metrics, extracted from the Hadoop logs; and (black-box) performance and resource-usage metrics, extracted from the operating system.

[Blind Men #2] Views from diagnostic algorithms. Faults in a distributed system can manifest on different sets of metrics in different ways. A given fault might manifest only on a specific subset of metrics or, alternatively, might manifest in a system-wide correlated manner, i.e., the fault originates on one node but also manifests on some metrics on other (otherwise untainted) nodes, due to the inherent communication/coupling across nodes. Thus, diagnosis algorithms that focus on or analyze a selective set of metrics will likely miss faults or in the case of cascading fault-manifestations, wrongly indict nodes that exhibit any correlated manifestation of the fault. By further synthesizing the outcomes of our diagnostic algorithms, Blimey gains greater insight into the distributed nature of the fault, allowing it to identify the kind of fault as well as the true culprit node.

IV. Diagnostic Approach

This section describes the internals of the Blimey diagnotic approach, including the instrumentation sources and the diagnosis algorithms that analyze the corroborating instrumentation-based views of the system.

A. High-Level Intuition

First, Hadoop uses heartbeats as a keep-alive mechanism. The heartbeats are periodically sent from the slave nodes (TaskTrackers and DataNodes) to the master node (JobTracker and NameNode). Upon receipt of the heartbeat, the master node sends a *heartbeatResponse* to the slave node, indicating that it has received the heartbeat. Both the receipt of heartbeats at the JobTracker and of the heartbeatResponses at the TaskTrackers are recorded in the respective daemon's logs, together with the heartbeat's unique id. We corroborate these log messages in one of our diagnosis algorithms. In addition, upon completion of tasks, the TaskTrackers *proactively* send heartbeats to the JobTracker indicating the task completion. As such, the rate at which a TaskTracker sends heartbeats is indicative of the workload it is experiencing. In another diagnosis algorithm, we corroborate the rate of heartbeats across the slave nodes.

Secondly, a job consists of multiple copies of Map and Reduce tasks, each running the same piece of code, albeit operating on different portions of the dataset. We expect that the the Map tasks exhibit similar behavior with other Map tasks, and that Reduce tasks exhibit similar behavior with other Reduce tasks. More abstractly, since each Map task M_i is an instance from the global set of all Map tasks, any property $P(M_i)$ of the Map task M_i can be treated as a single sample from a global distribution of the property P over all Map tasks. The same is true for Reduce tasks. In particular, we are most interested in the completion times of Map and Reduce tasks. The local distribution of task completion times on each node is an instrumentation source, and in the absence of faults, these times should corroborate across all TaskTrackers.

Metric	Description
user	% CPU time in user-space
system	% CPU time in kernel-space
iowait	% CPU time waiting for I/O
ctxt	Context switches per second
runq-sz	# processes waiting to run
plist-sz	Total # of processes and threads
ldavg-1	system load average for the last minute
bread	Total bytes read from disk /s
bwrtn	Total bytes written to disk /s
eth-rxbyt	Network bytes received /s
eth-txbyt	Network bytes transmitted /s
pgpgin	KBytes paged in from disk /s
pgpgout	KBytes paged out to disk /s
fault	Page faults (major+minor) /s
TCPAbortOnData	# of TCP connections aborted with data in queue
rto-max	Maximum TCP retransmission timeout

Table I
GATHERED BLACK-BOX METRICS (`SADC`-VECTOR).

Finally, we apply a similar principle to the observations on each slave node's OS-level performance counters. Since each slave node executes a subset of the global set of tasks, and the OS-level performance counters are dependent on the slave node's workload, its OS-level performance counters can be thought of as a sampling of a global distribution of OS-level performance counters too.

B. Instrumentation Sources

Black-box: OS-level performance metrics. On each node in the Hadoop cluster, we gather and analyze black-box (i.e., OS-level) performance metrics, without requiring any modifications to Hadoop, the MapReduce applications or the OS to collect these metrics. For black-box data collection, we use `sysstat`'s `sadc` program [4] and a custom script that samples TCP-related and `netstat` metrics to collect 16 metrics (listed in Table I) from */proc*, once every second. We use the term `sadc`-vector to denote a vector containing samples of these 16 metrics, all extracted at the same instant of time. We then use these `sadc`-vectors as our (black-box) metrics for diagnosis.

White-box: Execution-state metrics. We collect the system logs generated by Hadoop's own native logging code from the TaskTracker and DataNode daemons on each slave node. We then use our Hadoop-log analysis tool (called SALSA [5] and, its successor, Mochi [6]) to extract inferred state-machine views of the execution of each daemon. The log-analysis generates the durations of Map and Reduce tasks executed on every node in the cluster as part of its output. We then examine the durations of these execution states as the metrics in our (white-box) diagnosis.

White-box: Heartbeat metrics. Heartbeat events are also recorded in Hadoop's native logs, and we extract these from the master-node (JobTracker, NameNode) and the slave-node (TaskTracker, DataNode) logs. Although both are derived from white-box instrumentation sources, these heartbeat metrics are orthogonal to the previously described execution-state metrics.

For each heartbeat event between the master node and a given slave node, a log entry is recorded in both the master-node's log and that specific slave-node's logs, along with with a matching monotonically increasing heartbeat sequence-number. Each (master node, slave node) pair has an independent, unique space of hearbeat sequence-numbers. Each message is timestamped with a millisecond-resolution timestamp. The master-node's log first records a message as it receives the heartbeat from the slave node, and the slave node's log then records a message as it receives the master node's acknowledgment/response for the same heartbeat. Hadoop has an interesting implementation artefact where the master node logs the slave node's heartbeat message, and then performs additional processing within the same thread, before it acknowledges the slave. Analogously, the acknowledgement/response is first processed by the slave node before it is finally logged. This artefact is exploited, as we explain in the next section.

C. Component Algorithms

1) Black-box Diagnosis:

Intuition: Each slave node in the MapReduce system executes a subset of the global set of Map and Reduce tasks. We note that all MapReduce jobs follow the same temporal ordering: Map tasks are assigned, and begin by reading input data from DataNodes; upon completion, the MapOutput data is Shuffled to the Reduce tasks; eventually the job terminates after the Reduce tasks write their outputs to the DataNodes. Since each slave node executes a subset of the global set of Map and Reduce tasks, this temporal ordering is reflected on the slave nodes as well. Hence, we expect that within reasonably large windows of time, slaves nodes encounter similar workloads that are reflective of the global workload of the MapReduce system. In the language of Blimey, each slave node is a "blind-man" who has a limited view of the entire system.

The workload on each slave node at every instant of time is represented by the black-box metrics that we collect on the slave node. More abstractly, we can represent the global workload of the MapReduce system as a global distribution of black-box metrics. The observed black-box metrics on each slave node is then a sampling of the global distribution at the time of collection. Our black-box diagnosis algorithm then corroborates the black-box views on slave nodes. A slave node whose black-box view differs significantly from the that of the other slave nodes indicted. We describe below how we perform the comparison in practice.

Algorithm: Our black-box algorithm consists of three parts: collection, sampling and corroboration. Firstly, we collect 14 metrics from */proc* and 2 TCP-related metrics from `netstat` (Table I). This is done for every slave node at a fixed time interval of 1 second. Each slave node maintains a window of the 120 most recently collected `sadc`- vectors.

A naive pair-wise comparison of each slave node's `sadc`-vectors with every other slave node's `sadc`-vectors would require $O(n^2)$ comparisons. Instead, to maintain scalability, we maintain, on each slave node, an approximation of the global distribution of black-box metrics. This approximate global distribution is constructed by collecting samples of `sadc`-vectors from random peer slave nodes.

The `sadc`-vectors on each slave node are then corroborated against the approximate global distribution on that node [7], resulting in $O(n)$ total comparisons. Notice that the diagnosis can be performed in a distributed fashion. The work done by each slave node scales at a constant $O(1)$ with the number of slave nodes. An alarm is raised for a slave node whenever the `sadc`-vectors on that slave node differs significantly from the approximated global distribution. An alarm is treated merely as a suspicion; repeated alarms are needed for indicting a node. Thus, we maintain an exponentially weighted alarm-count for each slave node. The slave node is then indicted when its exponentially weighted alarm- count exceeds a predefined value.

2) White-box Diagnosis:

Intuition: From our log-extracted state-machine views on each node, we consider the durations of maps and reduces. For each of these states of interest, we can compute the histogram of the durations of that state on the given node. As mentioned in Section IV-A, the durations for the state on a given node is a sample of the global distribution of the durations for that state across all nodes. The local distribution of durations is hence an estimate of the global distribution. According to our Blimey framework, the local distribution is a limited view of the global distribution, which is a property of the MapReduce system. We corroborate each local distribution against a global distribution, indicting nodes with local distributions that are dissimilar from the global distribution as being faulty. The intuition is that, for a given job, the tasks on each node are multiple copies of the same code, and hence should complete in comparable durations.

Algorithm: First, for a given state on each node, probability density functions (PDFs) of the distributions of durations are estimated from their histograms using a kernel density estimation with a Gaussian kernel to smooth the discrete boundaries in histograms. Then, an estimate of global distribution is built by summing across all local histograms. Next, the difference between these distributions from the global distribution is computed as the pair-wise distance between their estimated PDFs. We repeat this analysis over each window of time. As with the black-box algorithm , we raise an alarm for a node when its distance to the global distribution exceeds a set threshold, and indict it when the exponentially weighted alarm-count exceeds a predefined

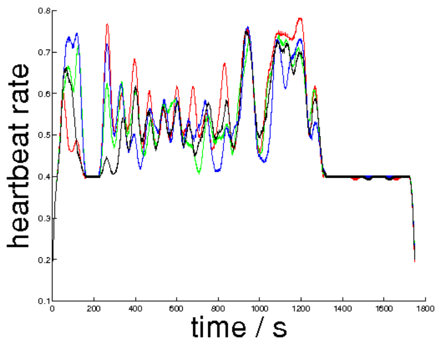

Figure 1. Heartbeat rates of 4 slave nodes throughout an experiment with no fault injected.

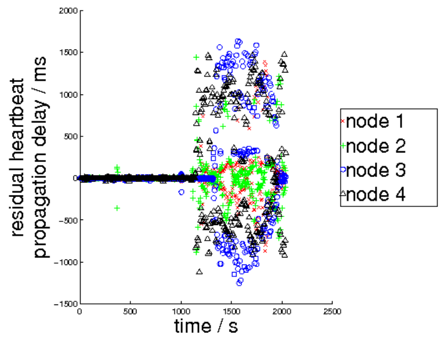

Figure 2. Residual heartbeat propagation delay of 4 slave nodes throughout an experiment with fault hang2051 injected at around 1000s.

value.

3) *Heartbeat-based Diagnosis:*

Heartbeat-rate Corroboration: In a Hadoop cluster, each slave node sends heartbeats to the master node at the same periodic interval across the cluster (this interval is adaptively increased across all TaskTracker nodes as cluster size increases). Hence, in the absence of faulty conditions, the same heartbeat rate (number of heartbeat messages logged per unit time) should be observed across all slave nodes (see Section IV-A). The heartbeat-rate is computed by smoothing over the discrete event series of heartbeats into a continuous time-series using a Gaussian kernel. Figure 1 shows the heartbeat rates of 4 slave nodes through an experiment with no fault injected. These heartbeat rates are then compared across slave nodes, by computing the difference between the rates and the median rate.

Heartbeat Propogation Delay: The Heartbeat Propagation Delay is the difference between the time at which a received heartbeat is logged at the JobTracker, and at which the received acknowledgement is logged at the TaskTracker for the same heartbeat. This delay includes both the network propagation delay, and the delay caused by computation occurring in the same thread as that for handling the heartbeat at both the JobTracker and TaskTracker. This difference in timestamps, however, is subject to clock synchronization and clock drift. We account for these differences by performing a local linear regression on the timestamp differences against time. If the true propagation delay is almost constant, the residuals of our local linear regression would be almost zero. On the other hand, if a heartbeat has a large residual heartbeat propagation delay, then either the heartbeat is anomalous compared to other heartbeats from the same TaskTracker, or there is a large variation in the true heartbeat propagation delay. Both cases are indicative of problems in the MapReduce system. Thus, we indict nodes for which there is a large average residual heartbeat propagation delay.

Figure 2 shows the residuals obtained from the local linear regression on timestamp difference against log message time. In this particular experiment, we injected hang2051, a JobTracker hang (see Table II). Before the fault was triggered, the residuals were mostly less than 100ms. After the fault was triggered, the residuals increased to about $\pm$1500ms.

V. SYNTHESIZING VIEWS

Different faults manifest differently on different metrics, resulting in different outcomes from our diagnostic algorithms. A particular fault may or may not manifest on a particular metric, and the manifestation may be correlated to varying degrees. Each of our diagnostic algorithms thus acts as a "blind-man" to give us a different perspective into the fault's effect on the MapReduce system. By synthesizing these perspectives, it is possible to identify the particular fault. More specifically, given a cluster, we would like to know, for each node, if it is faulty, and if so, which of the previously known faults it most closely resembles.

To this end, we represent each node by the diagnostic statistics that are generated by the algorithms. The diagnostic statistics for both our black-box and white-box algorithms are the exponentially weighted alarm-counts, for the heartbeat rate corroboration algorithm it is the difference between the node's heartbeat rate and the median rate, and for the heartbeat propagation delay algorithm it is the sum of residuals. For each node, we construct a vector consisting of the diagnostic statistics for each algorithm, and also the average of the diagnostic statistics across all other nodes in the cluster for each algorithm. The former captures the ability of the diagnostic algorithms to indict the faulty node, whereas the latter captures the degrees to which each fault manifests in a correlated manner on the diagnostic algorithms.

Using this representation, we are able to build classifiers for the faults. We chose to use decision trees as our classifiers, although it is also possible to use other types of classifiers. While decision trees tend not to have the best prediction errors, they have the added advantage of being easily understood, and reflect the natural manner in which human operators identify problems.

VI. Evaluation and Experimentation

A. Testbed and Workload

We analyzed system metrics from Hadoop 0.18.3 running on 10- and 50-node clusters on Large instances on Amazon's EC2. Each node had the equivalent of 7.5 GB of RAM and two dual-core CPUs, running amd64 Debian/GNU Linux 4.0. Each experiment consisted of one run of the `GridMix` workload, a well-accepted, multi-workload Hadoop benchmark. `GridMix` models the mixture of jobs seen on a typical shared Hadoop cluster by generating random input data and submitting MapReduce jobs in a manner that mimics observed data-access patterns in actual user jobs in enterprise deployments. The `GridMix` workload has been used in the real-world to validate performance across different clusters and Hadoop versions. `GridMix` comprises 5 different job types, ranging from an interactive workload that samples a large dataset, to a large sort of uncompressed data that access an entire dataset. We scaled down the size of the dataset to 2MB of compressed data for our 10-node clusters and 200MB for our 50-node clusters to ensure timely completion of experiments.

B. Injected Faults

We injected one fault on one node in each cluster to validate the ability of our algorithms at diagnosing each fault. The faults cover various classes of representative real-world Hadoop problems as reported by Hadoop users and developers in: (i) the Hadoop issue tracker [8] 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 seven specific faults listed in Table II.

VII. Results

A. Diagnostic algorithms

1) Slave node faults: We evaluated our diagnostic algorithms' performance at detecting faults by using true positive and false positive rates across all runs for each fault injected on a slave node, and for clusters of sizes of 10 and 50 slave nodes. A slave node with an injected fault that is correctly indicted is a true positive, while a slave node without an injected fault that is incorrectly indicted is a false positive. Thus, the true positive (TP) and false positive (FP) rates are computed as:

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

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

Figures 3 and 4 show the TP and FP rates of the algorithms for a 10 and 50 slave node cluster respectively.

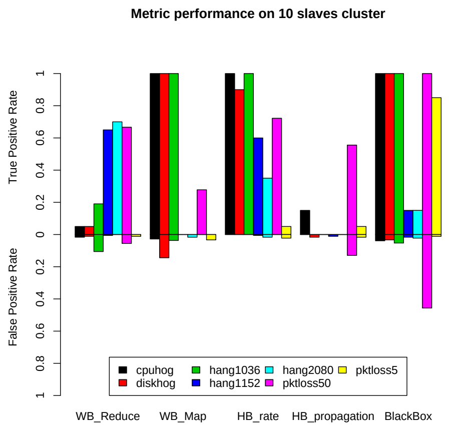

Figure 3. True positive and false positive rates for faults on slave nodes, on 10 slaves cluster.

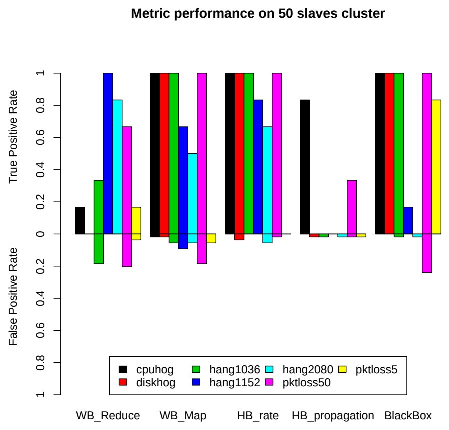

Figure 4. True positive and false positive rates for faults on slave nodes, on 50 slaves cluster.

The bars above the zero line represent the TP rates, and the bars below the zero line respresent the FP rates for each fault. Each group of 12 bars (6 above, 6 below zero line) show the TP and FP rates for a particular algorithm and instrumentation source. "WB_Reduce" and "WB_Map" are the diagnostic algorithms that corroborate Reudce and Map tasks' durations across the slave nodes, respectively. "HB_rate" and "HB_propagation" refer to the two hearbeat-based diagnostic algorithms that corroborate heartbeat rates across TaskTrackers, and heartbeat propagation delay between TaskTrackers and JobTrackers. The "BlackBox" algorithm, as previously described, corroborates OS-level performance counters across physical slave nodes.

From Fig 3 and 4, we observe that every fault is detected (with TP > 0.65) by at least one algorithm. In the case of resource- related faults (cpuhog, diskhog, pktloss5,

[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	[PacketLoss5/50] 5%, 50% packet losses by dropping all incoming/outcoming packets with probabilities of $0.01, 0.05, 0.5$
[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] Simulated the race by flagging a renamed file as being flushed to disk and throwing exceptions in the filesystem code
[HADOOP-2080] Reduces at TaskTrackers hang due to a miscalculated checksum.	[HANG-2080] Simulated by miscomputing checksum to trigger a hang at reducer
[HADOOP-2051] Hang at JobTracker due to an unhandled exception while processing completed tasks.	[HANG-2051] Revert to older version and trigger bug by throwing NullPointerException

Table II
INJECTED FAULTS, AND THE REPORTED FAILURES THAT THEY SIMULATE. HADOOP-XXXX REPRESENTS A HADOOP BUG DATABASE ENTRY.

pktloss5), our black-box algorithm has high TP rates of at least 0.83. BlackBox is also able to detect hang1036, but not hang1152 and hang2080. As hang1036 occurs in the Map task, an idle period results where the Reduce tasks block on waiting for output from the Map tasks. On the other hand, hang1152 and hang2080 occur in the Reduce tasks, so the slave node is able to continue consuming resources for execution of Map tasks, masking the hangs from the black-box point of view. Not suprisingly, the white-box algorithm WB_Map based on Map tasks' durations capture hang1036, and the algorithm WB_Reduce based on Reduce tasks' durations capture both hang1152 and hang 2080. The algorithm HB_rate detects most faults, except pktloss5, and the algorithm HB_propagation is most effective at detecting resource-related faults. Since heartbeat rate is a reflection of workload, we expect HB_rate to detect any fault that may adversely affect workload. On the other hand, HB_propagation targets a specific operation in the application: the sending of a heartbeat response from the JobTracker to the TaskTracker. The application hangs do not adversely affect this operation and are thus not detected, whereas the resource-related faults affect almost all operations in the system and are thus detected by HB_propagation.

We notice that pktloss5 is not sufficiently severe and can be eventually overcome by TCP's retransmissions. Thus, it fails to be detected by almost all algorithms (except BlackBox which explicitly tracks TCP-related metrics, and HB_Propagation, which targets a network-dependent operation). On the other hand, pktloss50 is sufficient severe that it affects the slave node's abiltiy to communicate and operate normally. All our algorithms detect, to varying TP rates, pktloss50. The severeness of pktloss50 also affect other slave nodes that block on reading or sending data to the faulty slave node, explaining the generally higher FP rates for pktloss50.

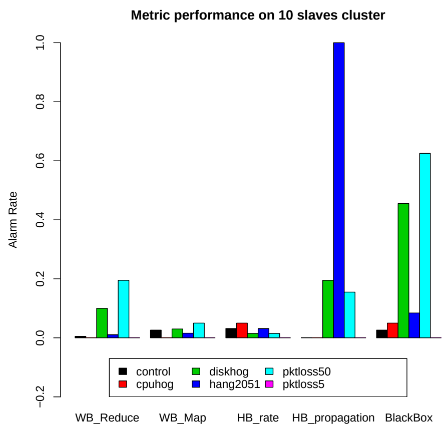

Figure 5. Alarm rates for faults on master nodes, on 10 slaves cluster.

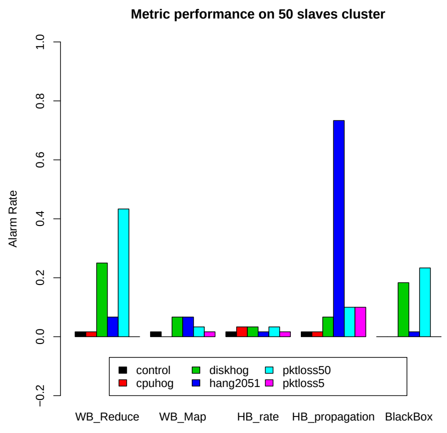

Figure 6. Alarm rates for faults on master nodes, on 50 slaves cluster.

2) Master node faults: As our diagnostic algorithms never explicitly indict the master node, it would be meaningless to discuss TP and FP rates for master node failures. Instead, we compute the alarm rate, that is, the proportion of slave nodes that were indicted by the algorithm:

$$alarm = \frac{\text{\# of indicted slave nodes}}{\text{\# of slave nodes}}$$

Figure 5 and 6 show the alarm rates for the 10 slave cluster and the 50 slave cluster respectively. Note that the fault "control" is not actually a fault; it refers to control experiments where we did *not* inject *any fault* into the system. We also used the control set to determine our thresholds. Specifically, we set the thresholds for each algorithm such that the alarm rates in the control sets would be 3% or less.

In the case of master node faults, the alarm rates only serve to give a notion of the effect of the master node fault on the slave nodes. An alarm rate significantly higher than 3% would indicate that the master node fault has a significant effect on the slave nodes. Note, however, that the diagnostic algorithms only indict slave nodes. As such, none of the algorithms localize the fault correctly, much less identify it. We fix this problem using the decision tree classification, as shown in the following section.

Nevertheless, we observe that the alarm rates vary between algorithms and faults. In particular, the alarm rate for hang2051 for HB_propagation is 1.0 on the 10-slave cluster, and 0.73 on the 50-slave cluster. This is because hang2051 is a master node hang, and HB_propagation is our only algorithm that explicitly accounts for the master node. All other algorithms corroborate views from multiple slave nodes. This demonstrates the usefulness of multiple types of corroboration.

B. Synthesizing outcomes of diagnostic algorithms

We generated a decision tree by using the `rpart` package of the statistical software R. The decision tree generated is shown in Fig 7.

The interior nodes (and the root) of the decision tree is labeled with an inequality of the form $X < t$ or $X >= t$, where X is a component of the representation (see Section V) and t is a threshold. X is of the form *algorithm_location*, where *algorithm* can be any of BB, WB_Map, WB_Reduce, HB_rate and HB_propagation (representing our black-box algorithm, white-box algorithm corroborating Map durations, white-box algorithm corroborating Reduce durations, heartbeat-based algorithm corroborating heartbeat rates, and heartbeat-based algorithm corroborating heartbeat propagation delays). *location* can be either self or other, with the former representing the diagnostic statistic of the algorithm for the node in concern, and the latter representing the mean of the diagnostic statistics of other nodes that were indicted by the algorithm.

Labels on the leaves have the form of *fault:suffix*, where *fault* indicates the most likely fault that occurred in the system; and a suffix of m indicates the fault occurred at the master node, s(+) indicates that the fault occurred on the slave node in concern, and s(-) indicates that the fault occurred on some slave node, but not on the node in concern.

Using the decision tree to classify the fault on a node would involve traversing the tree from root to leaf, following the left branch whenever the inequality at an interior node (or the root) evaluates to true, and the right branch otherwise. The labels at the leaves indicate whether the node in concern was faulty, and the fault that was most likely, among the known faults, to have occurred in the system.

In addition to merely visualizing the decision tree, we also evaluated the ability of our classification technique by using a N-fold cross- validation method. We randomly partitioned our experiments into N subsets, and classified the data in each subset using a decision tree trained on the remaining $N-1$ subsets. We chose $N = 296$ for our evaluation, as we had 296 experiments.

Fig 8 shows a confusion matrix of our classification results. Each row reprsents an actual class of faults, and each column represents the classification given by the decision tree. A cell in row i, column j would hold the proportion of nodes that were actually of class i, and were given the classification j by the decision tree. Lighter (less red) shades correspond to higher values. A perfect confusion matrix would have a lightly shaded diagonal and dark shades at all other non-diagonal cells.

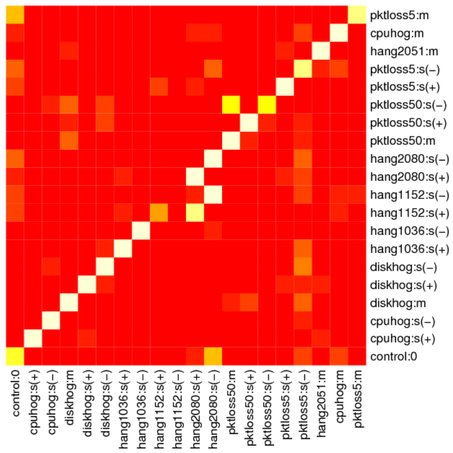

Figure 8. Confusion matrix of fault classification. Each row reprsents a class of faults, and each column represents the classification given. Lighter shades indicate the actual fault was likely to be given a particular classification.

Our confusion matrix shows that for most of the classes, we are able to achieve high classification accuracy. We discuss the few exceptions in greater detail in [7].

VIII. Discussions

A. Data skew and unbalanced workload

Three of the algorithms presented in this paper assume that Map and Reduce tasks are sufficiently similar for

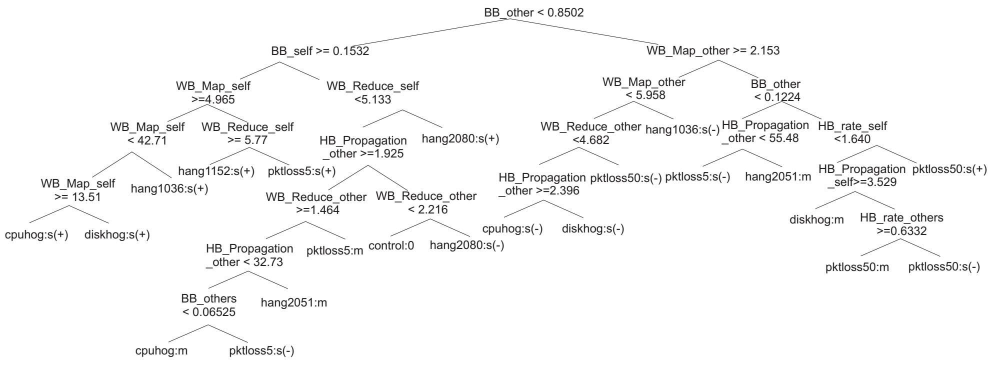

Figure 7. Decision tree, classifies faults by the outcomes of the diagnostic algorithms.

comparison. However, the tasks may execute the same code, but on different input data. In the case of a data skew or a heavily data-dependent load, the assumption may not hold. We argue, though, that in such cases, the job can be optimized by spreading the load more evenly, assuming homogeneous node capabilities. Thus, the alarms raised and indictments made by our algorithms do in fact indicate a performance problem and a potential for optimizing the distribution of load.

B. Heterogenous hardware

Another of our assumptions is that the hardware is homogeneous in the MapReduce cluster. While this may appear overly restrictive, we argue that it is not unrealistic. For instance, clusters operated on virtualized services like EC2 can be easily configured to have the same (virtualized) hardware. Nodes in physical clusters are often upgraded and replaced in batch. Furthermore, we do not insist that the entire MapReduce cluster be homogeneous. In the case where subsets of nodes have homogeneous hardware, our algorithms can be trivially adapted to work with the islands of homogeneous nodes.

C. Raw data versus secondary diagnostic outcomes

In generating the decision tree, we consumed the secondary diagnostic statistics generated by our algorithms as the representation for the nodes. An alternative would be to simply use the raw data (black-box metrics, white-box state durations, heartbeat rates and residual propagation delays) directly. This may not be feasible simply because of the large volume and high-dimensionality of the raw data. More importantly, it is often unclear what constitutes a fault from the raw data alone. For instance, an idle period from the black-box point-of-view may indicate a fault at a node if all other slave nodes are experiencing heavy workload, but the same idleness is legitimate behavior when the cluster is not processing any MapReduce jobs. Our diagnosis algorithms capture the notion of normal versus abnormal behavior. The secondary diagnostic statistics are thus semantically meaningful. The use of semantically meaningful data helps the sysadmin understand the fault pathology better; using raw data might not.

IX. Related Work

A. Diagnosing Failures and Performance Problems

[9] is the most similar to our work; Cohen *et al.* build signatures of the state of a running system by summarizing system metrics. These signatures are similar to our characterization of known performance problems, however our characterizations are based on the intermediate outputs of component diagnosis algorithms, and they serve to synthesize algorithms, while the signatures in [9] are built directly on observed system metrics. [10], [11] use path-based techniques to diagnose failures; [10] detects anomalously shaped paths (e.g. missing or additional elements) while [11] focused on accurately extracting causal paths. Both techniques were demonstrated with multi-tier Internet service systems, where paths for different requests can take on different shapes, so that path shape differences can highlight problems. However, shapes of processing paths in MapReduce are generally homogeneous. Thus, traditional path-based techniques will not be effective at diagnosing performance problems in MapReduce.

B. Diagnosing MapReduce Systems

Current techniques for diagnosing problems in MapReduce systems have examined only individual sources of instrumentation, while we have taken a holistic approach to utilize multiple information sources. [12] examined detailed causal network trace data generated using custom instrumentation and identified simple faults such as a slow disk by examining latencies along processing paths; these are similar to our white-box instrumentation, although our approach does not require the invasive instrumentation that [12] used. [13] considered Hadoop's logs as well, although they focused on only DataNode logs and only considered error events as opposed to the processing events which we considered. Consequently, we have been able to diagnose a larger range of faults than either work. [5], [14] constitute our prior work.

C. Instrumentation Tools

Magpie [15] enables causal tracing of request flows with attributed resource usage for each request, by using minimal programmer-specified program structure and inserted instrumentation. Unlike [15], our approach does not require any modification to the target application, namely, Hadoop. Also, [15] demonstrated anomaly detection by identifying previously unobserved behaviors. However, this is not possible in a MapReduce system which allows for arbitrary user code.

[12] discusses causal network request tracing via inserted instrumentation into the various communications layers; this represents an additional data source which can be included in our Blimey framework to enhance the robustness and fault coverage of our diagnosis.

X. Conclusions

A. Conclusion

We have presented the "Blind Men and Elephant" framework, and described how this approach is useful for fault diagnosis in a large parallel, distributed system like MapReduce. In particular, we have presented black-box, white-box, and heartbeat-based diagnostic algorithms within the Blimey framework, and demonstrated that by corroborating multiple instrumentation points of the system, one can identify suspect slave nodes. Further, in a repeated application of the Blimey approach, we show that the synthesis of the diagnostic algorithms' outcomes can aid the identify of the fault and localize the fault to the correct master or slave node.

B. Future work

An ongoing research aims to move the techniques presented in this paper online. This requires us to provide real-time tools to debug a live system, and this will be done using the ASDF framework [16]. It also requires that we can run our techniques in an incremental fashion. This can be easily done for the diagnostic algorithms by using finite windows or exponential weights.

To further increase the value of the tool to sysadmins, we need to present visualizations of the raw instrumentation data as well as the output from our algorithms, which respectively represent the primary and secondary viewpoints of Hadoop's behavior. Oftentimes it is easier to understand a visual, rather than textual, representation.

We are also looking to increase our coverage of instrumentation sources. We would like to incorporate X-trace or other path-based instrumentation. Corroboration of different instrumentation sources could possibly lead to other insights and algorithms in the Blimey framework.

References

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

[2] T. A. S. Foundation, "Hadoop," 2007, http://hadoop.apache.org/core.

[3] Y. D. Network, "Yahoo! launches world's largest hadoop production application (hadoop and distributed computing at yahoo!)," Feb 2008, http://developer.yahoo.net/blogs/hadoop/2008/02/yahoo-worlds-largest-production-hadoop.html.

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

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

[6] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Mochi: Visual Log-Analysis Based Tools for Debugging Hadoop," in *First Workshop on Hot Topics in Cloud Computing*, May 2009.

[7] X. Pan, "Blind men and the elephant: Piecing together hadoop for diagnosis," Master's thesis, Carnegie Mellon University, 2009.

[8] T. A. S. Foundation, "Apache's JIRA issue tracker," 2006, https://issues.apache.org/jira.

[9] 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*, Brighton, U.K., Oct 2005, pp. 105–118.

[10] 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*, vol. 16, no. 5, pp. 1027– 1041, Sep 2005.

[11] 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*, Oct 2003, pp. 74–89.

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

[13] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Mining console logs for large-scale system problem detection," in *Workshop on Tackling Systems Problems using Machine Learning*, Dec 2008.

[14] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Ganesha: Black-Box Diagnosis of MapReduce Systems," in *Second Workshop on Hot Topics in Measurement and Modeling of Computer Systems)*, Sep. 2008.

[15] 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.

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