Transparent System Call Based Performance Debugging for Cloud Computing

Nikhil Khadke, Michael P. Kasick, Soila P. Kavulya, Jiaqi Tan, Priya Narasimhan
Carnegie Mellon University
{*nkhadke, mkasick, spertet, jiaqit, priyan*}*@andrew.cmu.edu*

Abstract

Problem diagnosis and debugging in distributed environments such as the cloud and popular distributed systems frameworks has been a hard problem. We explore an evaluation of a novel way of debugging distributed systems, such as the MapReduce framework, by using system calls. Performance problems in such systems can be hard to diagnose and to localize to a specific node or a set of nodes. Additionally, most debugging systems often rely on forms of instrumentation and signatures that sometimes cannot truthfully represent the state of the system (logs or application traces for example). We focus on evaluating the performance debugging of these frameworks using a low level of abstraction - system calls. By focusing on a small set of system calls, we try to extrapolate meaningful information on the control flow and state of the framework, providing accurate and meaningful automated debugging.

1 Introduction

Performance problems are both common and inevitable in large scale computing, with root causes varying widely, from hardware issues to logical errors in software. One of the most prevalent forms of large scale computing is afforded by Google's MapReduce framework. MapReduce is a programming framework and paradigm for parallel distributed computation on commodity computer clusters [10], which allows programmers to easily process large data, by abstracting away low-level details of distributed execution from user code. The leading open-source implementation, Hadoop is used daily at large companies such as Yahoo! and Facebook to process petabyte-scale data [3] [4].

Debugging MapReduce frameworks is a conventionally hard problem due to its massive scale and distributed nature. Often various forms of instrumentation have been used to debug MapReduce frameworks that include programmer-chosen debugging logs, MapReduce system logs, application traces, etc. Although these methods have various benefits, they are often highly dependent on programmer responsibility and often have a limited use in a distributed setting, as they often only provide information from the context of the application, node or environment they are running and cannot be effectively used in the use of debugging the overall state of the MapReduce framework.

Our work with system calls focuses on diagnosing performance issues that cause significant degradation in the system's overall performance. Specifically, we focus on diagnosing disk and network related issues that can affect MapReduce performance. Our work seeks to *explore and evaluate* the extent to which syscall-based instrumentation is useful in diagnosing these problems in MapReduce frameworks.

The contributions of this paper are – (1) a new approach that exploits system call instrumentation to automatically and transparently diagnose performance problems in MapReduce frameworks, (2) a statistical diagnosis algorithm that correlates system calls occurrences and timings to localize node(s) that is/are responsible for a performance problem, and (3) a semantic diagnosis algorithm that parses and correlates system call output from different nodes to identify a node that is responsible for a performance problem.

2 Background & Problem Statement

2.1 MapReduce Framework

A MapReduce job consists of two main abstractions, a Map task and a Reduce task that are specified by the programmer [10]. The Map task is first applied locally on each node on some segment of the input data, and its output is then acted upon by the Reduce task. The MapReduce framework splits the input dataset into smaller independent partitions, and creates multiple instances of Map and Reduce tasks to operate on each partition in parallel. Our work focuses on Hadoop, which is an open-source, Java implementation of MapReduce. Hadoop has a master-slave architecture, with a single master and multiple slave hosts. Hadoop consists of an execution layer which executes Map and Reduce tasks, and the Hadoop Distributed Filesystem (HDFS), which is a an implementation of the Google FileSystem [15].

2.2 Motivation

We propose the use of system calls (syscalls) as a novel way of debugging MapReduce frameworks, as we believe that *syscall event-streams present a rich source of statistical and semantic information for performance problem diagnosis in MapReduce frameworks*. Syscalls are preferred in MapReduce frameworks for the following reasons:

- i **High Reliability.** Syscalls in various architectures are essentially consistent and provide a uniform way of analyzing a given characteristic, as opposed to application traces or programmer-inspired debugging that can be variable and unreliable.
- ii **No dependence on Hardware Architecture.** Often debugging solutions assume a given architecture, which is not a safe assumption in MapReduce frameworks that themselves make no assumptions on the underlying system architecture.
- iii **Insight into underlying system information.** Syscalls involve switches to kernel space and can provide more information on kernel decisions, the file system, networking, threading, etc.

2.3 Goals

- i **Application-transparency.** There should be no modifications to current MapReduce programs.
- ii **Minimize false-positive rate.** Our approach should be able to correctly distinguish between anomalous behavior with a low rate of false-positives.
- iii **Problem Coverage.** Our approach aims to diagnose performance problems that involve network and disk related issues that result in degraded performance of the MapReduce instance.

2.4 Non-Goals

- i **Code-level debugging.** We do not aim to provide indicators of where software might be failing such as isolating responsible sections or lines of source code, but only seek to identify the culprit node and the performance problem it is facing in the MapReduce framework.
- ii **Optimized instrumentation overheads.** Our current implementation of collecting syscall instrumentation imposes significant monitoring overhead and we focus only on an evaluation of a proof-of-concept implementation.

2.5 Assumptions

- i **A majority of the MapReduce nodes exhibit fault-free behavior.**
- ii **All of the MapReduce nodes have identical hardware configurations, memory, network access, etc.** This is a reasonable assumption as most instances of MapReduce clusters are designed to have the same "technical specifications".
- iii **Time on each MapReduce node is synchronized.** We rely on this assumption, since we use time-based syscall instrumentation to correlate behaviour across nodes in the framework.

3 System Call Instrumentation

3.1 Tools

We are using the unix tool `strace` [12] to attain system call instrumentation. `strace` provides various utilities that are useful in attaining time-based and count-based syscall instrumentation on a running MapReduce instance. Specifically we use two modes:

`strace -cf`. This provides the following information about each monitored syscall: total number of invocations, number of failed invocations, average and total time spent in a syscall and the percentage of time spent in a syscall with respect to other monitored syscalls. `strace -fTttt`. This provides a time-based log that prints out syscall invocations with their arguments, which can be used to trace the control flow of the MapReduce instance in a distributed manner. It also prints the total time spent in each syscall, with the appropriate return codes. The `-f` flag in both modes of `strace` ensures that child processes spawned on a given node are also recursively monitored and profiled.

3.2 System Call Scope

We focus on a small set of syscalls that to our knowledge can be used in determining common performance issues in a MapReduce framework. We focus on 2 primary classes of syscalls - network (`accept()`, `connect()`, `bind()`, `socket()`) and filesystem related syscalls (`access()`, `stat()`). We also monitor `execve()`.

3.2.1 Network related syscalls

We only monitor the above system calls because they form the basis of TCP/IP network communication [2]. We do not monitor any data sent across the network because there is a high overhead in tracing these calls and typically these calls had variable invocation times based on the data that each system call was responsible for sending/receiving. As a result, it was hard to tell if a node was undergoing genuine network-related problems or transferring variable data.

3.2.2 Filesystem related syscalls

We believe that distributed filesystems check the information of a file before performing a read/write operation on the file. Like the network related calls, we do not monitor read and write syscalls, because their invocation times are dependent on the data they act on. Hence, we only monitor calls that check the information of a file, which are typically quick to return unless possible disk-related issues may cause them not to.

3.2.3 Other syscalls

We believe that `execve()` and its variants provide an idea of how a spawned task performs on the overall in the MapReduce framework, and as a result can be useful in providing information on any underperforming nodes.

4 Approach

4.1 Predefined Analysis and Thresholds

Both the statistical and semantic algorithms, rely on predefined analysis of a workload, which we refer to as the process of profiling the steady, average state of the system. To do this, we run a series of control experiments for each workload and record the following information: All of `strace -cf` information and a record of every unique system call invocation with corresponding arguments, the return code and the time taken for the method to return. We call this time the **syscall invocation time**. Based on this information we do the following:

4.1.1 Statistical Diagnosis

For each run and syscall on a node, we construct a histogram of the number of syscall invocations for each invocation time bucket. Essentially, we bucket the system calls by their invocation times and present a histogram that has the number of such invocations for each bucket. We also store the total time spent in each system call on each node, which should equal the total area of the histogram referred to above. Across runs, we mantain and update the minimum and maximum counts for each time bucket and the total time spent in each system call. It is these minima and maxima for both histograms' buckets and overall time for a system call that constitute the lower and upper thresholds for admissible Hadoop behavior. Theoretically with sufficient runs, any behavior outside these minimum and maximum thresholds indicate the possibility of anomalous behavior.

4.1.2 Semantic Diagnosis

For each control run, we store the time taken for each unique system call invocation for each node and over many runs update the minimum and maximum times. It is these minima and maxima that constitute the thresholds for non-anomalous Hadoop beahvior in the semantic algorithm. This way when we semantically parse verbose syscall logs, a system call invocation with matching parameters and return codes that has a timing that is outside these admissible thresholds, could indicate the possibilty of a performance problem.

4.2 Statistical System Call Based Diagnosis

In this approach, we build a histogram of the number of invocations of a given syscall for different invocation time buckets for that syscall (like above) and also store the total time spent in each syscall for each node. We then compare this histogram and total time values to that constructed in the predefined analysis above and use the comparisons to automatically detect possible performance problems. We base our algorithm on the hypotheses highlighted above. Our algorithm in rough pseudocode is:

```
1. disk-hog detection: examine the following stat() information across all nodes:
   (a) $max \leftarrow$ node with the highest total time spent in stat(). $stat_{max} \leftarrow$ total time in stat() for max.
   (b) if $stat_{max}$ is outside the total time stat() threshold then: compute pairwise percentage differences between the total timings of stat() across all nodes. if percentage differences between all other nodes and max is greater than 25% and the total timing in access() on max is also outside threshold then: we have disk-hog. goto Node Isolation.
2. network-hog detection: examine the following connect() information across all nodes.
   (a) $max \leftarrow$ node with the highest total time spent in connect(). $connect_{max} \leftarrow$ total time in connect() for max.
   (b) if $connect_{max}$ is outside the total time connect() threshold then: compute pairwise percentage differences between the total timings of connect() across all nodes. if percentage differences between all other nodes and max is greater than 25% then: we have network-hog. goto Node Isolation.
3. Node Isolation: examine the following information across all nodes.
   (a) foreach system call $s$ in (bind,socket,execve) do: For every syscall bucket value that is outside the threshold for that bucket, add the difference to $scr_s$. Compute pairwise percentage comparison between $scr_s$ values across all nodes if any node has more than a 50% difference then: mark that node as "possibly faulty".
```

4. Examine total time spent in `accept()` across all nodes. `if` there exist values that are outside thresholds of `accept()` timings `and` we did not arrive here from a possible *network-hog* `then:` mark that node as "possibly faulty".
5. return the set of all nodes that are marked *max* and "possibly faulty".

4.3 Semantic Syscall Call Based Diagnosis

In this approach, we consider syscall invocations as events in time stream and parse `strace` output for each syscall and try to isolate spikes. The algorithm bases itself on scores *scr* for each pairwise node combination and semantic parsing of `strace` logs. We take each sycall invocation and check if it matches (same syscall, arguments and return code) any syscall stored in the predefined analysis. If a match exists and the invocation time is outside the admissible thresholds for that syscall invocation, we mark that node as "possibly faulty". If the error involves filesystem calls, we report a *disk-hog*. If we see out-of-threshold `connect` values we report a *network-hog*. In the case of network syscalls we try to parse parameter information when available to see which node the current node is trying to communicate with. In the case the system call invocation is out-of-threshold, we increment the corresponding node pair's *scr* value. At the end, we compare the *scr* values across all pairs and isolate a node as "faulty" if all the highest *scr* values involve that node with a different node. If we are unable to satisfy this condition or we cannot find a possible hog, we do not diagnose anything. This algorithm aims to isolate the single node that is undergoing a performance problem.

5 Experimental Setup

We perform our experiments and instrumentation on a cluster of 5 identical machines running Hadoop 0.20.1. Each node consists of an AMD Opeteron 1220 dual-core CPU with 4GB memory, Gigabit Ethernet, dedicated 320GB disk for Hadoop and runs the amd64 version Debian/GNU Linux 4.0. The machines run in stock configuration and with no background tasks. The results we report are based on Hadoop MapReduce jobs being run on these 5 machines, with a single master node and 4 slave nodes. These experiments are designed to run in approximately at most 20 minutes. To ensure a consistent experimental setup, we reboot and reinitialize Hadoop settings on each machine before each experiment. From here we run a given workload and either run a control run (no fault injection) or inject a given fault. Once the workload begins, we monitor the syscall activity with `strace` and record the local output of `strace` on each node. After the workload is finished, we run into the diagnosis phase if a fault was injected. Here we analyze all the logs from each node and make an assertion on which node is a culprit node in the setup, and diagnose it as a problematic node with a given cause.

5.1 Workloads

We run one of two possible MapReduce workloads:

i *wc* : a naive wordcount program that outputs the total number of occurrences of all the words in a given text corpus of 100,000 words.

ii *sort* : a naive program that sorts a randomly generated set of 100,000 integers.

Additionally, we run each workload an equal number of times in both speculative and non-speculative execution. Hadoop attempts to realize performance problems in its nodes under speculative execution [19]. In this mode of execution, Hadoop makes an effort to reduce the impact of underperforming nodes by scheduling redundant copies of a given Map task on various nodes, and picking the fastest node's result for that computation.

5.2 Performance Problems Injection

When we run the workloads above they run in two possible modes, either as a control experiment or as an instance that has a fault injected into it. If we choose to execute a fault-injected run of a workload, we inject a fault into a predetermined node at a specific time and for a fixed duration of 360s during the workload. Specifically these faults are:

i *disk-hog.* Write 2GB chunks continually to the disk and "hog" access to the disk.

ii *network-hog.* Drop 5%, 20% and 50% of the packets in a network stream.

To handle the fast-growing and verbose output of `strace`, we have designed a framework that is able to run `strace` on all the nodes in MapReduce cluster and synchronize this information across the nodes. After the experiment terminates, we collect the log information in an automated manner and diagnose a possible bug or problem on a specific node in the MapReduce instance, by analyzing the syscall instrumentation.

6 Results & Discussion

6.1 Terminology

We define the *output node set* as all the nodes that are outputted by our agorithms (and believed to be faulty). We define the *diagnosis success* or *true-positive rate* as the ratio of runs where a fault-injected node was correctly identified in the *problem node set* of size at most 3 with the right cause over that of all runs for that workload with that fault. If the *output node set* is non-empty and does not contain the fault-injected node or we provide a wrong cause, we contribute this run to a false positive run and define *false positive* to be the ratio of false positive runs over all fault-injected runs for a particular fault.

	Diagnosis Success	False Positive
Speculative	0.88	0.13
Non-Speculative	0.88	0.13

Table 1: Statistical Diagnosis for *disk-hog*

6.2 Statistical Syscall Based Diagnosis

We realize that for *disk-hog* behaviour we have the following characteristics for nodes in the *output node set*:

i **Majority of process' time is spent in a `stat()` syscall.** Since the node is experiencing a disk-hog, it takes it a much larger time to access file information for files.

ii **The average time spent in a `stat()` and `access` call is significantly higher than that of other nodes.** Since disk-hog is affecting a common distributed filesystem, the HDFS, we see that it should take a longer time for file information to be accessed across nodes affected by the disk hog.

iii **With high occurrence, a much smaller chunk of the process' time is spent in the `execve()` call as compared to that of other non-faulty nodes.** A large file-related time reduces the percentage of time spent in other non-file related calls such as `execve()`

We realize that for *network-hog* behaviour we have the following characteristics for a node in the *output node set*:

i **A significant increase in `connect()` latency.** Since the faulty node is experiencing a network-hog, it follows that there should be an increased latency for making an end-to-end connection with another listening/broadcasting network port.

ii **A significant drop in `accept()` times as compared to that of other nodes.** Non-faulty nodes can accept connections normally, however because the faulty node is experiencing a network-hog, the faulty node cannot validate and accept a connection as fast as non-faulty nodes and often returns quickly as an error.

iii **More marked exhibition of the above behaviour at higher network-hog levels.** This intuitively follows since the greater we hog the network, the more apparent the effects it would have on the syscalls responsible on handling network-related information and flow.

6.3 Semantic Syscall Based Diagnosis

We summarize our results for *disk-hog* in Table 2. For *network-hog*(Figure 2), we realize that at smaller network-hog levels, it becomes semantically harder to differentiate genuine performance problems due to network-hog as opposed to inherent network latency that is possible in normal execution. We realized that at lower levels, the "spikes" we were seeking were admissible in the context of normal execution, and as a result this reduced the effectiveness of our semantic diagnosis. On higher network-hog levels, the "spikes" due to network hog were more apparent and lasted consistently enough for our diagnosis algorithm to identify culprit nodes. Under speculative execution, the Hadoop framework reduces the effective underperformance on a culprit node, by reducing the overall execution on that node. As a result, it became harder to realize faults on a node under speculative execution.

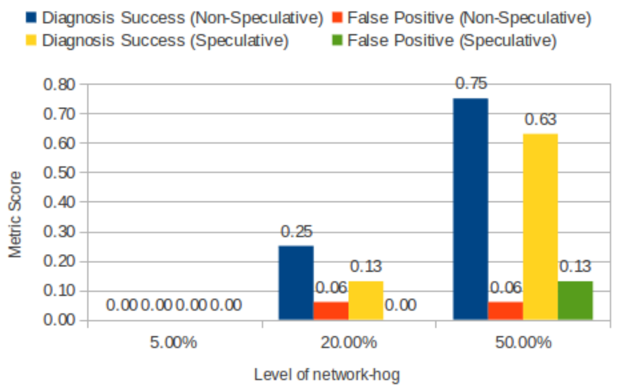

Figure 1: Plot of various metrics against *network-hog* levels for statistical diagnosis.

	Diagnosis Success	False Positive
Speculative	0.63	0.00
Non-Speculative	0.69	0.06

Table 2: Semantic Diagnosis for *disk-hog*

6.4 Execution Models

We realize the following:

i **Statistical diagnosis is more effective on non-speculative execution as compared to speculative execution workloads.** Since Hadoop makes a speculative effort to reduce the load on what it thinks is an underperforming node, by scheduling redundant copies of a task on various nodes, and choosing the first successful completion, it reduces the effectiveness of our semantic diagnosis algorithm. Since an underperforming node will traditionally run for a much longer period and get a signal to abandon its computation after another node successfully computes the same result, its period of poor performance becomes less significant over the running time of the workload. As a result, the performance problem on a given culprit node become increasingly semantically insignificant, and it becomes harder to realize and diagnose.

ii **Semantic diagnosis is less effective than statistical diagnosis under speculative execution.** Since

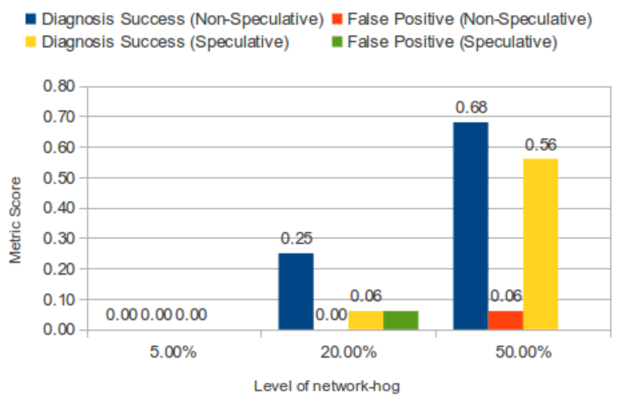

Figure 2: Plot of various metrics against *network-hog* levels for semantic diagnosis.

speculative execution prevents further execution of a given task on an underperforming node, our semantic algorithm is less exposed to faulty behavior and as a result this makes it harder to realize a performance problem. From a statistical point of view, the underperforming node consistently fails across various Map tasks and as a result can be statistically isolated as a probable underperforming node in a run of many Map tasks in a given workload. More importantly, our semantic algorithm focuses strictly on isolating only 1 node that is responsible for the performance problem. In the statistic case, a set of nodes are sometimes returned, which is much easier to ascertain in any runtime environment. Because of this qualitative difference, it is easier for the statistic algorithm to identify at least the node which is undergoing a performance problem.

6.5 Workload Independence

We realized that in all workload runs under various settings, *the performance issues largely affected the framework, and the diagnosis success was independent of the workload run*. Since the Hadoop framework is a complex, distributed framework, we hypothesize that a large amount of the system calls being traced were attributed to calls to initialize, run and maintain the Hadoop framework. Since the performance issues we tested were generic performance problems that are common to most distributed systems, we can possibly extend this approach to other complex distributed frameworks. Firstly, this finding implies that we can diagnose a greater variety of problems on various workloads with a relatively similar diagnosis success since a large amount of the system calls are attributed to the Hadoop framework, and we can analyze the changes in the system call behavior in the framework itself to diagnose generic performance problems that can apply to a wide variety of workloads. Secondly, this implies that further work must be directed into exploring Hadoop-specific bugs such as faulty/slow Map and Reduce keys and evaluating its associated system call instrumentation.

7 Related Work

7.1 Diagnosis for MapReduce frameworks

X-Trace [7] was used to instrument Hadoop and provided fine-grained trace events and a summarized views as a form of instrumentation. Additionally, [18] [13] [1] focused on mining various log/error tracing methods in addition to supervised learning to aid in problem diangosis. State machine views of logs and visualization tools on top of this analysis was explored by [17] [11] [16]. These approaches use many other forms of instrumentation, but our approach to our knowledge is the first syscall based performance debugging method that identifies the root-cause of a performance problem in MapReduce frameworks.

7.2 System Call Based Diagnosis

[5] looked at developing a reliable system view based on system call instrumentation to prevent and detect any security issues. [6] [9] use sequences of system calls to isolate atypical program execution. Forensix [8] uses system calls to capture system call timing and parameters to infer possible security flaws or incidents. [14] focused on a similar use of syscalls to debug the PVFS framework, but our approach differs from that of both Forensix's and [14], as we deal with a qualitatively different framework from PVFS or do not concern ourselves with security. Additionally, we do not use the error-based semantic correlation approach highlighted in [14] or log security concerns as in Forensix. To the best of our knowledge, we do not know any other approach that uses system calls to diagnose performance problems in MapReduce frameworks.

8 Conclusion & Future Work

We have presented a novel way of debugging performance problems in MapReduce frameworks using system call instrumentation. We realize that this a relatively effective way to diagnose performance problems, with greatest effect in non-speculative environments. Our approach resulted in the greatest success when isolating disk related performance problems. In the future, we aim to reduce strace's runtime overhead with a custom syscall tracer. We are also presently evaluating our diagnosis apporach on a 20-node cluster with real-world workloads, with a focus on diagnosing Hadoop-specific bugs.

9 Acknowledgements

This research was sponsored in part by the project CMUPT/RNQ/0015/2009 (TRONE—Trustworthy and Resilient Operations in a Network Environment), and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC).

References

[1] The Blind Men and the Elephant: Piecing Together Hadoop for Diagnosis. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*. Mysuru, India, Nov 2009.

[2] ANUPAMA, B. Know your TCP system call sequences. `http://www.ibm.com/developerworks/aix/library/au-tcpsystemcalls/index.html`, 2012.

[3] APACHE SOFTWARE FOUNDATION. Hadoop. `http://hadoop.apache.org/core`, 2011.

[4] APACHE SOFTWARE FOUNDATION. Powered by Hadoop. `http://wiki.apache.org/hadoop/`, 2011.

[5] CROSBIE, M.J. AND KUPERMAN, B.A. A building block approach to intrusion detection. In *Recent Advances in Intrusion Detection (RAID)* (2001).

[6] ESKIN, E. AND LEE, W. AND STOLFO, S.J. Modeling system calls for intrusion detection with dynamic window sizes. In *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01. Proceedings* (2001), vol. 1, IEEE, pp. 165–175.

[7] FONSECA, R. AND PORTER, G. AND KATZ, R.H. AND SHENKER, S. AND STOICA, I. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation* (2007), USENIX Association, pp. 20–20.

[8] GOEL, A. AND FENG, W.C. AND MAIER, D. AND WALPOLE, J. Forensix: A robust, high-performance reconstruction system. In *Distributed Computing Systems Workshops, 2005. 25th IEEE International Conference on* (2005), IEEE, pp. 155–162.

[9] HOFMEYR, S.A. AND FORREST, S. AND SOMAYAJI, A. Intrusion detection using sequences of system calls. *Journal of computer security 6*, 3 (1998), 151–180.

[10] 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.

[11] JIAQI TAN, KAVULYA. S, GANDHI. R AND NARASIMHAN. P,. Visual, Log-Based Causal Tracing for Performance Debugging of MapReduce Systems. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference*. Mysuru, India, 2009.

[12] LINUX MANUAL PAGES. strace(1) - Linux man page. `http://linux.die.net/man/1/strace`, 2012.

[13] LOU, J.G. AND FU, Q. AND WANG, Y. AND LI, J. Mining dependency in distributed systems through unstructured logs analysis. *ACM SIGOPS Operating Systems Review 44*, 1 (2010), 91–96.

[14] MICHAEL P. KASICK AND KEITH A. BARE AND EUGENE E. MARINELLI III AND JIAQI TAN AND RAJEEV GANDHI AND PRIYA NARASIMHAN. System-Call Based Problem Diagnosis for PVFS. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability* (Lisbon, Portugal, June 2009).

[15] S. GHEMAWAT AND S. LEUNG. The Google file system. In *In ACM Symposium on Operating Systems Principles*. Lake George, NY, Oct 2003, pp. 29–43.

[16] TAN, J. AND PAN, X. AND KAVULYA, S. AND GANDHI, R. AND NARASIMHAN, P. Mochi: visual log-analysis based tools for debugging hadoop. In *Proceedings of the 2009 conference on Hot topics in cloud computing* (2009), USENIX Association, p. 18.

[17] TAN, JIAQI AND PAN, XINGHAO AND KAVULYA, SOILA AND GANDHI, RAJEEV AND NARASIMHAN, PRIYA. SALSA: analyzing logs as state machines. In *Proceedings of the First USENIX conference on Analysis of system logs* (Berkeley, CA, USA, 2008), WASL'08, USENIX Association, pp. 6–6.

[18] XU, W. AND HUANG, L. AND FOX, A. AND PATTERSON, D. AND JORDAN, M.I. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 117–132.

[19] YAHOO DEVELOPER NETWORK. Speculative Execution. `http://developer.yahoo.com/hadoop/tutorial/module4.html`, 2012.