Transparent System Call Based Performance Debugging for Cloud Computing

Nikhil Khadke and Prof. Priya Narasimhan
School Of Computer Science Senior Thesis - Working Thesis Draft, with certain parts incomplete
nkhadke@andrew.cmu.edu, priya@cs.cmu.edu

ABSTRACT

Problem Diagnosis and debugging in concurrent environments such as the cloud and popular distributed systems frameworks has been a traditionally hard problem. We explore an evaluation of a novel way of debugging distributed systems frameworks by using system calls. We focus on Google's MapReduce framework, which 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. 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 of the debugging these frameworks using the lowest 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.

I. 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 [1]. The MapReduce framework allows programmers to easily process large data, by abstracting away low-level details of distributed execution from user code and as a result, the framework has gained enormous popularity both in academia and the industry. The leading open-source implementation, Hadoop is used daily at large companies such as Yahoo! and Facebook to process petabyte-scale data [2] [3].

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. Although, logs, traces and its variants may provide a verbose description of the state of a current node in MapReduce frameworks, 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.

Another issue that makes MapReduce profiling difficult is the nonhomogeneous performance of MapReduce nodes. Unlike conventional distributed frameworks where most nodes in the framework can be thought of as replicas, MapReduce nodes make no guarantee on being identical replicas. Since we do not know the scheduling of tasks on nodes, it is inevitable that certain nodes may be used or accessed more often than others. Some reasons for this could be the locality of accesses to a certain node, a shorter network latency from the master node or Namenode or the nature of a type of MapReduce job. This node asymmetry is further complicated because MapReduce follows a master-slave architecture and as a result, master nodes are qualitatively different from slave nodes. Since our

black-box debugging technique has no guarantee on being provided with information on which nodes are masters or slaves, it becomes even harder to accurately localize a fault to a given node.

Of the most interesting and relevant problems to localize in such systems are errors that do not cause an outright "crash" in the system, but cause significant degradation in the system's overall performance. Our work with system calls targets problem diagnosis in the distributed MapReduce framework used for high performance computing (HPC), and focuses on diagnosing any performance issues that might occur within the system. 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 a node that is 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.

II. BACKGROUND & PROBLEM STATEMENT

A. MapReduce Framework

A MapReduce job consists of two main abstractions, a Map task and a Reduce task that are specified by the programmer. 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. Another intermediate phase before the Reduce phase is the Shuffle phase that is transparently managed by the MapReduce framework. This phase sorts the output of the Map phase and feeds it to respective Reduce tasks. Our work focuses on Hadoop, which is an open-source, Java implementation of MapReduce.

B. Hadoop Architecture

Hadoop has a master-slave architecture as indicated in Fig. 1, 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. The master node runs the JobTracker daemon, which is responsible for scheduling task execution on slaves and implements fault-tolerance using heartbeats sent to slaves. Another daemon, the NameNode provides the namespace for HDFS in the framework. The TaskTracker daemon, is run by each slave host and is responsible for executing Map and Reduce tasks locally on each node. Additionally. the DataNode daemon, stores and serves data blocks for HDFS. In Hadoop, each daemon is a Java process, and locally generates logs which record error traces and system execution events such as start and end of the Map and Reduce phases.

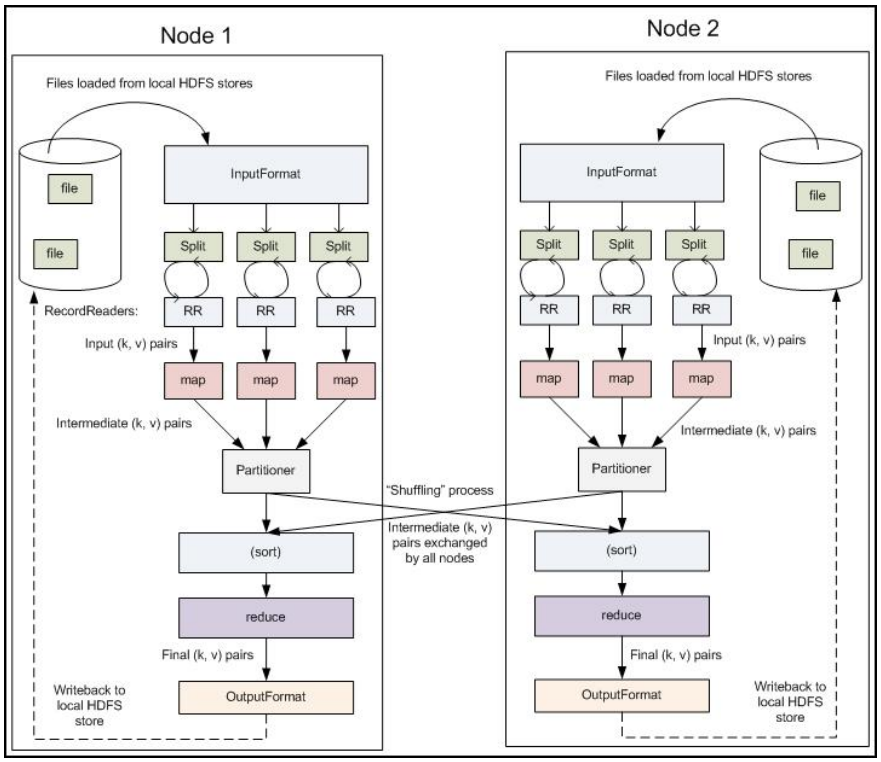

Fig. 1 : Hadoop Architecture

C. Motivation

We propose the use of system-calls 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 problem diagnosis* in MapReduce frameworks. Syscalls are preferred in MapReduce frameworks for the following reasons:

(i) *High Reliability*. Syscalls in various architectures are essentially always 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) *Low Overhead*. Syscalls tracing can be much cheaper than other alternatives of debugging.

(iii) *No dependence on Hardware Architecture*. Often many 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 of the cluster/system.

(iv) *Deep insight into underlying system information*. Syscalls involve switches to kernel space and can provide arguably superior information on kernel decisions, the involved file system, networking, threading, etc. Such rich insight is rarely achieved in alternative debugging solutions that do not leverage the use of system calls.

D. Goals

(i) *Application-transparency.* There should be no modifications to current MapReduce programs or software. Our approach should ensure it is independent of the underlying MapReduce software or operation.

(ii) *Minimize false-positive rate.* Our approach should be able to correctly distinguish between anomalous behaviour with a low rate of false-positives.

(iii) *Problem Coverage*. Our approach aims to diagnose performance problems, system misconfigurations, resource exhaustion or degradation and terminal errors that result in the termination of the MapReduce instance.

E. Non-Goals

(i) *Code-level debugging.* We do not aim to provide indicators of where software might be failing, but only seek to identify the culprit node in the MapReduce framework that is experiencing problems.

(ii) *Optimized instrumentation overheads.* Our current implementation of syscall-instrumentation imposes significant monitoring overhead on I/O in the MapReduce system and this paper focuses only on an *evaluation of a proof-of-concept* implementation that can provide automated performance debugging in MapReduce frameworks.

F. Assumptions

(i) *A majority of the MapReduce nodes exhibit fault-free behaviour.*

(ii) *All of the MapReduce nodes have identical hardware configurations, memory, network access, etc.* This is not an unreasonable or unrealistic assumption as most instances of MapReduce clusters are designed to have the same "technical specifications", as varying specifications can only contribute to a bottleneck in the overall performance of the system.

(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 MapReduce framework.

III. SYSTEM CALL INSTRUMENTATION

A. Tools

We are using the unix tool `strace` to attain syscall instrumentation. The tool `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 of `strace`:

(i) `strace -cf`. This provides the following information --

i. Total calls made to a each syscall in a set of chosen syscalls
ii. Average time spent in each syscall
iii. Total time spent in each syscall
iv. Number of failed syscall invocations for each syscall
v. Percentage of time of spent in the syscall with respect to other monitored syscalls

A sample output is shown below in Fig 2, which shows part of a strace -cf output file, that specifically shows information for the first two nodes in the MapReduce framework running the *wc* workload.

```
nkhadke@fp38:~/data/final_dh$ cat strace_20120328_4.logs
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 96.15    0.001524         117        13         7 execve
  3.85    0.000061           0      1705      1198 stat
  0.00    0.000000           0        74        44 access
  0.00    0.000000           0        72         1 socket
  0.00    0.000000           0        23        11 connect
  0.00    0.000000           0         1           accept
  0.00    0.000000           0        47        45 bind
------ ----------- ----------- --------- --------- ----------------
100.00    0.001585                  1935      1306 total
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 88.19   30.336849        6108      4967      3257 stat
  4.15    1.427172        5532       258       144 access
  3.12    1.073599        6353       169         2 socket
  1.92    0.659200        6400       103       101 bind
  0.11    0.038400        6400         6         2 accept
  1.38    0.475994        6025        79        53 execve
  1.13    0.388799        6271        62        35 connect
------ ----------- ----------- --------- --------- ----------------
100.00   34.400013                  5644      3594 total
```

Fig. 2 : Sample strace -cf output

(ii) `strace -fttt`. 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.

A sample output is shown below in Fig 3, which shows part of a `strace -fttt` output file for a given node in the MapReduce framework running the *wc* workload.

```
nkhadke@fp38:~$ head -n20 strace.test
1826  1332614360.922431 execve("/grid0/sw/jdk1.7.0/jre/bin/java", ["/grid0/sw/jdk1.7.0/jre/bin/java", "-Xmx1000m", "-Dhadoop.log.dir=/grid0/sw/hadoo"..., "-D
hadoop.log.file=hadoop.log", "-Dhadoop.home.dir=/grid0/sw/hado"..., "-Dhadoop.id.str=", "-Dhadoop.root.logger=INFO,consol"..., "-Dhadoop.policy.file=hadoop-p
oli"..., "-classpath", "/grid0/sw/hadoop-0.20.1-dev/conf"..., "org.apache.hadoop.fs.FsShell", "-put", "/h/nkhadke/data/corpus_tiny", "/user/nkhadke/input/"],
 [/* 25 vars */]) = 0 <0.000146>
1826  1332614360.922826 access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory) <0.000011>
1826  1332614360.922975 access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory) <0.000011>
1826  1332614360.923045 stat("/grid0/sw/jdk1.7.0/jre/bin/../lib/amd64/jli/tls/x86_64", 0x7fff1e443d50) = -1 ENOENT (No such file or directory) <0.000013>
1826  1332614360.923114 stat("/grid0/sw/jdk1.7.0/jre/bin/../lib/amd64/jli/tls", 0x7fff1e443d50) = -1 ENOENT (No such file or directory) <0.000012>
1826  1332614360.923177 stat("/grid0/sw/jdk1.7.0/jre/bin/../lib/amd64/jli/x86_64", 0x7fff1e443d50) = -1 ENOENT (No such file or directory) <0.000011>
1826  1332614360.923242 stat("/grid0/sw/jdk1.7.0/jre/bin/../lib/amd64/jli", {st_mode=S_IFDIR|0755, st_size=16, ...}) = 0 <0.000012>
1826  1332614360.923335 stat("/grid0/sw/jdk1.7.0/jre/bin/../jre/lib/amd64/jli/tls/x86_64", 0x7fff1e443d50) = -1 ENOENT (No such file or directory) <0.000011>
1826  1332614360.923398 stat("/grid0/sw/jdk1.7.0/jre/bin/../jre/lib/amd64/jli/tls", 0x7fff1e443d50) = -1 ENOENT (No such file or directory) <0.000011>
1826  1332614360.923461 stat("/grid0/sw/jdk1.7.0/jre/bin/../jre/lib/amd64/jli/x86_64", 0x7fff1e443d50) = -1 ENOENT (No such file or directory) <0.000011>
1826  1332614360.923522 stat("/grid0/sw/jdk1.7.0/jre/bin/../jre/lib/amd64/jli", 0x7fff1e443d50) = -1 ENOENT (No such file or directory) <0.000012>
1826  1332614360.923644 access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory) <0.000010>
1826  1332614360.924053 access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory) <0.000010>
1826  1332614360.924297 access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory) <0.000010>
1826  1332614360.925142 access("/grid0/sw/jdk1.7.0/jre/lib/amd64/libjava.so", F_OK) = 0 <0.000013>
1826  1332614360.925372 stat("/grid0/sw/jdk1.7.0/jre/lib/amd64/server/libjvm.so", {st_mode=S_IFREG|0755, st_size=9770563, ...}) = 0 <0.000014>
1826  1332614360.925520 execve("/grid0/sw/jdk1.7.0/jre/bin/java", ["/grid0/sw/jdk1.7.0/jre/bin/java", "-Xmx1000m", "-Dhadoop.log.dir=/grid0/sw/hadoo"..., "-D
hadoop.log.file=hadoop.log", "-Dhadoop.home.dir=/grid0/sw/hado"..., "-Dhadoop.id.str=", "-Dhadoop.root.logger=INFO,consol"..., "-Dhadoop.policy.file=hadoop-p
oli"..., "-classpath", "/grid0/sw/hadoop-0.20.1-dev/conf"..., "org.apache.hadoop.fs.FsShell", "-put", "/h/nkhadke/data/corpus_tiny", "/user/nkhadke/input/"],
 [/* 26 vars */]) = 0 <0.000215>
```

Fig. 3 : Sample strace -fttt output

The -f flag in both modes of `strace` ensures that child processes spawned on a given node are also recursively monitored and profiled.

B. System call scope

We focus on a small set of syscalls that to our knowledge can be used in determining common performance and

erroneous issues in a MapReduce framework. We focus on 2 primary classes of syscalls - network and filesystem related syscalls. We also monitor the syscall `execve()` and its other variants.

Network related syscalls:

- `accept()` (accepts a network request to connect)
- `connect()` (sends a request to connect to a port)
- `bind()` (adds an address to a given connection)
- `socket()` (creates a connection socket to listen or broadcast across the network)

Filesystem related syscalls:

- `access()` (checks if a file can be accessed by a process)
- `stat()` (gets information on a file)

Other syscalls:

- `execve()` (runs a process)

C. Justification for choice of System-calls

Network related syscalls:
We believe that the above syscalls form the basis of networking at a low level and performance of network communication can be estimated by analyzing the performance of these syscalls. For example, `socket()` gives us an estimate of the number of valid connections that are currently open, while the remaining syscalls give us a better understanding of the behaviour of network requests and responses in the MapReduce framework.

Filesystem related syscalls:
We believe that any filesystem access much use the following two methods at least once and as a result paint a good picture of how the filesystem is used and how well it performs.

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 any underperforming or terminal-error prone error nodes.

IV METHODOLOGY

A. Experimental Setup

We perform our experiments and instrumentation on a cluster of <Specific tech specs to filled in>. The machines run in stock configuration and with no background tasks. The results we report are based on Hadoop MapReduce jobs being run on 6 machines, with a single master node and 5 slave nodes. Additionally, we restrict the number of maximum maps to be 4 on each node and the number of maximum reduce tasks as 4 on each node. However, Hadoop's architecture ultimately decides the number of map tasks based on the input the MapReduce job acts on. Experimentally, for our workloads this ends up running 2 map tasks on each node and 1 reduce task on each node.

To ensure a consistent experimental setup, we reboot each machine immediately prior to the start of each experiment. This reboot process in detail is as follows:

1. Each node is a rebooted
2. Time synchronization is performed at boot-time using `ntpdate`
3. NFS file daemons are restarted on each node

4. NameNodes in each Hadoop node are formatted
5. A data transfer of the relevant files into appropriate HDFS directories is performed
6. A source sync is performed
7. Each node is put to sleep for 30 seconds

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.

B. Workloads:

We run one of two possible workloads --

(i) *wc* : this workload is a naive MapReduce wordcount program that outputs the total number of occurrences of all the words in a given text corpus. Abstractly, each Map task simply outputs each word *w* it sees as a key-value pair, (w,1). The Reduce phase performs a sum reduction for each unique word and outputs the result. We run this workload on a relatively large corpus of 100000 words.

(ii) *sort* : this workload is a naive MapReduce program that sorts 100000 integers and outputs the result. Abstractly, each Map and Reduce task run as identity functions that simply output their input. We rely on the ShuffleSort phase to sort the numbers. We run this workload on a randomly generated set of 100000 numbers.

Additionally, we run each workload 10 times in both speculative and non-speculative execution. We discuss the results of our algorithm on these workloads in section V.

C. 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 (360s in this setup) during the workload. Specifically these faults are -

(i) *disk-hog.* In our setup we write 2GB chunks continually to the disk and "hog" access to the disk for a fixed time period.

(ii) *network-hog.* In our setup we perform three levels of network-hog, by dropping 5%, 20% and 50% of the packets in a network stream for a fixed time period.

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.

D. Statistical Syscall-based diagnosis

In this approach, we build a histogram of a syscall counts for a given syscall for its average duration. We then use some predefined analysis to automatically distinguish any anomalous behaviour. We go into detail into predefined

analysis later. In short, we hypothesize syscall behaviour in an anomalous setting and see if a given node fits that behaviour, and if so flag it as a culprit node. This analysis essentially considers a histogram of each syscall (number of invocations that have a certain time) as a Probability Density Function (PDF) and checks if a given node's syscall instrumentation passes a given threshold and/or fits a certain behaviour. This then allows our method to indicate a certain type of failure or problematic performance in a given node. Based on these indicators, if we realize that a certain predetermined threshold is exceeded in these tests, then we can diagnose a node as being faulty and provide its cause.

E. Semantic Syscall-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 (sudden increases of a given syscall, based on predetermined threshold). Using the `strace` output from various nodes, we try to isolate the cause of a problem, by correlating which nodes are involved in that spike and trace the control flow to declare a given node as faulty if it fits the characteristics of a given fault.

F. Predefined Analysis

Both the algorithms in section D and E, rely on predefined analysis, 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 generate for each syscall a histogram that plots the number of invocations of a given syscall for a given duration. We then seek the "average" of these histograms from each run, and use this to define the steady state of a given syscall. Specifically, we use the averages, minimum and maximum for each syscall of the information below as predefined thresholds for non-anomalous behaviour. For each workload run, we record the following information -

i. Percentage of time spent in all invocations of a given syscall with respect to other monitored syscalls.
ii. Total time spent in all invocations of a given syscall.
iii. Average time spent in a given syscall.
iv. Number of total invocations of a given syscall.
v. Number of failed invocations of a given syscall.

G. Speculative Execution

Hadoop attempts to realize performance problems in its nodes under speculative execution. 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. This way, if a given node is underperforming and could potentially be a bottleneck to the overall performance of the framework, another node can run the task normally in its place quicker, and prevent the potential bottleneck. The speculative execution process ensures that when the copy of a task finishes successfully first, a message is sent to the JobTracker to stop and abandon the execution of the other copies of this task on other nodes and choose this copy as the definitive output for this replicated task.

In our setup we run an equal amount of workloads in both speculative and non-speculative execution and evaluate the performance of our diagnosis algorithms on them.

V. RESULTS & DISCUSSION

A. Terminology

We define *diagnosis success* as the ratio of runs where a fault-injected node was correctly identified with the right

cause over that of all fault-injected runs for a particular fault. If we incorrectly identify a node or 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.

B. Results: Statistical Syscall-based diagnosis

(i) *disk-hog*

We realize that for *disk-hog* behaviour we have the following characteristics for a faulty node that is experiencing a significant *disk-hog* load:

- *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. Since MapReduce files are allocated in blocks of at least 64MB, most of these files are stored on disk, and a *disk-hog* will significantly increase the time needed to access accurate information on the state of the file.
- T*he average time spent in a* `stat()` *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.
- *With high occurrence, a much smaller chunk of the process' time is is spent in the* `execve()` *call as compared to that of other nodes.* Since a large amount of time is spent in accessing file information, we hypothesize that either the Hadoop schedulers schedules a task on other nodes till the the task can be run normally, or that a large file-related time reduces the percentage of time spent in other non-file related calls such as `execve()`
- *Occasional very high* `access()` *times.* Permissions for accessing a file can be reduced when the node is experiencing a disk-hog in a distributed file system, where other nodes are also possibly accessing the same file (if it is not replicated sufficiently on other nodes).

We were able to diagnose disk-hog faults with an diagnosis success of **1.00** for both speculative and non-speculative execution. Our false positive rate was **0.00** for both cases. Hence our overall diagnosis success is **1.00** with a false positive rate of **0.00**.

(ii) *network-hog*

We realize that for *network-hog* behaviour we have the following characteristics for a faulty node that is experiencing a significant *network-hog* load:

- *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.
- *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*, it cannot validate and accept a connection as fast as non-faulty nodes.
- *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.

We summarize our results below:

Network-hog level	False positive rate	Diagnosis Success (Speculative)	Diagnosis Success (Non-Speculative)	Diagnosis Success (Overall)
5%	0.00	0.63	0.89	0.76
20%	0.00	0.74	0.94	0.84
50%	0.00	1.00	1.00	1.00

C. Results: Semantic Syscall-based diagnosis

(i) *disk-hog*

We were able to diagnose disk-hog faults with an diagnosis success of **1.00** for speculative and non-speculative execution. Our false positive rate was **0.00** for both cases. Hence our overall diagnosis success is **1.00** with a false positive rate of **0.00**.

(ii) network-hog

Network-hog level	False positive rate	Diagnosis Success (Speculative)	Diagnosis Success (Non-Speculative)	Diagnosis Success (Overall)
5%	0.00	0.17	0.05	0.11
20%	0.00	0.24	0.44	0.34
50%	0.00	0.68	1.00	0.84

We realize that at smaller network-hog levels, it becomes semantically to differentiate genuine performance problems due to network-hog as opposed to inherent network latency that is possible in normal execution. We realized that 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. This improved in speculative execution, as were able to realize a sudden reduction of activity on faulty nodes consistently and had a better chance to diagnose the culprit node. 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.

D. Discussion: Speculative and Non-Speculative execution

Hadoop can run with an additional setting - speculative execution. Speculative execution is Hadoop’s way of isolating struggling nodes as a workload progresses and recognize if a node might be undergoing performance problems. 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 statistical 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 statistically insignificant, and it becomes harder to realize and diagnose.

(ii) *Semantic diagnosis is more effective than statistical diagnosis under speculative execution.* Since semantic diagnosis is able to trace the exact sequence of syscalls on each node, it is able to realize a sudden spike in the reduction of syscall invocations on a given node. As a result, it is able to trace Hadoop's effort to abandon a task's execution if another redundant copy of the task finishes the computation first.

VI. RELATED WORK (Still incomplete)

<todo: insert citations!>

A. Diagnosis for MapReduce frameworks

X-Trace was used to instrument Hadoop and provided fine-grained trace events and a summarized views as a form of instrumentation on Hadoop workloads. Additionally, other work has focused on handling errors, and the use of log-based diagnosis. Xu et.al mined DataNode logs to detect outlier errors and Lou et.al traced errors to originating components. S. Kavulya et.al synthesized results from various algorithms using supervised learning and log-based tracing to enable root-cause diagnosis.

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.

B. System call based diagnosis

M. Kasick et.al focused on a very similar use of syscalls to debug the PVFS framework, but our approach differs as we deal with a qualtiatively different framework from PVFS. Additionally, we do not use error-based semantic correlation approach highlighted in that paper.

VII. CONCLUSION & FUTURE WORK

A. Reduced Instrumentation overhead

Currently running strace on the workloads can almost double the runtime in certain cases, and we want to build our own custom system call tracer that is optimized for our own use and as a result reduces the overall instrumentation overhead involved.

B. Focus on time-based debugging

We can extend the semantic algorithm to build a time-based debugging technique that lets us know, at what point in time a certain performance fault is experienced and at what node.

C. Conclusion (More work needed)

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.

We plan to focus on a greater set of syscalls in the future and build our own custom syscall tracer to optimize the instrumentation overhead. Additionally, we plan to integrate this with existing black-box debugging techniques.

VIII. ACKNOWLEDGEMENTS

I would really like to thank my advisor Priya Narasimhan for making time for me in her busy schedule and offering advice on how to write a good paper and follow the correct research methodology involved. Also I would like to thank Soila Kavulya and Mike Kasick for their advice on running the experiments/workloads involved.

IX. BIBLIOGRAPHY

<Incomplete, as I need to consult my advisor on this>

[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] Apache Software Foundation, "Hadoop," 2007, http://hadoop.apache.org/core.
[3] ——, "Powered by Hadoop," 2009, http://wiki.apache.org/hadoop/ PoweredBy.