Transparent System Call Based Performance Debugging for Cloud Computing

Nikhil Khadke and Prof. Priya Narasimhan
School Of Computer Science Senior Thesis - Working Thesis Draft
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 it 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 correlates error numbers ..

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 ouptut 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: Hadoop MapReduce programs consist of Map and Reduce tasks written as Java classes.

B. Hadoop Architecture **<largely based on the visual log based paper>**

Hadoop has a master-slave architecture as indicated in Fig 1., with a single master and multiple slave hosts. Hadoop involves an execution layer which executes Map and Reduce tasks, and the Hadoop Distributed Filesystem (HDFS), 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 natively generates logs which record error messages, as well as system execution events, e.g. starts and ends of Maps and Reduces.

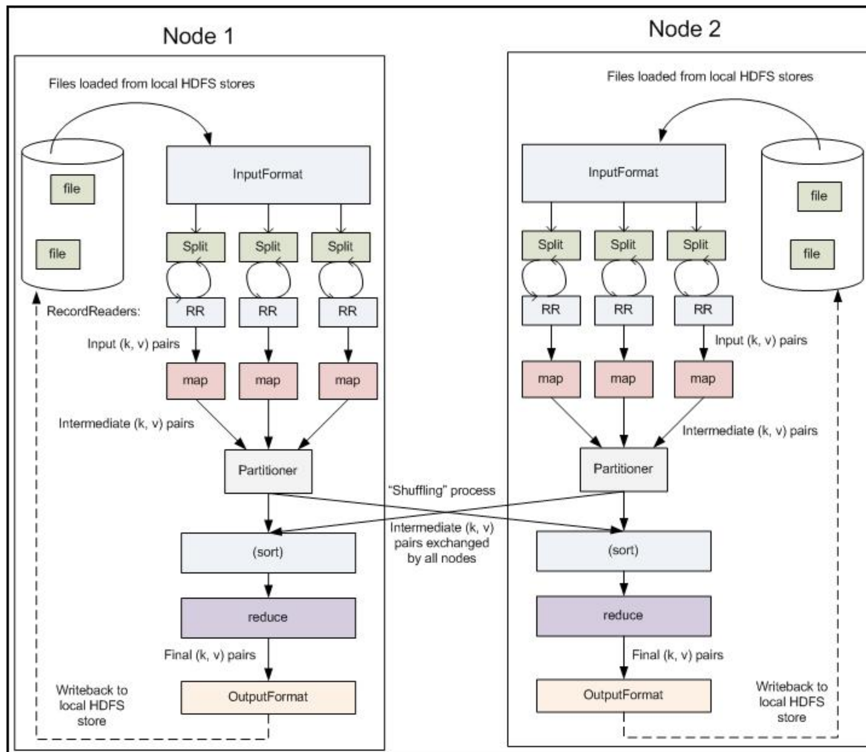

Fig 1 : Hadoop Architecture <I'll replace this with a better image soon>

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 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 need 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. `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 --

- Total calls made to a set of chosen syscalls
- Average time spent for each syscall
- Number of failed syscalls
- Percentage of time of spent in the syscall with respect to other monitored syscalls

(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.

The -f flag in `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. The first workload *wc* is a naive MapReduce wordcount of each word occurring in a relatively large corpus of 100000 words. The second workload *sort* sorts 100000 integers. We run two instances of this program, one as a control experiment and one instance with a certain fault injected into a specific node at a given time and for a certain duration (360s in our setup). 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. We evaluate the discussion of common faults such, which specifically are -

C. Performance problems:

(i) *disk-hog.* In our setup we write 2GB chunks 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.

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.

Semantic Syscall-based diagnosis

<Need to talk to advisor about this >

V. RESULTS AND DISCUSSION

Very Summarized Results (only Statistic for now. Semantic analysis will be completed after talking to advisor).

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 128MB, 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 a diagnosis success of **1.00**. Our false positive rate was **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:

Level of network-hog	False positive rate	Diagnosis success
5%	0.00	0.76
20%	0.00	0.84
50%	0.00	1.00

C. Results: Semantic Syscall-based diagnosis

<NEED TO DISCUSS with advisor >

Bibliography

[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.

TODOS:
<Talk about different workloads and their minimal effect on diagnosis algorithm in detail after discussing with advisor>
<Discuss Future Work + overhead in method>
<Acknowledgements + Biblio>

<Look into future work of DRAM clusters, which links well into embedded and cloud computing. >
http://www.youtube.com/watch?v=lcUvU3b5co8&fb_source=message