Visual, Log-based Causal Tracing for Performance Debugging of MapReduce Systems

Jiaqi Tan
DSO National Laboratories, Singapore
Singapore 118230
tjiaqi@dso.org.sg

Soila Kavulya, Rajeev Gandhi
and Priya Narasimhan
Electrical & Computer Engineering Dept.
Carnegie Mellon University, Pittsburgh, PA 15213
{spertet,rgandhi}@ece.cmu.edu, priya@cs.cmu.edu

***Abstract*—The distributed nature and large scale of MapReduce programs and systems poses two challenges in using existing profiling and debugging tools to understand MapReduce programs. Existing tools produce too much information because of the large scale of MapReduce programs, and they do not expose program behaviors in terms of Maps and Reduces. We have developed a novel non-intrusive log-analysis technique which extracts state-machine views of the control- and data-flows in MapReduce behavior from the native logs of Hadoop MapReduce systems, and it synthesizes these views to create a unified, causal view of MapReduce program behavior. This technique enables us to visualize MapReduce programs in terms of MapReduce-specific behaviors, greatly aiding operators in reasoning about and debugging performance problems in MapReduce systems in a scalable fashion. We validate our technique and visualizations using a real-world workload, showing how to understand the structure and performance behavior of MapReduce jobs, and diagnose injected performance problems reproduced from real-world problems.**

I. Introduction

MapReduce (MR) is a programming paradigm and framework introduced by Google [1] for parallel distributed computation on commodity clusters. It allows programmers to easily process large datasets, hiding low-level intricacies of distributed execution from user code. As a result, MapReduce has gained enormous popularity: the main open-source Java implementation, Hadoop [2], is used at large companies such as Yahoo! and Facebook to process petabytes of data daily [3]. Hadoop has been deployed at scale on clusters with hundreds to thousands of nodes and petabytes of storage [4] [5].

Debugging performance problems of MR programs, e.g. in Hadoop, is difficult due to their scale and distributed nature. The increasing scale of MR clusters necessitates automating diagnosis of performance problems–inefficient large programs waste compute resources, and the large scale of these clusters renders manual debugging infeasible. Currently, Hadoop programs can be debugged by examining its per-node logs of execution. However, these logs can be large[1], and must be manually synthesized across nodes to debug system-wide problems. Alternatively, Hadoop MR programs can be debugged using tools such as `jstack` and `jprof` [6], that are specific to the programming language used to implement the MR framework (i.e. Java). However, they target programming-language abstractions to debug local code-level errors rather than distributed problems across multiple nodes. Also, these tools instrument only user code, and expose only behaviors in user-written code, but will not capture MR abstractions such as Maps and Reduces, nor framework behaviors such as task scheduling and data movement which also affect program performance. Similarly, path-tracing tools for distributed systems [7], [8], [9] produce fine-grained views at the language level, and their views need to be further synthesized to produce program views at the MR level of abstraction.

Our survey of the Hadoop user's mailing list indicates that the most frequent performance-related questions are indeed at the level of MR abstractions (see II-C). The MR framework affects program performance at the macro-scale through task-scheduling and data distribution. This macro behavior is hard to infer from low-level programming-language views of user-code behavior because of the glut of low-level detail, and the macro behavior occurs outside of user code. We propose that these MR-specific aspects of dynamic, macro-scale behavior are relationships in time (e.g., orders of execution), space (e.g., which tasks ran on which nodes), and the volumes of data processed in each MR stage. This motivated us to extract and analyze Hadoop's time-, space-, and volume-related behavior to capture the full distributed data- and execution-views that impact MR performance. Finally, we cope with the scale (number of nodes, tasks, and durations) of Hadoop MR programs by visualizing program behavior to aid users in quickly and intuitively detecting deviations from expected program behavior and performance.

We build on prior work, using the SALSA log analysis technique [10] that extracts state-machine views of each node's behavior from its logs, and use visual metaphors of Hadoop's behavior [11]. Our contributions are: an algorithm and scalable tool to correlate state-machine views across nodes and layers to build a distributed, conjoined causal, control- and data-flow view of MR behavior for Hadoop, and new visualizations that enable causal tracing of program behavior.

II. Background & Problem Statement

A. MapReduce Paradigm

A MapReduce program consists of two user-written functions: a Map and a Reduce. The Map function is first applied

[1] A Sort benchmark running for 850s on a 5-slave cluster generates about 7 MB or 42,000 lines of logs per node

Category	Question	Fraction
Configuration	How many Maps/Reduces are efficient? Did I set a wrong number of Reduces?	50%
Data behavior	My Maps have lots of output, are they beating up nodes in the shuffle?	30%
Runtime behavior	Must all mapper s complete before reducers can run? What is the performance impact of setting X? What are the execution times of program parts?	50%

TABLE I
COMMON QUERIES ON USERS' MAILING LIST

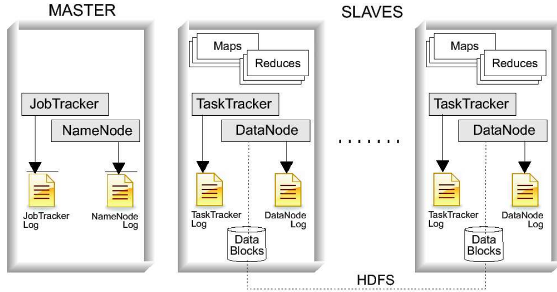


Fig. 1. Architecture of Hadoop, showing the locations of the system logs of interest to us.

to the data, and the Reduce is then applied to the Map's output. The MapReduce framework splits the input dataset into smaller partitions, and spawns multiple copies of Maps and Reduces to operate on each partition in parallel. The framework also transparently manages the Shuffle, which moves the Maps' outputs to the Reduces. Hadoop is an open-source, Java implementation of MapReduce: Hadoop MapReduce programs consist of Map and Reduce tasks written as Java classes.

B. Hadoop's Architecture

Hadoop has a master-slave architecture, with a single master and multiple slave hosts, as shown in Figure 2. Hadoop consists of an execution layer which executes Maps and Reduces, and the Hadoop Distributed Filesystem (HDFS), an implementation of the Google FileSystem [12]. The master host runs the JobTracker daemon, which schedules task execution on slaves and implements fault-tolerance using heartbeats sent to slaves, and the NameNode daemon, which provides the namespace for HDFS. Each slave host runs the TaskTracker daemon, which executes Maps and Reduces locally, and the DataNode daemon, which stores and serves data blocks for HDFS. Each Hadoop daemon is a Java process, and natively generates logs which records error messages, as well as system execution events, e.g. starts and ends of Maps and Reduces.

C. Hadoop Mailing List Survey

Next, we studied messages posted on the Hadoop users' mailing list [13] to survey the performance-related problems and questions users had with their MapReduce programs. We examined posts from October 2008 to April 2009, and focused on questions from users about optimizing the behavior of MapReduce programs (the other common kinds of questions were pertaining to "How do I get my cluster running", and "How do I write a first MapReduce program", which we excluded). Out of the 3400 threads, we found approximately 30 relevant posts. We list common types of questions in Table I, and the fraction of these posts they made up (some posts had multiple categories). All user questions focused on MR-specific aspects of program behavior, and we found that the answers tended to be from experienced users based on heuristics that might not work in different environments. Also, many questions were on dynamic MR-specific behavior, e.g. relationships in time (orders of execution), space (which tasks ran on which nodes), and amounts of data in various program stages, showing a need for tools to extract such information.

D. Goals

Our goals for this work are:

To expose MapReduce-specific behavior that results from the MR framework's automatic execution, that affect program performance, but is neither visible from not exposed to user MR code, e.g., when Maps and Reduces are executed and on which nodes, where data inputs and outputs flow from/to.

To expose aggregate and dynamic behavior that can provide different insights. For instance, in the time dimension, system views can be instantaneous or aggregated across an entire job; in the space dimension, views can be of individual Maps and Reduces or aggregated at each node.

To enable causal tracing of all program data and control flows in a MR program. This enables detailed what-if analysis to be carried out on MR programs to perform bottleneck analysis and trace performance deviations to their source.

To be transparent to Hadoop without requiring additional instrumentation of Hadoop, by either administrators to the framework, or by users in their own MapReduce code, allowing our techniques to work on Hadoop MapReduce programs running on commodity, default installations of Hadoop.

E. Non-goals

Our focus is on exposing MR-specific aspects of program behavior, rather than behavior within each Map or Reduce. Thus, the execution specifics and correctness of code within a Map or Reduce is outside the scope of our work. Also, we aid programmers in discovering the root-cause of performance problems using visualizations that can provide useful insights, but do not automatically identify the root-cause of problems.

F. Assumptions

We assume that Hadoop's logs faithfully capture the system's execution state, i.e. all relevant events of interest as defined in our model (Section III) are logged, with no missing nor additional (spurious) events. We assume that the timestamps on each host's locally-generated logs accurately reflect

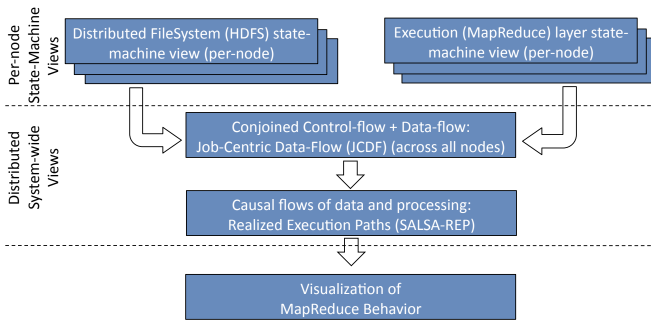


Fig. 2. Overview of our approach

the order in which the statements are logged. However, we do not assume globally synchronized clocks as we use unique identifiers to causally correlate events (Section IV-B).

III. Approach: Abstracting MR Behavior

Next, we describe our novel abstractions of MR behavior; we leave the Hadoop specifics of our abstractions to Section IV, and we describe the visualizations which build on these abstractions in Section V. We create novel abstractions of MR behavior which: (i) capture user-written Maps and Reduces and automated MR framework behaviors, (ii) account for the distributed nature of MR, and (iii) enable tracing of causality, i.e. temporal flows of data into and out of processing tasks, across multiple stages. These abstractions are based on Hadoop MR behavior extracted from its native system logs.

A. Per-node State-machine Views

SALSA [10] is a general log-analysis technique for extracting from logs state-machine views of system execution in terms of its control- and data-flows. We use SALSA to extract per-node state-machine views of each Hadoop node's behavior from its logs, from which we build further abstractions. SALSA extracts these control-[2] and data-[3]flows from the system's logs, given *a priori* knowledge of the structure of the system's execution: possible states of execution, normal sequence of these states, and tokens in log statements identifying these states. SALSA does not infer the correct state-machine model of a system's execution; rather, given the state-machine model, SALSA extracts runtime properties (runtimes of and data-flows between states) of the state-machine, which is assumed to be structurally correct[4]. Each state-machine view consists of a list of states executed with their unique identifiers, and the times at which the state began and ended. SALSA extracts the state-machine views of each node's execution from its logs, but the states in these per-node views are later correlated to form distributed views using their identifiers.

In a Hadoop MR system, two types of state-machines are extracted from each node's logs: one for the distributed filesystem, from the DataNodes' logs, representing the data-flow, and one for the execution-layer from the TaskTrackers' logs representing the control-flow[5].

B. Job-Centric Data-Flows

Next, we correlate the per-node state-machine views generated by SALSA to construct a view of the full causality of processing in an MR system, i.e. all data and processing pathways in the MR program, which flow through the state-machines of every node. We call this abstraction a Job-Centric Data-Flow (JCDF), as each JCDF is an abstraction of the execution of a single job, consisting all the data-flows necessary for the input and output of data to and from the job, and all the control-flows of processing tasks performed. The JCDF is a directed graph, with a vertex for each control-flow and data-flow state in the SALSA state-machines, and an edge every flow of causality: either an explicit data-item being transferred, or the passing of control from one processing stage to another. Edges are then annotated with the volume of data transferred, and the time taken for that transfer of control.

C. SALSA-REP: Realized Execution Paths

Finally, each path beginning from a vertex with 0 in-degree and ending at a vertex with 0 out-degree in the JCDF directed graph represents a single thread of causal flow in the system. These are end-to-end paths in the JCDF graph, and we call these threads of executions Realized Execution Paths (REP), as they represent a single persistent flow through the system.

IV. Methodology

We describe Hadoop's state-machines of execution as derived from its logs using SALSA, the resulting JCDF from correlating these per-node state-machines, and the extracted Realized Execution Paths (SALSA-REP), followed by our visualizations of MR behavior based on these views.

A. Node-Local State-Machine Extraction

SALSA extracts one set of execution states for each of the TaskTracker and DataNode daemon's state-machines respectively, with the former describing both control-flows and data-flows, and the latter describing data-flows.

Control-Flows The main control-flow states in the TaskTracker are Maps and Reduces; in addition, control passes from Maps to Reduces via the Shuffle stage which copies outputs of Maps to the relevant Reduces, transparent of user code. Each Map can pass control to multiple Shuffles to copy its output to multiple Reduces, and each Reduce can receive control from multiple Shuffles when it requires input from multiple Maps. We further decompose the Reduce into three states: the ShuffleWait, ReduceSort, and Reducer states, in chronological order of execution. ShuffleWait is the period when Reduces wait for all Maps to complete execution, and to receive data from all its Shuffles; the ReduceSort state presorts the Maps' outputs, while the Reducer executes the user-written Reduce code. In addition, we introduce a MapWait

[2]Orders and durations of execution of processing tasks, e.g. Maps, Reduces
[3]Explicit data-items transferred.
[4]This implies that incorrect orders of state executions are out of scope of this work. See [14] for a discussion.
[5]As well as data-flow in transfers of Map outputs to Reduces.

state, to account for the scheduling delays between the start of the job to the time the Map is executed.

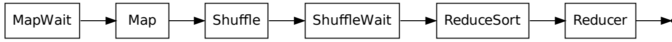


Fig. 3. States in the state-machine view of Hadoop's control-flow

Hence, the control-flow view of MR programs is as illustrated in Figure 3, along each independent thread of execution. We preserve causality along each thread of execution using unique identifiers of Maps and Reduces in log messages.

Data-Flows There are two types of data-flows in Hadoop MR programs: the movement of data blocks written to/read from HDFS, and the movement of Map outputs to be used as inputs to Reduces. In HDFS data-flows, the two states of execution of the DataNode are block reads (ReadBlock) and writes (WriteBlock), between DataNodes, and HDFS clients (e.g., Maps and Reduces). We further classify these states into local/remote block reads/writes. The data-flows between Maps and Reduces in TaskTrackers occurs at the Shuffle stage, and they are captured in the source Map ID and destination Reduce ID recorded in each Shuffle log message.

B. Cross-Node, Cross-Layer Stitching

Next, we describe the construction of the Job-Centric Data-Flows (JCDF) for Hadoop MR programs from the per-node SALSA control-flow and data-flow state-machines. The JCDF is a graph, with vertices made up of data-items and control-items, as shown by the ellipses and boxes in Figure 4. These data-items are states in the data-flow state-machines, and these control-items, are states in the control-flow state-machines. The directed edges in the graph then shows the flows of causality from one state to the next.

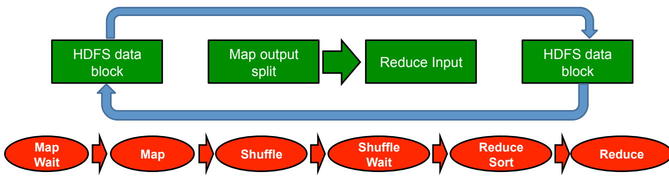


Fig. 4. Control-flow items (ellipses, in read) and data-flow items (boxes, in green) of states in the state-machine view of Hadoop's execution.

First, the control items are the MapWait, Map, Shuffle, ShuffleWait, ReduceSort, and Reducer states, while the data-items are the ReadBlock, WriteBlock, MapOutput, and ReduceInput states, as shown in Figure 4.

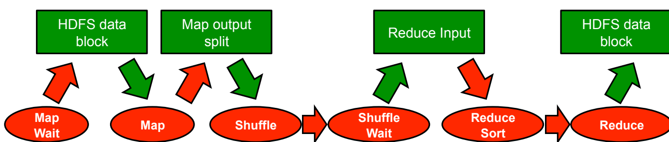


Fig. 5. Job-Centric Data-Flow formed by "stitching" control-items (ellipses in red) and data-items (boxes in green) of Hadoop's execution states. Directed edges show direction of full causality of execution in a MapReduce system.

Second, the JCDF is formed by "stitching" together control-items and data-items by identifying the input and output data-items for each control-item. This flow is shown in Figure 5. Vertices are created for each control- and data-item observed in the state-machines on each node, and edges are added between them corresponding to specific data-items which are inputs to and outputs from specific control-items. Additionally, each of these edges is annotated with two weights: one representing the duration of the operation (e.g., for the `(MapWait,DataBlock)` edge, this represents the scheduling delay before the first data block is read by the corresponding Map), and the second representing the volume of data transferred (e.g. for the `(MapWait,DataBlock)` edge, this represents the size of the block read).

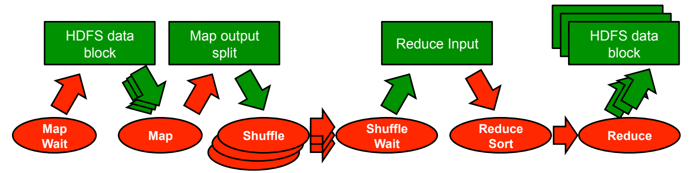


Fig. 6. Job-Centric Data-Flow formed by "stitching" together control-items (ellipses in red) and data-items (boxes in green) of Hadoop's execution states. Directed edges show direction of causality of execution in an MR system.

The input/output relationships between the control- and data-items are fully specified in the log messages emitted by Hadoop: in particular, the identifiers of clients of block reads and writes (i.e. Maps and Reduces respectively), and the identifier of each source Map read by each Reduce [15], are logged by Hadoop's `ClientTrace` log messages. This forms the directed JCDF graph of full control- and data-flows in an Hadoop MR program, with vertices for the Map and ShuffleWait states potentially having in-degree > 1 (when the Map reads from multiple blocks, and the ShuffleWait receives inputs from multiple maps), and the vertices for the Map and Reduce states potentially having out-degree > 1 (when the Map's outputs is shuffled to multiple Reduces, and when the Reduce writes to multiple blocks), as shown in Figure 6.

C. Causal Flows

Finally, each causal flow in the system is simply a path that begins from a vertex with in-degree 0 and ends on a vertex with out-degree 0, and we call each causal flow a Realized Execution Path, or SALSA-REP. All REPs begin on MapWait states, and traverse the following states in each path: MapWait, ReadBlock, Map, MapOutput, Shuffle, ShuffleWait, ReduceInput, ReduceSort, Reduce, WriteBlock. Then, edges are weighted with the duration taken for that operation, or the volume processed in that transition. The SALSA-REPs can be recovered using a Depth-First Search on the JCDF. The total edge weights represent the total processing volume in that flow, or the total processing and scheduling delay times along that flow. Then, the duration of an SALSA-REP flow is the sum of all duration edges along that flow.

V. VISUALIZATIONS FOR PERFORMANCE DEBUGGING

Next, we introduce statistical properties and visualizations of our abstractions which are useful for performance debugging, and we describe heuristics based on these for understanding MR behavior. As described in Section I, we can decompose the behavior of a MR program along the dimensions of time (times, orders, and durations of execution), space (which nodes states executed on), and volume (volumes of data processed and transferred). Each visualization exposes MR behavior along a combination of these dimensions, and we describe the useful insights for program performance that each combination shows. We also describe outlier selection and analysis algorithms based on the SALSA-REPs extracted from our abstraction for understanding performance problems. To illustrate our visualizations and views of MR behavior, we use plots of one typical job in the multi-job HADI workload on our *fp* testbed (described in Section VII). We leave performance debugging case studies to Section VII-C.

A. Performance Characterization

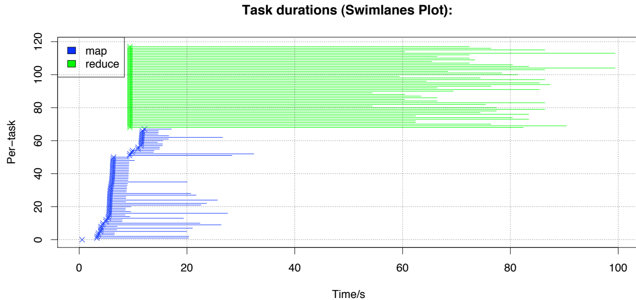


Fig. 7. Swimlanes plot of candidate job illustrating execution of Maps and Reduces over time in the job.

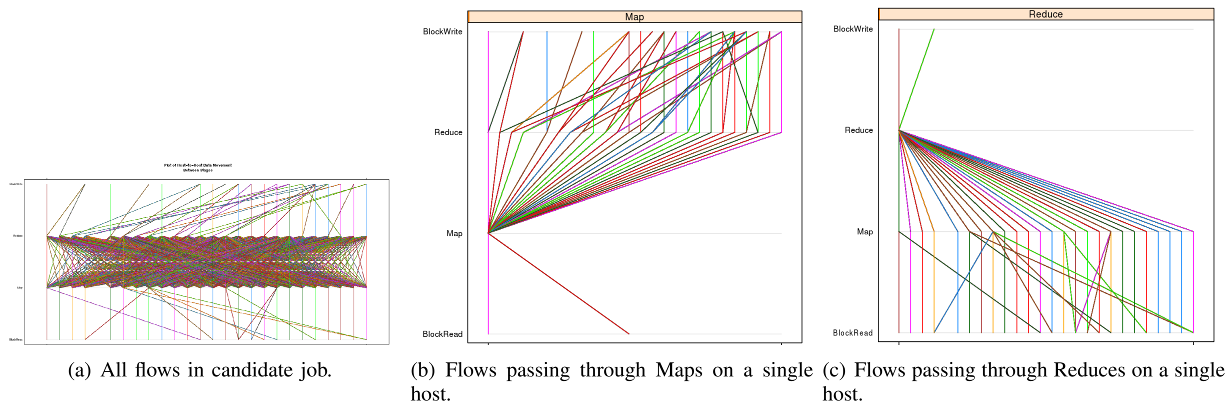
1) Macro-level Space-Time Behavior: First, the "Swimlanes" visualization, first introduced in [11], shows a summary of task progress in a job as its execution unfolds in time; the Swimlanes plot shows the execution of only Maps and Reduces, hiding the finer grained details (e.g. ShuffleWait, ReduceSort), for readability. For each plot, the horizontal axis shows the time elapsed since the time the job was submitted, and for each task (Map or Reduce) in the job, and a horizontal line is plotted from the time the task began executing, to its time of completion. On the vertical axis, each tick corresponds to a unique task. Horizontal lines representing tasks and their times of execution can be sorted in different orders to yield different useful views. In the basic Swimlanes view, tasks are sorted by the order in which they were launched, showing chronological progress of the job's execution. In general, the Swimlanes plot provides a quick, intuitive way of understanding the behavior of a MR program at a high level, and captures the dynamic behavior of where the job, and nodes, spend their time.

2) Macro-level Space-Volume Behavior: Next, we describe the Parallel-Coordinate Plot, as shown in Figure 8(a), which shows the data exchanges between cluster hosts/nodes between successive states of execution in a given job. This plot shows the behavior of jobs in space, where data is being transferred to and from, and in volume, showing the number of transfers in the topology across hosts. This plot allows us to quickly identify workload imbalances across hosts in the cluster, by evaluating the degree of data locality of reads and writes from and to HDFS, and of processing locality between Maps and Reduces. Each host occupies one tick on the horizontal axis, and the execution of the job progresses from the bottom of the graph to the top along the vertical axis. The four horizontal axes across the plot represent the hosts in the BlockRead, Map, Reduce, and BlockWrite states. Lines plotted represent transfers from a host in one state to a host in the next. For instance, a line plotted from a host in the BlockRead state to a host in the Map state represents a flow in the job where a block is read from the first host, to a Map executing on the second host.

Figure 8(a) shows a full Parallel-Coordinate Plot, which plots all flows occurring in our candidate job; this view effectively summarizes all the SALSA-REP flows in the job, hiding away the details of durations and volumes of data processed. This full plot shows the complexity of the interactions in a MR job, and the tight, highly dense interconnection between hosts in the Map and Reduce states is common in MR jobs. To simplify the job views, we filter plots by hosts, and show only flows in which a given state (i.e. Map, Reduce, BlockRead, or BlockWrite) executed on that particular host. Figure 8(b) shows a typical filtered Parallel-Coordinate Plot, filtered by Map host, for a given host, and from this plot we can see that Map tasks on a given host typically read blocks from few other hosts, but their outputs are sent to Reducers on almost all hosts in the cluster. Likewise, Figure 8(c) shows a typical filtered plot filtered by Reduce host, and we can see that Reduces on a given host often receives outputs from Maps on all hosts, and usually write their outputs to a small number of hosts.

3) Detailed views of SALSA-REP durations: Next, we present visualizations and statistical properties of the SALSA-REP causal flows in each job's execution. The SALSA-REP flows are the finest-grained information our abstractions capture, and provide the most detail into each job's execution. We present macro-level, summary views of the SALSA-REPs across a job, as well as fine-grained views, and highlight insights and interpretations for understanding job behavior.

Distribution of SALSA-REP durations: First, we present an overview of the performance of the job across all SALSA-REP flows that comprise the job. We plot the histogram and cumulative density function (CDF) of the total duration of all states occurring in every flow of the job. Ideally, every causal flow should have equal durations, since the job is as slow as its slowest flow. However, there are various sources of non-determinism in real systems, such as scheduling, imperfect data distribution, and other perturbations. In general, we have observed that in most jobs, the durations of SALSA-REP flows tend to have most flows tightly clustered around the modal duration. Poorly performing jobs will then have outlier flows with durations that are significantly longer than the



(a) All flows in candidate job.

(b) Flows passing through Maps on a single host.

(c) Flows passing through Reduces on a single host.

Fig. 8. Parallel-Coordinate Plots of end-to-end host data exchanges across the execution states of the candidate job, filtered by Map and Reduce hosts respectively. Horizontal axes across the plot represent hosts in the BlockRead Map, Reduce, and BlockWrite states respectively, from bottom to top.

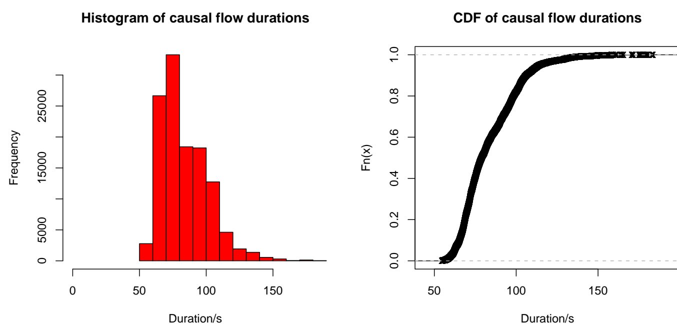


Fig. 9. Histogram (left) and Cumulative Density Function (right) of durations of all SALSA-REP causal flows in our candidate job.

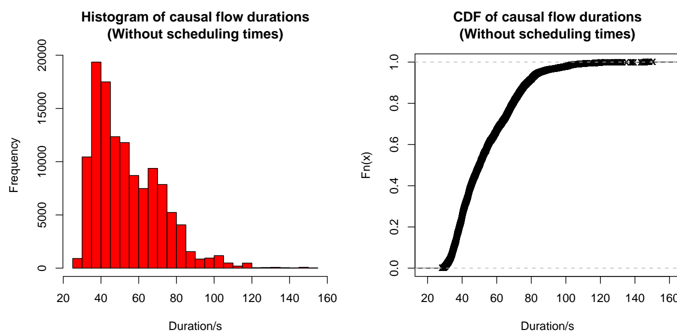


Fig. 10. Histogram (left) and Cumulative Density Function (right) of durations of all SALSA-REP causal flows in our candidate job, excluding times spent in scheduling delays (i.e. MapWait and ShuffleWait states).

median/modal duration, and this will be easily seen from the histogram of flow durations. Figure 9 shows a typical plot of the histogram and CDF for one job in our workload, which illustrates our observed trend.

SALSA-REP Scheduling delays: Next, we show the time in the SALSA-REP flows in the job spent on actual processing and data transfers, by plotting the durations along each flow after excluding the time spent in "waiting" states, namely the MapWait state, in which processing waits for a free Task-Tracker before a Map can be run, and the ShuffleWait state, which is spent waiting on the required outputs from all relevant

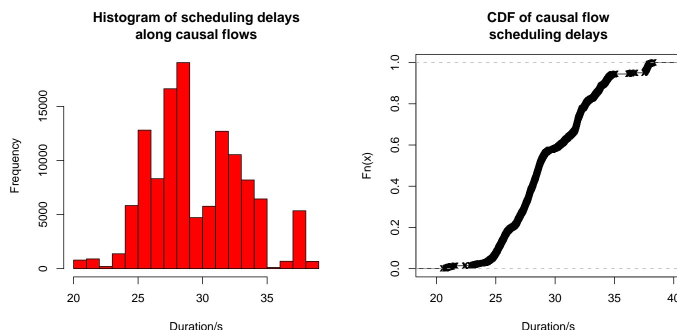


Fig. 11. Histogram (left) and Cumulative Density Function (right) of times spent in scheduling delays (i.e. MapWait and ShuffleWait states) all SALSA-REP causal flows in our candidate job, excluding.

Maps to be copied before Reduces begin execution. We plot the histogram and CDF of these actual processing durations of all flows, as shown in Figure 10. Comparing the statistical distributions of durations of actual processing (e.g. Figure 10) and of total durations including scheduling delays (e.g. Figure 9) allows for disambiguation between problems due to inefficient scheduling, and problems occurring during actual processing, and also allows the quantification of scheduling inefficiencies. In addition, we present characteristics of the times spent in scheduling delays along all the causal flows in the job, as shown in Figure 11, which directly quantifies scheduling delays. In our candidate job, we can see that scheduling delays are a modest fraction (at most $20\% - 30\%$ of total durations) of the overall flow durations, an acceptable tradeoff for the ease of parallelization provided by Hadoop.

Decomposing SALSA-REP durations by state: Next, we decompose the durations of SALSA-REP flows in the job by the durations of each of the states in each flow, and plot the distribution of durations of each state across all flows. Figure 12 shows a box-plot of the durations of states in every causal flow. Although this plot does not allow any inferences to be made across causal flows since the flows have been decomposed into their component states, it can quickly highlight problematic states (e.g. slow Maps, or slow

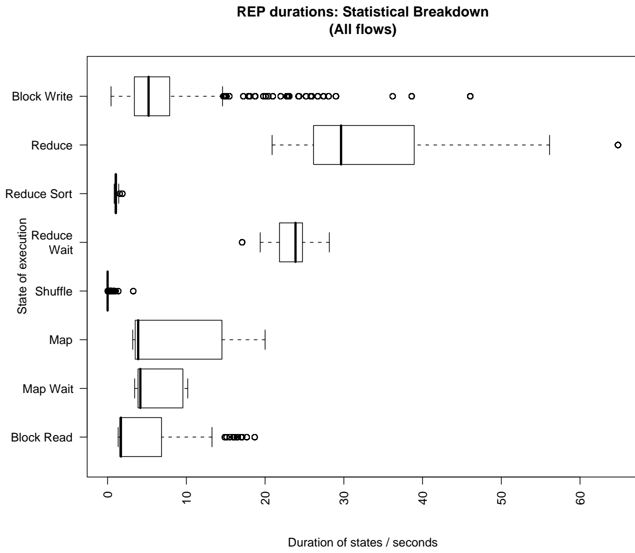
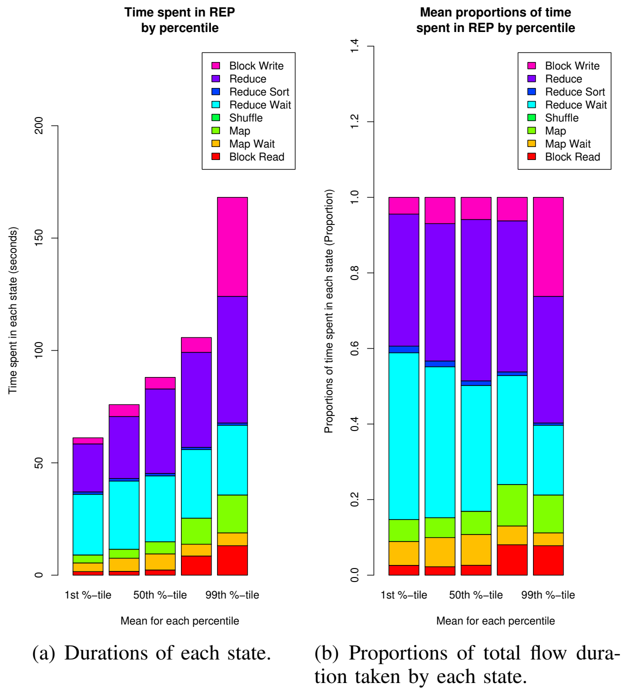


Fig. 12. Box-plot showing durations of each state across all SALSA-REP causal flows in our candidate job. Middle rectangles show 25-th, 50-th, and 75-th percentiles, and the two "notches" at the extreme ends show extents where values fall within twice the median value. Circles indicate outliers.



(a) Durations of each state.

(b) Proportions of total flow duration taken by each state.

Fig. 13. Representative ranks of SALSA-REP flows for our candidate job, and durations and proportions of total runtime of the flows, decomposed by state, within flows of that rank.

Block Reads) which have long durations, or which have many outliers, enabling further investigation.

Decomposing SALSA-REP flows by rank-order: In our next view, we rank SALSA-REP flows by their total flow durations, and show the breakdown of durations by state for representative ranks. Figure 13(a) shows the rank-ordered SALSA-REP flows decomposed by state: the 1st, 25th, 50th, 75th, and 99th percentiles of flows are selected, and the means

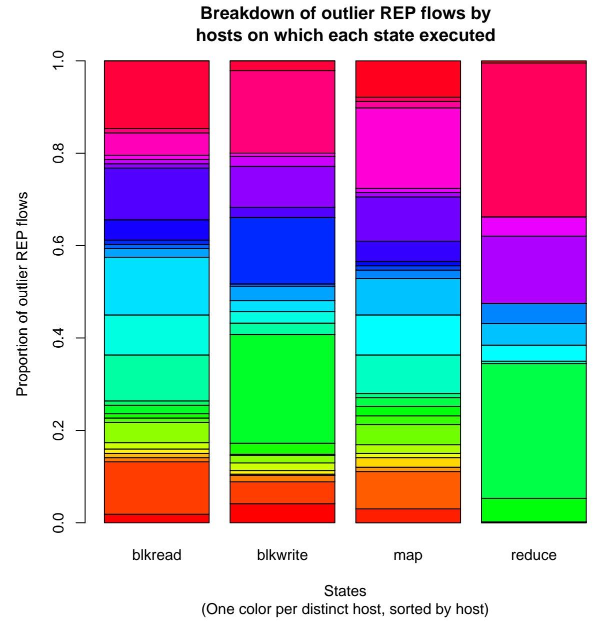


Fig. 14. Host-State Decomposition of outlier SALSA-REP flows, showing the proportion of flows occurring on each host for each state.

of the durations of flows spent in each state are shown. Selecting only representative ranks of flows to decompose into states preserves the causality along flows, and allows us to reason causally about performance. For instance, we can identify if slow states in a later state are associated with an earlier state that was slow, along the same flow. In addition, Figure 13(b) shows, for each of the representative ranks of SALSA-REP flows, the proportions of time spent in each state along the flow, rather than actual durations. Reasoning about flows in terms of the proportions of time rather than actual time spent in each state gives us an intuition of whether a particular state is disproportionately slower (or faster) on flows of a given rank, indicating possible problems with that state.

B. Anomaly Detection

Next, we describe anomaly detection and analysis as performed on the SALSA-REP causal flows of a job. This allows debugging efforts to focus on the slowest flows in the job to uncover causes of these slow flows. Our anomaly detection does not indict any of these flows; rather, we isolate them and highlight interesting properties to users to expedite the process of reasoning about the performance of the program.

Outlier Selection: We identify SALSA-REP flows with anomalously long durations, relative to all flows in the job, as outliers. First, we compute the mean (μ) and standard deviation (σ) of the durations of all the SALSA-REP flows in the job. Then, we select all flows with durations x such that if the flow durations followed a Normal distribution, the probability of observing flows with the particular duration is $< 1.5\%$:

$$x > \mu + (2.17\sigma)$$

Outlier Groupings: Next, we decompose the outlier SALSA-REP flows identified in the last step by their component states,

and by the hosts on which they executed. This allows us to identify if the outlier flows were significantly correlated with a particular state, and/or a particular host, enabling us to indict that state and/or host as being the cause of the slow, outlier flows. Concretely, for each outlier flow, we identify the Maps (Reduces, Block Reads, Block Writes) in the flow and the host on which that state executed, and we compute the proportion of outlier flows with Maps (Reduces, Block Reads, Block Writes) which executed on that host. Figure 14 illustrates this state-host decomposition of the outlier SALSA-REP flows. Each bar shows the breakdown of flows for a given state (i.e. Block Reads and Writes, Maps, and Reduce), and each cell shows the proportion of outlier flows occurring on a given host, with hosts assigned fixed colors (i.e. each distinct host's share of flows has the same color).

Thus, we can identify two types of problems from this state-host decomposition of outlier flows: (i) problems originating from a single state, when all the outlier flows involve processing in a particular Map (or Reduce, Block Read, or Block Write), and (ii) problems originating from a single host, when a large number of outlier flows were executing on a given host.

VI. Implementation

Next, we describe the key architectural and implementation features of the SALSA state-machine construction and the correlation of states across nodes for Hadoop's logs.

A. Architecture

Hadoop Chukwa: Chukwa [16] is a log collection and analysis framework which leverages Hadoop for scalable processing of logs. Chukwa stores collected logs in HDFS, providing a format friendly to processing using Hadoop MapReduce programs and the Pig [17] declarative data processing language. We leverage Chukwa's log-storage formats, and implement our algorithms as log processing modules written as MapReduce programs and Pig scripts, which process data collected, formatted, and stored in HDFS by Chukwa. This enables Chukwa to act as a log normalization engine for consolidating logs collected from Hadoop, and for our tools to be automatically scheduled by Chukwa to process collected logs.

State-Machine Construction: First, we construct per-node state-machine views from the collected Hadoop logs from the JobTracker, DataNodes, and TaskTrackers using an MR program. Mappers consolidate the log-messages indicating the start and end of each state, and emit both with the same key to ensure the same Reducer processes both entries. Reducers then construct entries for each state by examining the start and end entries for the state, and annotate the state with information for correlating each state with its successor in the JCDF, e.g. BlockReads are annotated with the Mapper performing the read. See [14] for a detailed list of these added annotations.

State Stitching: JOINs in Hadoop Pig: Next, the full conjoined, causal, distributed state-machine view of execution for each job is extracted using Pig [17]. This involves specifying the "correlation" of states across nodes and between the control- and data-flows as a series of inner joins on the list of states, using the annotations added by the MapReduce program during the state-machine construction. The JCDF is then generated as a list of all causal paths occurring in the job, along with computed sums of processing durations and processed volumes along path.

Visualization and Statistical Analysis: Finally, statistical analysis is performed using Hadoop Pig to identify outlier SALSA-REP flows, and group them by various properties (e.g. by state, by host), and the visualization graphics are generated using the GNU R statistical package.

B. Performance

Table II describes the runtime and output sizes of our tool-chain for different numbers of nodes logged, and for different numbers of nodes used for processing. The runtimes were comparable across multiple runs. The large increase in the sizes of generated data as compared to the original log sizes shows the need for scalable processing. The reported output sizes are divided into that of generated numerical data, and that of plotted graphs. The reported runtimes are divided into that of the MapReduce and Pig processing, which is currently parallelized, and that of the graph plotting, which is currently sequential, although GNU R can be parallelized with Hadoop using the RHIPE [18] environment for using R scripts in Hadoop. The performance of our tool-chain shows promise for gains in runtime from parallelization, suggesting that MR is a good fit for our implementation.

Slave Nodes	Nodes Logged	Jobs Logged	Total Log Size	Output Size (Data)	Output Size (Graphs)	Runtime (Hadoop)	Runtime (Plotting)
1	25	45	65 MB	1.5 GB	0.4 GB	840 min	65 min
25	25	45	65 MB	1.5 GB	0.4 GB	238 min	42 min
15	100	18	500 MB	33.5 GB	2.4 GB	546 min	549 min

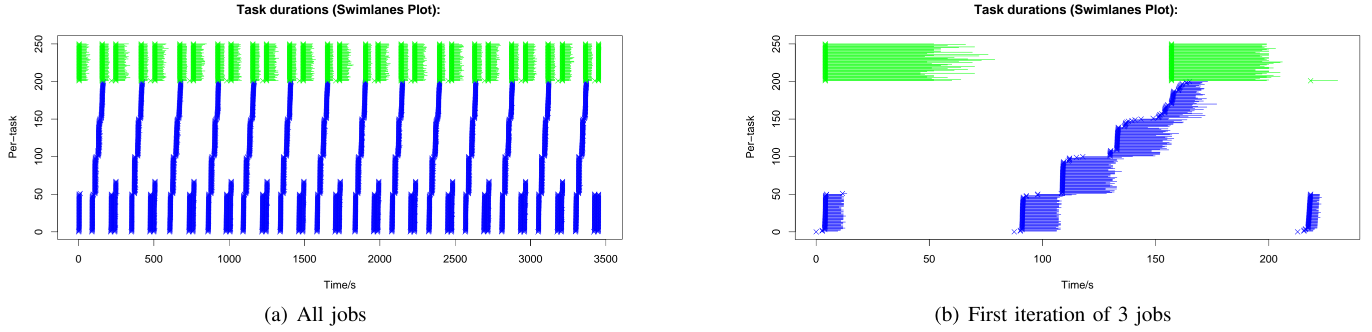
TABLE II
RUNTIMES OF OUR TOOL-CHAIN ON MR CLUSTERS OF VARIOUS SIZES (SLAVE NODES), FOR THE HADI WORKLOAD.

VII. Evaluation: Performance Debugging

Next, we study the MR program behaviors captured by our abstractions and visualizations, and evaluate the efficacy of using these to understand program behaviors and diagnose injected performance problems. We ran real-world and benchmark workloads and injected a real-world fault to evaluate our ability to detect these problems on these workloads.

A. Environment and Instrumentation

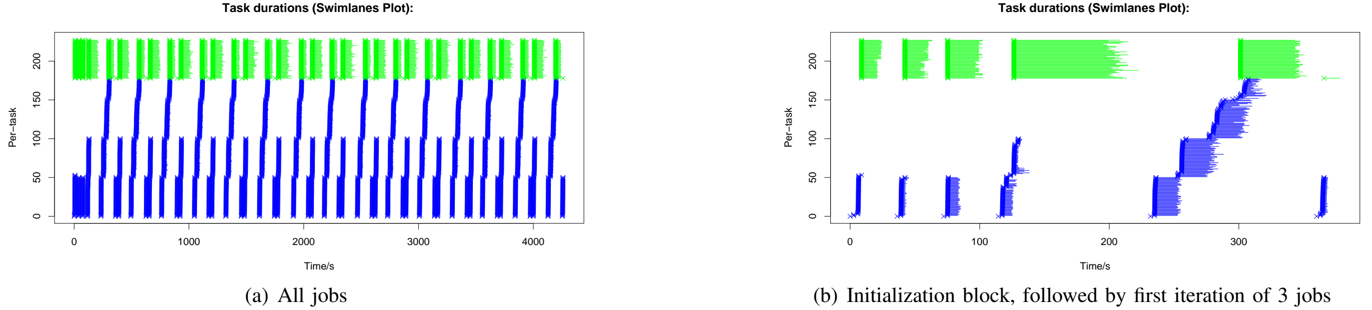
We performed experiments on our internal testbed, called *fp*, and on Amazon's EC2 [19] pay-as-you-use service to conduct experiments at large scales. We used Hadoop 0.20 and used default Hadoop-generated job history logs and ClientTrace log messages, and back-ported one enhanced logging feature from 0.21 [15]. Logs were collected at the end of each experiment and processed using Hadoop Chukwa's Backfilling tool for



(a) All jobs

(b) First iteration of 3 jobs

Fig. 15. Swimlanes plot of simple HADI, of entire workload, and of first iteration of 3 jobs; Maps are colored blue, Reduces are colored Green.



(a) All jobs

(b) Initialization block, followed by first iteration of 3 jobs

Fig. 16. Swimlanes plot of block-optimized HADI (HADI-BLK), of entire workload, and of first iteration of 3 jobs; Maps are colored blue, Reduces are colored Green. The first 3 jobs are an initialization phase not present in simple HADI.

retroactively loading logs[6]. We then used visualizations generated by our tool-chain to analyze the experiments, iterating through the visualizations in the order as they were described in Section V. The configurations of our testbeds are as follows:
FP: Each node consisted of an AMD Opteron 1220 dual-core CPU with 4GB of memory, and a dedicated 320GB hard-disk for Hadoop, and we ran the amd64 version of Debian/GNU Linux 4.0. The testbed ran 25 slave nodes and 1 master node.
EC2: Each node was a medium instance, with 4 EC2 compute units, equivalent to 2 dual-core processors, 7.5GB of memory, and used the Amazon Elastic Block Store (EBS) high-performance I/O service for Hadoop storage. The testbed consisted of 100 slave nodes and 1 master node.

B. Workloads

HADI: HADI [20] is a real-world application, developed by CMU researchers in large-scale data-mining, which performs large-graph radius and diameter estimation using matrix multiplications. It iteratively executes a sequence of three MR jobs until the output has converged. HADI typically operates on large connectivity graphs, such as web-graphs. In our tests, we used Wikipedia article connectivity graphs as inputs.

C. Experiments

Next, we describe the scenarios and injected problems in our experiments. We simulated a problem reported in the Hadoop users' mailing list [13] (Disk hogs), a data-skew scenario in which tasks received unevenly-sized amounts of data to process, and studied the difference between two versions of the HADI workload with differing designs.

1) Understanding Job Structure: In this scenario, we studied the behavior and structure of the HADI workload using the Swimlanes plots. In particular, we contrast an earlier, simple implementation of the matrix multiplication (that handled matrices in rows and columns (HADI), and a more sophisticated one that attempts to optimize processing by handling matrices in blocks (HADI-BLK).

2) Disk Bottleneck: In a Hadoop user's mailing list post dated Sept 26, 2007, excessive logging messages being generated led to a disk quickly filling up, slowing Hadoop down as a result. We simulated this problem by running a sequential disk-write workload which wrote 20GB of data to the disk on one slave node alongside Hadoop.

3) Data Skew: HADI-BLK is an enhanced version of HADI which performs matrix-vector multiplications by processing the matrix in blocks rather than rows and columns, which would in principle enhance data locality. We investigate the degree to which improved (i.e. reduced) data-skew between the Map and Reduce states contributed to performance improvements, and investigate the differences between the data-skew characteristics of these two workloads.

D. Results

Next, we describe how we detected the injected problems and analyzed the MR jobs using our visualizations, and

[6]Chukwa can be configured to process logs as they were collected, but we did not do so to avoid confounding our experiments with an additional Hadoop workload of log aggregation.

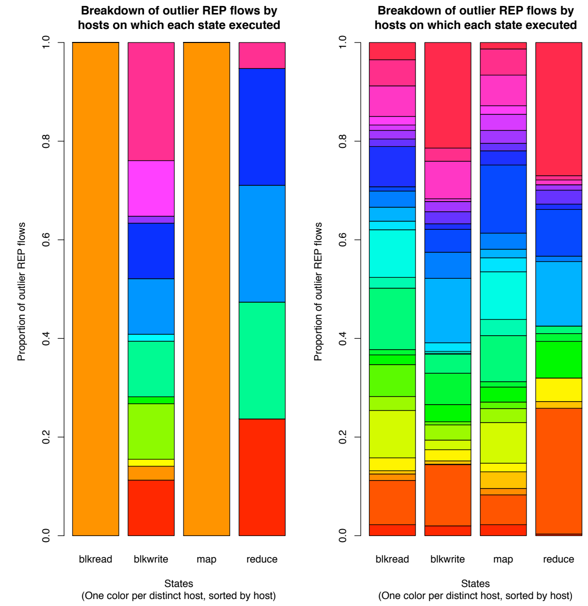
we walk through the intuition involved. Results from both *fp* and EC2 experiments were comparable, and we present results from our smaller *fp* cluster due to space constraints in presenting visualizations of the larger EC2 cluster.

1) Understanding Job Structure: Figures 15(a) and 16(a) illustrate the overall structure of the HADI and HADI-BLK workloads, which consist of multiple jobs. Each job consists of a block of Map tasks, in blue, and a block of Reduce tasks, in green, as the entire output of each job must be written permanently to HDFS before the next job can begin processing. Both workloads consist of a series of a basic sequence of three jobs (Figures 15(b) and the second three jobs in Figure 16(b)) that are iterated until convergence of the graph being processed. Given that each Map processes a single data block's worth of input, we can see that for HADI-BLK, the first job generates significantly more output than its input as compared to HADI, as the second job consists of a much larger number of Maps than the first. This also suggests that the Reduce tasks of the first job would have the heaviest workload, while the Reduce tasks of the second job perform aggregation and collation of results to generate total output of a smaller size than its input, as seen from the smaller number of Maps in the third job as compared to the second. Finally, the third job ends with a single Reduce task to assemble the results. Also, the first three jobs in Figure 16(a) of the HADI-BLK workload consist of an initialization sequence of three jobs (first three jobs in Figure 16(b)).

2) Disk Bottleneck: To determine if a particular node was responsible for slow program execution, we examined the Host-State Decomposition of outlier SALSA-REP flows, to check if (i) any given host was involved in a majority of the slow outlier flows, and if (ii) there were correlations between the hosts involved across different states.

In the case of the injected Disk Hog, during the period in which the fault was injected, we observed that amongst the outlier SALSA-REP flows, the Block Reads and Maps in these flows executed on the host on which the fault was injected for the vast majority of these flows ($> 80\%$). This can be seen in Figure 17(a): most of the slow outlier flows occur on the same host (single contiguous block accounting for the majority of flows), and this host is the same for both Block Reads and Maps (same color indicating identical hosts). This indicates that of the slowest flows in the job, the Block Reads and Maps all occurred on the same host. This also suggests that the Disk Hog affects the HADI workload more severely in the Block Read/Map phases of its execution, than in the Block Write/Reduce phase, as the outliers occurred on various hosts and were not associated strongly with any particular host. Conversely, for jobs that occurred during the period when no fault was injected, the Host-State Decomposition plot, in Figure 17(b), various states in the outlier flows ran on various hosts, suggesting that no particular host is the cause of the fault, and that these outliers are more likely to be statistical outliers than indicative of a performance problem.

3) Data Skew: We studied the difference in the data-skew in HADI and HADI-BLK by considering the Parallel-Coordinate



(a) Host-State Decomposition for job during injected Disk Hog (b) Host-State Decomposition for job before Disk Hog was injected

Fig. 17. Host-State Decomposition of outlier SALSA-REP flows for a job before and during the Disk Hog injection. Host labels have been omitted for readability.

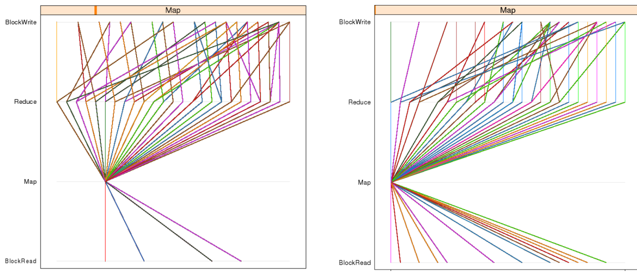
plots of the jobs in each workload, and focused on the Map-host-filtered plots. We found that in the case of HADI, the Maps on most hosts read data blocks from a small fraction of the hosts in the cluster, whereas in the case of HADI-BLK, the Maps on most hosts read data blocks from a larger fraction of hosts. This can be illustrated by looking at the plots from a typical job in the two workloads[7]. This is seen in Figure 18(a) which shows the lower density of BlockRead-to-Map flows of a typical Map-filtered plot for HADI, as compared to that in Figure 18(b) in the same plot for HADI-BLK. This can also be an artifact of the HADI-BLK having more Maps in its Map phase than in HADI for the second of its three-sequence job, as discussed in Section VII-D1. When there are more Maps in the job than Hadoop allows to run concurrently, subsequent Maps may be scheduled on hosts that do not already store the blocks for the task, leading to remote block reads that will give rise to the larger number of hosts that Maps read data from. Hence, this also shows that the HADI-BLK workload is more network-intensive when executing Maps than HADI.

VIII. Related Work

A. Distributed Tracing and Failure Diagnosis

Our view of system logs as providing a control-flow view of system execution, coupled with log messages with unique identifiers for requests/processing tasks, allows us to extract request-flows in distributed systems. Such system views have been extracted for performance debugging using other techniques e.g. using instrumented middleware or libraries to

[7]We omit showing all jobs due space constraints, and the plots exhibited similar patterns.



(a) Maps running on a given host in HADI read blocks from fewer hosts.

(b) Maps running on a given host in HADI-BLK read blocks from more hosts.

Fig. 18. Parallel-Coordinate Plots of host data exchanges filtered by Map hosts for the HADI and HADI-BLK workloads.

track requests [21], [22], [23], [24], and by disambiguating concurrent causal flows in RPC messages [7]. Instead of using added instrumentation to track flows, we extract causally-related request flows by exploiting system logs of the form amenable to our SALSA technique [10]. Also, most previous techniques for tracing have extracted distributed execution traces at the programming language level, as they worked with systems without a limited programming model unlike MR, whereas we generate views at the higher-level MR abstraction.

Previous techniques for diagnosing failures in distributed systems have largely examined multi-tier Internet services which process large numbers of requests, and which are designed to complete requests within a specified (typically low) latency, giving rise to a natural Service Level Objective (SLO) such that if the SLO is violated, a fault is present by definition. These techniques hence assume SLO violations are readily available, and given them, identify the root-causes of failures. [23], [21], [7], [25] all assume the availability of SLO violations. However, such SLOs are not readily available in MapReduce systems because MapReduce programs are designed to be large batch jobs with potentially long runtimes. Instead, we provide visualizations which give operators insight and intuition into various MR-specific aspects of program behavior, to enable them to determine if a performance problem exists, and if so, to identify and localize these faults and performance problems.

B. State-Machine Extraction from Logs

[26], [27] automatically inferred state-machine views of execution from textual logs from multi-tier J2EE web-transaction processing systems, but requires significant amounts of training data from fault-free runs to characterize normal traces for detecting failed runs, whereas we use *a priori* knowledge of MR program structure and do not need training data. Both techniques target problematic executions exhibited via state-machines of differing shapes, while we use our state-machine abstraction to encode performance characteristics (i.e. runtimes), and consider differences in edge characteristics rather than state-machine shapes. [28] constructed fine-grained finite-state automata which modeled individual program statements, whereas our abstraction is at a coarse granularity of high-level logical blocks of execution (i.e. Maps and Reduces). Such fine-grained techniques, when used with MapReduce programs, would quickly run into scalability issues. Also, [28] uses existing formal state-machine models, whereas we use state-machines as an informal abstraction, and we explicitly encode time and data volumes in our informal model.

C. Diagnosis for MapReduce Systems

X-Trace [8] was used to instrument Hadoop for collecting fine-grained trace events akin to language-level views [29] and provided summarized views. Our state-machine abstractions can also be generated by adding X-Trace instrumentation messages; the value of our work is in presenting interpretations and visualizations of these abstractions for performance debugging. Other work has focused on handling errors: [30] mined DataNode logs to detect outlier error messages; [31] traces errors to their originating components. We differ from these approaches in building a complete abstraction of MR execution. [32], [33] presented diagnosis algorithms based on the peer-comparison of durations of states (Maps, Reduces) and black-box OS performance counters counters for diagnosis. [34] synthesized results from multiple algorithms using supervised learning to enable root-cause diagnosis of previously-observed faults.

D. Visualization Tools

Artemis [35] provides a pluggable framework for distributed log collection, data analysis, and visualization. We have presented specific MR abstractions and ways to build them, and our techniques can be implemented as Artemis plugins. Other frameworks have also been built for managing the visualization and summarizing of the large amounts of data from large clusters [36], [37]. These frameworks collect general system metrics such as OS counters, and deal with the problem of scalable visualization and presentation of general system data in a way that enhances operator understanding. We have focused on building abstractions of program behavior specific to MapReduce rather than for general systems, and we use visualizations as a tool to express our abstractions.

IX. Conclusion and Future Work

We have presented a non-intrusive, log-based approach to tracing the causality of execution in MapReduce systems based on a state-machine abstraction of its behavior, and a novel way to characterize this behavior by decomposing performance along the dimensions of time, space, and volume. In addition, we have presented visualizations based on our abstractions that enable operators to gain insight into the performance of large-scale MapReduce systems for performance debugging and characterizing system behavior, and this for a real-world workload.

Initial parts of our state-machine abstraction and visualization have been available as part of the Hadoop Chukwa [38] log analysis framework, and we plan to make available our work as a part of Chukwa. We also plan to extend our work to analyze workloads consisting of multiple MR jobs where

the output of each job is then processed by the next in a workflow, such as those automatically generated by higher-level languages such as Hadoop Pig [17], and to extend our log-based abstractions and visualizations to more general forms of distributed processing such as Dryad [39].

ACKNOWLEDGMENT

The authors would like to thank U Kang and Christos Faloutsos for providing the HADI workload for our case studies, and Julio Lopez for assisting with our Yahoo! M45 experiments on earlier versions of this work. The authors would also like to thank Greg Ganger for feedback on earlier versions of this work and for suggesting the Map-Wait state, and Xinghao Pan for providing invaluable feedback and assistance with earlier versions of this work. This work was partly funded by the Defence Science & Technology Agency (Singapore) via the DSTA Overseas Undergraduate Scholarship, and sponsored in part by the National Science Foundation, via CAREER grant CCR-0238381 and grant CNS-0326453.

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] Apache Software Foundation, "Hadoop," 2007, http://hadoop.apache.org/core.
[3] ——, "Powered by Hadoop," 2009, http://wiki.apache.org/hadoop/PoweredBy.
[4] 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-pr%oduction-hadoop.html.
[5] Facebook, "Engineering @ facebook's notes: Hadoop," Jun 2008, http://www.facebook.com/note.php?note_id=16121578919.
[6] A. Murthy, "Hadoop MapReduce - Tuning and Debugging," 2008, http://tinyurl.com/c9eau2.
[7] 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*, Bolton Landing, NY, Oct 2003, pp. 74–89.
[8] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica, "X-Trace: A pervasive network tracing framework," in *USENIX Symposium on Networked Systems Design and Implementation*, Cambridge, MA, Apr 2007.
[9] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. Ganger, "Stardust: tracking activity in a distributed storage system," *SIGMETRICS Perform. Eval. Rev.*, vol. 34, no. 1, pp. 3–14, 2006.
[10] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "SALSA: Analyzing Logs as State Machines," in *USENIX Workshop on Analysis of System Logs (WASL)*, San Diego, CA, Dec 2008.
[11] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Mochi: Visual Log-Analysis Based Tools for Debugging Hadoop," in *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, San Diego, CA, May 2009.
[12] S. Ghemawat, H. Gobioff, and S. Leung, "The Google file system." in *ACM Symposium on Operating Systems Principles*, Lake George, NY, Oct 2003, pp. 29 – 43.
[13] Apache Software Foundation, "Hadoop Users' Mailing List," 2008, http://mail-archives.apache.org/mod_mbox/hadoop-core-user.
[14] J. Tan, "Log-based Approaches to Characterizing and Diagnosing MapReduce Systems," Carnegie Mellon University Master's Thesis, Tech. Rep. CMU-CS-09-143, Jul 2009.
[15] J. Tan and C. Douglas, "Add ReduceID to Shuffle ClientTrace," 2009, http://issues.apache.org/jira/browse/MAPREDUCE-479.
[16] J. Boulon, A. Konwinski, R. Qi, A. Rabkin, E. Yang, and M. Yang, "Chukwa: A Large-scale Monitoring System," in *Cloud Computing and Its Applications*, Chicago, IL, Oct 2008.
[17] Apache Software Foundation, "Pig," 2007, http://hadoop.apache.org/pig.
[18] S. Guha, "Rhipe - r and hadoop integrated processing environment," 2009, http://ml.stat.purdue.edu/rhipe/.
[19] Amazon Web Services LLC, "Amazon Elastic Compute Cloud," 2009, http://aws.amazon.com/ec2/.
[20] U. Kang, C. Tsourakakis, A. Appel, C. Faloutsos, and J. Leskovec, "Hadi: Fast diameter estimation and mining in massive graphs with hadoop," *CMU ML Tech Report CMU-ML-08-117*, 2008.
[21] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *IEEE Conference on Dependable Systems and Networks*, Bethesda, MD, Jun 2002.
[22] 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.
[23] 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.
[24] E. Koskinen and J. Jannotti, "Borderpatrol: isolating events for black-box tracing," in *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. New York, NY, USA: ACM, 2008, pp. 191–203.
[25] 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, United Kingdom, Oct 2005, pp. 105–118.
[26] L. Mariani and F. Pastore, "Automated Identification of Failure Causes in System Logs," in *International Symposium on Software Reliability Engineering (ISSRE)*, Seattle, WA, Nov 2008.
[27] G. Jiang, H. Chen, U. Ungureanu, and K. Yoshihira, "Multi-resolution Abnormal Trace Detection Using Varied-length N-grams and Automata," in *International Conference on Autonomic Computing (ICAC)*, Seattle, WA, Jun 2005.
[28] D. Lorenzoli, L. Mariani, and M. Pezze, "Inferring State-based Behavior Models," in *Workshop on Dynamic Analysis (WODA)*, Shanghai, China, May 2006.
[29] A. Konwinski, M. Zaharia, R. Katz, and I. Stoica, "X-tracing Hadoop," *Hadoop Summit*, Mar 2008.
[30] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Detecting large-scale system problems by mining console logs," in *Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct 2009.
[31] J. Lou, Q. Fu, Y. Wang, and J. Li, "Mining dependency in distributed systems through unstructured log analysis," in *2nd USENIX Workshop on Analysis of System Logs (WASL)*, Big Sky, MT, Oct 2009.
[32] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Ganesha: Black-Box Diagnosis of MapReduce Systems," in *Workshop on Hot Topics in Measurement & Modeling of Computer Systems (HotMetrics)*, Seattle, WA, Jun 2009.
[33] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Kahuna: Problem Diagnosis for MapReduce-based Cloud Computing Environments," in *12th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Osaka, Japan, Apr 2010.
[34] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, "The Blind Men and the Elephant: Piecing Together Hadoop for Diagnosis," in *IEEE International Symposium on Software Reliability Engineering (ISSRE), Industrial Track*, Mysuru, India, Nov 2009.
[35] G. Cretu-Ciocarlie, M. Budiu, and M. Goldszmidt, "Hunting for problems with artemis," in *USENIX Workshop on Analysis of System Logs*, 2008.
[36] P. MaLachlan, T. Munzner, E. Koutsofios, and S. North, "LiveRAC - Interactive Visual Exploration of System Management Time-Series Data," in *SIGCHI Conference on Human Factors in Computing Systems (CHI'08)*, Florence, Italy, Apr 2008.
[37] Q. Liao, A. Blaich, A. Striegel, and D. Thain, "ENaVis: Enterprise Network Activities Visualization," in *Large Installation System Administration Conference (LISA)*, San Diego, CA, Nov 2008.
[38] J. Tan, "SALSA state-machine extraction from Hadoop logs," 2009, http://issues.apache.org/jira/browse/CHUKWA-94.
[39] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," in *European Conference on Computer Systems (EuroSys)*, Lisbon, Portugal, Mar 2007.