Log-based Approaches to Characterizing and Diagnosing MapReduce Systems

Jiaqi Tan

CMU-CS-09-143

July 2009

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Priya Narasimhan, Chair
Gregory R. Ganger

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

This research 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.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Singapore Government, or the U.S. Government.

Keywords: MapReduce, Hadoop, Failure Diagnosis, Log analysis

Our deepest fear is not that we are inadequate. Our deepest fear is that we are powerful beyond measure. It is our light, not our darkness that most frightens us.

Abstract

MapReduce programs and systems are large-scale, highly distributed and parallel, consisting of many interdependent Map and Reduce tasks executing simultaneously on potentially large numbers of cluster nodes. They typically process large datasets and run for long durations. Thus, diagnosing failures in MapReduce programs is challenging due to their scale. This renders traditional time-based Service-Level Objectives ineffective. Hence, even detecting whether a MapReduce program is suffering from a performance problem is difficult. Tools for debugging and profiling traditional programs are not suitable for MapReduce programs, as they generate too much information at the scale of MapReduce programs, do not fully expose the distributed interdependencies, and do not expose information at the MapReduce level of abstraction. Hadoop, the open-source implementation of MapReduce, natively generates logs that record the system's execution, with low overheads. From these logs, we can extract state-machine views of Hadoop's execution, and we can synthesize these views to create a single unified, causal, distributed control-flow and data-flow view of MapReduce program behavior. This state-machine view enables us to diagnose problems in MapReduce systems. We can also generate visualizations of MapReduce programs in combinations of the time, space, and volume dimensions of their behavior that can aid users in reasoning about and debugging performance problems. We evaluate our diagnosis algorithm based on these state-machine views on synthetically injected faults on Hadoop clusters on Amazon's EC2 infrastructure. Several examples illustrate how our visualization tools were used to optimize application performance on the production M45 Hadoop cluster.

Acknowledgments

I would like to acknowledge and thank the many people whom, without their support, this work would not have been possible.

First, I would like to thank my advisor, Prof. Priya Narasimhan, for her unwavering support, trust, and belief in me and my work, and for her advice, guidance, infectious enthusiasm and unbounded energy, even when the road ahead seemed long and uncertain; and Prof. Gregory R. Ganger, for his belief in my work, and for taking the time to serve on my Thesis Committee. I would also like to thank Dr. Rajeev Gandhi, for his support, guidance, and advice on all our algorithmic, Mathematical, and Machine Learning questions, and Soila Kavulya, for sharing with us her experience and for her careful, methodical, and very hard work putting together all our surveys, experimental apparatus and data collection.

I would like to mention Eugene Marinelli, Michael Kasick, and Keith Bare, collaborators at Carnegie Mellon with whom I worked on the ASDF (Automated System for Diagnosing Failures) project, that was the precursor to this work, and thank Michael for his patience and support in helping out with technical issues, and Eugene for being a bouncing wall for ideas. I would also like to mention colleagues and friends at Carnegie Mellon whom I've worked with, Wesley Jin, Tudor Dumitras and Geeta Shroff.

I would like to acknowledge U Kang and Prof. Christos Faloutsos, for their many discussions with us and for running experiments, sharing with us their log data, and guiding us through their workloads on many occasions on the M45 cluster. Thanks also goes to Dr. Julio Lopez, for evangelizing our work, and taking the time to share his ideas with us. I would also like to mention Wittawat Tantisiriroj, Prof. Jamie Callan, and Mark Hoy, for discussions on their Hadoop workloads, and for sharing their log data and use-cases with us.

I would also like to acknowledge my manager, Mac Yang, and co-workers, especially Eric Yang, at my internship at Yahoo! in Summer 2009, for their guidance in implementing this work as part of the Hadoop Chukwa project.

I would also like to thank my scholarship mentors at DSO National Laboratories Singapore, Thiam Poh Ng and Dr. Yang Meng Tan, for their guidance and advice over the course of my study at Carnegie Mellon.

I would also like to thank Jennifer Engleson, for all her help processing our travel and other needs efficiently, Joan Digney and Karen Lindenfelser for always being so helpful and accommodating with our questions, posters, and for making my time in the Parallel Data Laboratory a very enjoyable one, and Deborah Cavlovich, for all her assistance during the MS program. I would also like to thank all my fellow student members and the faculty of the Parallel Data Laboratory for all the invaluable feedback on my work, and for the exciting research community to work in.

Special mention also goes to Xinghao Pan, collaborator and trusted friend, for over a decade of friendship, and for always being the devil's advocate and one of the harshest critics of my work.

Finally, I would like to thank my family, my father, Yew Koon Tan, my mother, Sau Wan Leong, sister Zhilin Tan, and brother-in-law Aik Kwan Liang, for their unwavering support through these years of being away from home, and for always believing in me and being proud of me.

Contents

1 Introduction **1**

1.1 MapReduce and its Applications 1

1.2 Performance Debugging of MapReduce Programs 2

1.3 Diagnosing MapReduce Systems 3

1.4 Logs as an Information Source 4

1.5 Understanding and Diagnosing MapReduce Systems from Logs 4

1.6 Key Contributions 5

2 Motivation and Problem Statement **7**

2.1 Motivation 7

2.1.1 Existing Debugging Tools for MapReduce Systems and Hadoop . 7

2.1.2 Hadoop Mailing List Survey 11

2.1.3 Hadoop Bug Survey 11

2.2 Thesis Statement 13

2.2.1 Hypothesis 13

2.2.2 Goals 13

2.2.3 Non-Goals 14

2.2.4 Assumptions 14

3 Background **17**

3.1 MapReduce and Hadoop Architecture 17

3.1.1 MapReduce and Hadoop 17
3.1.2 Logging in Hadoop 18
3.2 Chukwa Log Aggregation and Analysis Framework 19
3.2.1 Monitoring, Log Collection, and Log Aggregation 20
3.2.2 Log Processing and Analysis 21

4 Approach **23**
4.1 Abstract State-Machine Views 24
4.1.1 SALSA: Node-Local State-Machine Views 24
4.1.2 Job-Centric Data-Flows: Global System View 28
4.1.3 Realized Execution Paths: Causal Flows 29
4.2 Diagnosis 30
4.2.1 Intuition and Diagnostic Hypothesis 30
4.2.2 Synopsis of Algorithm 31
4.3 Visualization of MapReduce Behavior 31
4.3.1 Aspects of Behavior 31
4.3.2 Aggregations 32

5 Methodology **35**
5.1 Abstracting MapReduce Behavior 35
5.1.1 SALSA: Hadoop's Node-Local State-Machines 35
5.1.2 Job-Centric Data-Flows 37
5.1.3 Realized Execution Paths 39
5.2 Diagnosis 40
5.3 Visualization 42
5.3.1 "Swimlanes": Task progress in time and space. 42
5.3.2 "MIROS" plots: Data-flows in space. 46
5.3.3 REP: Volume-duration correlations. 47

6 Implementation **49**

6.1 Extracting Abstract Views of MapReduce Behavior 49
6.1.1 Intermediate Representation of MapReduce Behavior 50
6.1.2 Extracting Intermediate Representations 51
6.2 White-Box Diagnosis 54
6.3 Visualization 55
6.3.1 Offline Operation 56
6.3.2 Online Console – Chukwa HICC 56

7 Evaluation **59**
7.1 Diagnosis of Synthetic Faults 59
7.1.1 Testbed and Workload 59
7.1.2 Injected Faults 60
7.1.3 Results 60
7.2 Performance Debugging in the Wild 64
7.2.1 Testbed and Workloads 64
7.2.2 Understanding Hadoop Job Structure 64
7.2.3 Performance Optimization 65
7.2.4 Hadoop Misconfiguration 66

8 Discussion **69**
8.1 Implementation Notes 69
8.1.1 MapReduce Implementation of Job-Centric Data-Flow Extraction 69
8.2 Lessons for Logging 70
8.2.1 Log Statement Evolution and System Diagnosability 70
8.2.2 SALSA for Log Compression 71
8.2.3 Formal Software Verification and State-Machines 72

9 Related Work **75**
9.1 Log Analysis 75
9.1.1 Event-based Analysis 75

9.1.2 Request Tracing 76
9.1.3 Log-Analysis Tools 76
9.1.4 State-Machine Extraction from Logs 76
9.2 Distributed Tracing and Failure Diagnosis 77
9.3 Diagnosis for MapReduce 79
9.4 Visualization Tools 79

10 Conclusion and Future Work **83**
10.1 Conclusion 83
10.2 Future Work 84
10.2.1 Causal Backtrace 84
10.2.2 Alternative *REP* Representations 84
10.2.3 Anomaly Detection for *REP* 85

A Appendix **87**
A.1 Diagnosis Algorithm 87

Bibliography **89**

List of Figures

2.1 Screenshot of Hadoop's web console. Vertical axes show the percentage completion of each Map (top graph) or Reduce (bottom graph), and the progress of each Map or Reduce is plotted using a vertical bar. In this screenshot, the job had a single node experiencing a 50% packet-loss, which caused all reducers across the system to stall. 9

2.2 Manifestation of 415 Hadoop bugs in survey. 11

3.1 Architecture of Hadoop, showing the locations of the system logs of interest to us. 18

3.2 `log4j`-generated TaskTracker log entries. Dependencies on task execution on local and remote hosts are captured by the TaskTracker log. 19

3.3 `log4j`-generated DataNode log. Local and remote data dependencies are captured. 20

4.1 Overall approach to MapReduce program performance debugging and failure diagnosing using Hadoop's logs. 23

4.2 Illustration of (i) abstract state-machine on left, and two realized instantiations of the abstract state-machine; (ii) concurrent threads of execution in log (above) shown as two disambiguated state-machines on right. 26

4.3 Visual illustration of the intuition behind comparing probability distributions of durations of the `WriteBlock` state across DataNodes on the slave nodes. 30

5.1 States in the state-machine view of Hadoop's control-flow 36

5.2 Control-items (ellipses in red) and data-items (boxes in green) of states in the state-machine view of Hadoop's execution 37

5.3 Job-Centric Data-Flow formed by "stitching" together control-items (ellipses in red) and data-items (boxes in green) of Hadoop's execution states. The directed edges show the direction of the full causality of execution in a MapReduce program. 38

5.4 Illustration of a Realized Execution Path. The multiple copies of a particular directed edge and of a particular vertex show the vertices with states with in-degree or out-degree greater than 1, and each Realized Execution Path is formed by picking one of the multiple edges in each group of edges. 40

5.5 *Swimlanes* plot for the Sort workload. Top plot shows tasks grouped by host, bottom plot shows tasks grouped by start time. 42

5.6 *Swimlanes*: detailed states: Sort workload 44

5.7 *Swimlanes* plot for 49-node job for the Matrix-Vector Multiplication; top plot: tasks sorted by node; bottom plot: tasks sorted by time. 45

5.8 *MIROS*: Sort workload; (volumes in bytes) 46

5.9 *REP* plot for Sort workload 47

6.1 Visualization of time-series of diagnosis outcomes from the white-box diagnosis algorithm; the x-axis shows time in seconds, while the y-axis shows the diagnosis outcome for each node for each point in time; cool colors indicate high peer similarity with all other nodes; warm colors indicate low peer similarity with all other nodes and indicates the presence of a fault on the given node. 55

6.2 Screenshot of *Swimlanes* interactive visualization widget for Chukwa HICC. Screenshot on right shows mouseover details for state. 57

7.1 True-positive and False-positive ratios of diagnosis of white-box diagnosis algorithm using Map durations. 62

7.2 True-positive and False-positive ratios of diagnosis of white-box diagnosis algorithm using Reduce durations. 63

7.3 Summarized *Swimlanes* plot for RandomWriter (top) and Sort (bottom) . 65

7.4 Matrix-vector Multiplication before optimization (above), and after optimization (below) 66

7.5 *REP* plot for Matrix-Vector Multiplication 67

7.6 SleepJob with delayed socket creation (above), and without (below) . . . 68

9.1 Comparison of the “Area Chart” (left) and *Swimlanes* chart of the same job. 81

List of Tables

2.1 Common queries on users' mailing list 10

5.1 Edge weights in *REP*s for Hadoop MapReduce programs extracted from the JCDF for each causal flow. 41

7.1 Injected faults, and the reported failures that they simulate. HADOOP-xxxx represents a Hadoop bug database entry. 60

8.1 Space savings from parsed log views 72

Chapter 1

Introduction

1.1 MapReduce and its Applications

MapReduce [DG04] is a programming paradigm and framework for easily executing large-scale parallel distributed computation on large datasets in a cluster environment on large numbers of cluster nodes. Programmers specify programs as a Map function and a Reduce function, and specify an input dataset, typically stored on a distributed filesystem such as the Google Filesystem [GGL03]. The framework then automatically executes multiple copies of the Map and Reduce functions on different nodes, each processing a segment of the large dataset. This enables the processing of large-scale datasets for applications such as building inverted indexes of the World Wide Web and mining extremely large datasets that would not fit in the main memory of a single host.

While MapReduce and Cloud Computing are orthogonal, MapReduce has become a popular application that is run on Cloud Computing infrastructures. This is because Cloud Computing users can quickly scale up their MapReduce installations by renting more processing nodes from service providers. Cloud Computing [Bro09] is an amalgamation of several methods of providing IT services, such as Infrastructure-as-a-Service (IaaS), and Software-as-a-Service (SaaS), as realized by providers such as Amazon's Amazon Web Service (AWS), and their flagship product, the Amazon EC2 (Elastic Compute Cloud) [Ama09]. Instead of the traditional model of companies purchasing IT hardware and software, these large providers "rent" compute-time to companies. This provides companies the advantage of flexibility, allowing users to rapidly scale up the amount of computational resources they need during periods of peak processing demands, e.g. annual processing of customer records, without needing to purchase expensive hardware and software. Also,

users need not incur the infrastructure cost of maintaining hardware and software in periods they do not require as much compute power.

MapReduce enables programmers to easily process large datasets without the accompanying complexity of typical parallel distributed programs, such as in traditional supercomputing programming frameworks such as the Message-Passing Interface (MPI) [GLS99] and the Parallel Virtual Machine (PVM) [GBD+94]. Scaling MapReduce programs to large numbers of cluster nodes is relatively seamless as compared to in MPI or PVM, as the framework manages the scheduling of jobs across cluster nodes transparently from the application programmer. Hence, processing time of large datasets can typically be linearly reduced simply by scaling up the number of nodes used to process the dataset.

As a result, MapReduce has gained enormous popularity in recent years, and its open-source implementation, Hadoop [Apa07b], is currently used at many large companies such as Yahoo!, Facebook, Amazon, Last.fm, New York Times, and is used to process terabytes to petabytes of data daily. Applications include log processing, web-crawling and inverted-index construction, business analytics, data mining for computational biology to machine learning.

1.2 Performance Debugging of MapReduce Programs

However, it is also notoriously difficult to debug and diagnose performance problems in MapReduce programs. While programmers can quickly write MapReduce programs that run, it is often not obvious how to make these programs run faster and more efficiently.

Current tools for debugging programs are not suitable for MapReduce programs. The tools currently recommended for debugging and tuning Hadoop MapReduce programs are specific to the underlying programming language used to implement the MapReduce framework and MapReduce programs, such as `jstack` and `jprof` [Mur08] for Java. However, these tools are designed for single monolithic Java programs, and produce large amounts of data when used at scale with large MapReduce programs, rendering it difficult to process and understand the deluge of trace information. While Hadoop provides an option to sample Maps and Reduces and run these tools on only a select number of Maps and Reduces, this limits the scope of the information collected and does not yield a complete picture of program execution.

In addition, these tools do not expose program behavior in terms of the specific primitive abstractions that MapReduce provides, i.e. in terms of Maps, Reduces, and all of the accompanying behaviors of the MapReduce framework. Tools such as `jstack` and

`jprof` provide fine-grained trace data, such as the amount of time spent in given Java statements and the stack usage and contents at particular points of execution at the Java statement level of granularity. This information cannot be easily assimilated with the MapReduce level of abstraction to relate it back to Maps and Reduces. Also, these tools trace single executions within single Java Virtual Machines (JVMs), but do not correlate the execution across multiple Maps and Reduces, so that the trace information will not provide a complete picture of the entire distributed execution. In addition, these tools only trace the behavior of user-written code, i.e. Maps and Reduces, but the behavior of a MapReduce program is also highly dependent on the actions of the MapReduce framework such as task scheduling and data movement, which are not reflected in user-written code, and hence will not even be visible to these traditional debugging tools.

Our survey of posts on the Hadoop users' mailing list over a six-month period from October 2008 to April 2009 (see §2.1.2) also revealed that the most common performance-related questions users had concerned information about their programs at the MapReduce level of abstraction, in terms of Maps and Reduces, rather than at the finer level of granularity of a single line of code.

1.3 Diagnosing MapReduce Systems

Diagnosing performance problems and failures in a MapReduce cluster is difficult because of the scale of MapReduce clusters and the highly distributed nature of MapReduce programs. Currently, the only tool available for diagnosis on MapReduce clusters is the web-console provided by the framework's Master node. However, the console provides only static views of MapReduce jobs at particular instances in time, with little additional information for diagnosis. Also, the web console is cumbersome to use for large clusters with many nodes (see §2.1.1 for a detailed discussion).

In addition, Hadoop clusters are growing in size. For instance, the Yahoo! Search Webmap is a large production Hadoop application which runs on a cluster with more than 10,000 cores (approximately 1000s of nodes) and more than 5 petabytes of raw disk capacity [Net08], while Facebook has deployed multiple Hadoop clusters, with the largest consisting of over 250 nodes with over a petabyte of raw disk capacity [Fac08]. With the growing sizes of Hadoop clusters, it becomes necessary to automatically identify the sources of problems (e.g. faulty nodes) in MapReduce systems.

Previous techniques for diagnosing failures in distributed systems have largely examined multi-tier Internet services which process large numbers of low-latency requests, giving rise to a natural Service Level Objective (SLO) [KF05, CKF+02, AMW+03, CZG+05].

These techniques then identify the root-causes of failures given SLO violations. However, it is difficult to define SLOs for MapReduce programs because they are batched jobs that operate on large datasets, and are designed to run for long periods of time, as we further explain in §2.1.3. Hence, it is necessary to solve the more fundamental problem of identifying whether a fault is present in a MapReduce system, in addition to identifying the node(s) that contributed to the fault.

1.4 Logs as an Information Source

Hadoop natively generates logs of its execution by default, and these logs record anomalous system events such as error messages and exceptions, as well as regular system operation, such as what tasks are being executed (§3.1.2). These logs are generated with relatively low overhead, and are available in any default Hadoop installation. Hence, this represents a cheap, easily and widely available source of information for debugging Hadoop and its MapReduce programs. However, this information can be onerous as every Hadoop node generates its own logs.

For instance, a fairly simple benchmark Sort workload running for 850 seconds on a 5-node Hadoop cluster generates about 6.9MB of logs on each node, with each log containing over 42,000 lines of logged statements. Furthermore, to reason about system-wide, cross-node problems, the logs from each node must be collectively analyzed.

1.5 Understanding and Diagnosing MapReduce Systems from Logs

Hence, the main focus of our work is to develop techniques for understanding the behavior of Hadoop MapReduce programs by leveraging the information available in Hadoop's logs, with the goal of enabling performance debugging and failure diagnosis of MapReduce programs and Hadoop clusters.

First, we tackle the problem of turning the glut of information available in Hadoop's logs into a meaningful form for understanding Hadoop and MapReduce program behavior for diagnosis and performance debugging. We develop a novel technique for log-analysis known as SALSA (SALSA: Analyzing Logs as State Machines, [TPK+08]), which when applied to Hadoop, allows us to abstract Hadoop's execution on each of its nodes as state-machines and to extract these state-machine views of its behavior from the logs of each

Hadoop node.

The MapReduce framework affects program performance at the macro-scale through task scheduling and data distribution. This macro behavior is hard to infer from low-level language views because of the glut of detail and because this behavior results from the framework outside of user code. For effective debugging, tools must expose MapReduce-specific abstractions. This motivated us to *capture Hadoop distributed data- and execution-related behavior that impacts MapReduce performance*.

We develop a technique called Mochi [TPK+09], which exposes these MapReduce abstractions by extracting Hadoop's behavior in terms of MapReduce abstractions. We correlate the state-machine views of Hadoop's execution across (i) multiple nodes, and (ii) between the execution and filesystem layers, to build an abstraction of Hadoop's system-wide behavior. Doing so enables us to extract full-causal paths through Hadoop's execution.

Finally, given the scale (number of nodes, tasks, interactions, durations) of Hadoop's programs, there is also *a need to visualize a program's distributed execution* to support debugging and to make it easier for users to detect any deviations from expected program behavior/performance, and to automatically identify nodes contributing to a failure.

Thus, we use the SALSA-extracted state-machine views of Hadoop for automatically diagnosing failures in Hadoop (§4.2), and we also build visualizations of MapReduce program behavior (§5.3) to aid programmers in performance debugging of MapReduce programs.

1.6 Key Contributions

Our contributions are:

1. A technique for log analysis that extracts state-machine views of a system's behavior (§4.1.1)

2. A technique for correlating state-machine views across multiple nodes and between data and execution elements in a system to build a conjoined distributed data-flow and distributed control-flow model of its execution (Job-Centric Data-Flow, §4.1.2)

3. An algorithm for extracting these conjoined distributed data-flow and control-flow views for MapReduce systems (§5.1.2, 5.1.3)

4. An algorithm for diagnosing failures in a MapReduce system given its state-machine view (§4.2)

5. Visualizations of the behavior of a MapReduce program (§5.3)

Chapter 2

Motivation and Problem Statement

2.1 Motivation

To motivate the need for improved tools for performance debugging of MapReduce programs, we examine the existing tools available to Hadoop users. We show that these tools are inadequate as they do not expose MapReduce-specific program behavior, and we describe where they can be improved to address the needs of users. We also survey posts by users to the Hadoop users' mailing list to motivate the need for better debugging tools which present MapReduce-specific information. In addition, we survey the bugs reported in Hadoop to motivate the need to automatically diagnose failures, in particular performance faults, in Hadoop.

2.1.1 Existing Debugging Tools for MapReduce Systems and Hadoop

The main tools currently available to Hadoop users for debugging MapReduce programs are traditional language-based debuggers for the underlying programming language for writing MapReduce programs, such as Java in the case of Hadoop, the web console of the Hadoop Master node JobTracker daemon, and the logs natively generated by each of Hadoop's nodes which record system activity. We describe each of these and show where they fall short of the debugging needs of MapReduce programmers today.

Traditional Language-based Debuggers

The main tools used for performance debugging of MapReduce programs on Hadoop are traditional language-based debuggers such as `jstack` and `hprof` which ship with the Java SDK [Mur08].

`jstack` allows the inspection of the stack of Hadoop programs written in Java, while `jprof` is a profiler that enables the collection of various program metrics such as thread metrics, cycle counts, instructions counts, and heap usage. However, these profilers trace single executions in great detail, and MapReduce programs consist of many Maps and Reduces running in parallel. Thus, tracing every Map and Reduce in the program would generate onerous amounts of information; although Hadoop allows the selective enabling of profilers for a given number of Maps or Reduces, sampling reduces the coverage of data collected.

Such generic language-based profiling tools do not provide direct insight into MapReduce-specific parameters that affect performance, so that users would need to make their own inference to relate the profiling data with MapReduce parameters–this would require significant user experience with MapReduce. While new users may have trouble doing such correlation, even experienced users would find the exercise onerous due to the overwhelming amount of information generated from programs running at large scales, e.g. with many Maps and Reduces running on many cluster nodes. In addition, a significant factor which affects MapReduce program behavior is the behavior of the MapReduce framework itself, such as task scheduling and data placement. However, traditional language debuggers allow the observation of only user-written Map and Reduce code, whereas the framework behavior occurs in system-level Hadoop code rather than in user-written Map or Reduce code, so that traditional debuggers would not be able to capture this behavior.

In addition, traditional profiling tools focus on latencies in program components, but the performance of MapReduce programs is sensitive to the distribution of the amounts of data processed as well–given identical Map and Reduce operations, differences in runtimes could arise from either data-skews (some Maps or Reduces having more data to process than others) or underlying hardware failures. Also, such profiling tools trace only single execution flows but do not correlate them; although recent tools [TSS+06, FPK+07] perform causal tracing of execution across distributed systems, these tools produce fine-grained views of the execution that may overwhelm users with information.

As a result, users can directly observe MapReduce programs only at the lower-level of the specific language used (i.e. Java for Hadoop), but not at the higher level of the MapReduce-specific abstractions such as Maps and Reduces that the MapReduce framework imposes on the programs. It is difficult to infer the higher-level view from the lower-

level view due to the glut of detail from low-level profiling. This presents a major obstacle to optimizing MapReduce programs, because program parameters change high-level MapReduce-characteristics of the programs, and the distribution of data to Maps and Reduces is performed by the framework outside the scope of user code.

Hadoop Web Console

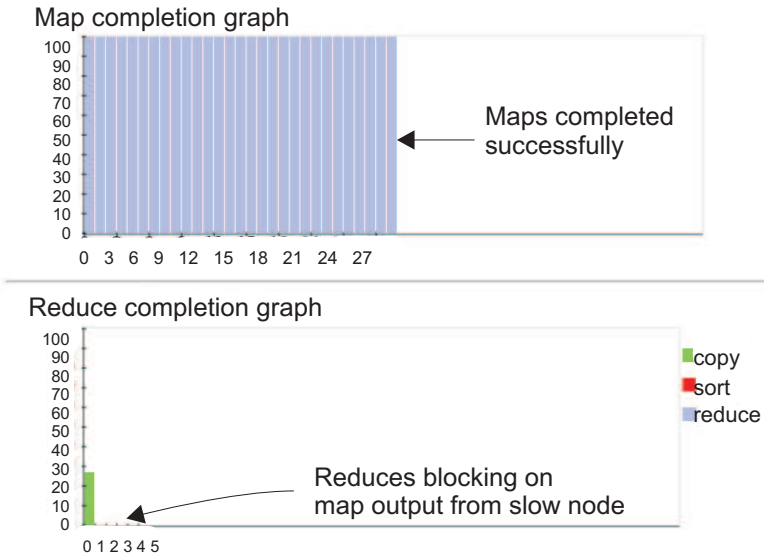

Figure 2.1: Screenshot of Hadoop's web console. Vertical axes show the percentage completion of each Map (top graph) or Reduce (bottom graph), and the progress of each Map or Reduce is plotted using a vertical bar. In this screenshot, the job had a single node experiencing a 50% packet-loss, which caused all reducers across the system to stall.

Currently, the only built-in Hadoop user tool for monitoring the performance of a Hadoop job is the web interface exposed by the JobTracker; while Hadoop also exposes interfaces such as for collecting metrics using the Java Management Extensions (JMX), these are programmatic interfaces not designed for general users and require administrator access.

This web console reports, for a given MapReduce job, the duration of each Map and Reduce task, the number of Maps and Reduces completed so far, and the cluster nodes being used for the execution of the job. The web console is relatively simple, providing only static instantaneous information about the completion times of tasks. Figure 2.1 shows a screenshot of the web console's display of the task completion status at a particular

instant in time. The web console also provides links to a status page for each node, from which users can follow links to the directory containing the Hadoop's activity logs for that node.

The web console is insufficient for debugging MapReduce programs for various reasons. First, it presents only a static, instantaneous snapshot of the behavior of a MapReduce program; users must continuously monitor the web console to study the dynamic behavior of the program as it executes. Second, the web console does not perform any aggregation of raw data or provide alternative views which may present additional insights to users (see §4.3.2 for examples types of aggregation of Hadoop's behavior along various dimensions). Third, the web console presents only raw data without suggesting nor highlighting where a problem may be present. Thus, users need to manually investigate any problem by manually inspecting the data at individual nodes, which is not scalable for large clusters.

Hadoop Logs

In its default configuration, each Hadoop node natively generates its own local logs which record activities occurring on that node, such as tasks executed and errors and Exceptions generated. These logs are described in more detail in §3.1.2. These are readily accessible to Hadoop users, and users frequently search through logs for Exceptions and error messages to aid debugging. However, these logs are large, scaling linearly in size with the number of nodes in the cluster, and can quickly become unwieldy (see sizes of logs generated in §8.2.2). Hence, manually inspecting logs quickly becomes overwhelming, and more scalable representations of the rich information captured in the logs is necessary.

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 2.1: Common queries on users' mailing list

2.1.2 Hadoop Mailing List Survey

We studied messages posted on the Hadoop users' mailing list [Apa08] 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 posts, we found approximately 30 relevant posts. We list common types of questions in Table 2.1, a simple categorization of the queries, and the fraction of these posts they made up (some posts had multiple categories). All the user questions focused on MapReduce-specific aspects of program behavior, and we found that generally the answers tended to be from experienced users based on heuristically-selected "magic-numbers" that might not work in different environments. Also, many questions were on dynamic MapReduce-specific behavior, such as the relationships in time (e.g. 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. We found a user query about tools for profiling the performance of MapReduce programs, and the main answer was to use tools for profiling Java, but this runs into the problems we described in §2.1.1.

2.1.3 Hadoop Bug Survey

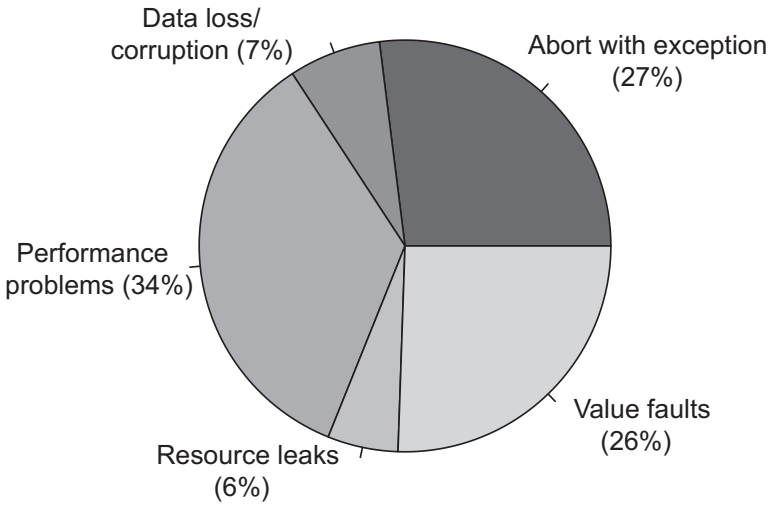

Figure 2.2: Manifestation of 415 Hadoop bugs in survey.

We conducted a survey of bugs in Hadoop reported via its Issue Tracker [Apa06] over

a two-year period from February 2007 to February 2009. The survey covered 415 closed (resolved or otherwise declared not-resolvable) bugs in Hadoop's MapReduce and HDFS (Hadoop Distributed FileSystem) components. We classified these bugs by their manifestation, as shown in Figure 2.2. Given our goal of remaining transparent to Hadoop and amenable to production environments (§2.2.2), we have chosen to focus on faults with manifestations that can be observed without code-level instrumentation. Of these bugs, 34% were performance problems, in which Hadoop ran slower than expected, 6% were resource leaks, such as unusually high CPU or memory usage, and 27% were aborts with exceptions. We omit value faults (26%) and data corruption/loss (7%) as these require semantic knowledge from the application programmer and hence cannot be detected without invasive instrumentation and programmer input.

Of these faults, performance problems and resource leaks can lead to degraded performance in which programs run slower than they otherwise would without the presence of these faults; it is important to diagnose these faults to improve the performance of MapReduce programs and reduce their runtimes to improve their efficiency. Diagnosing degraded performance in systems with long-running, batched jobs such as MapReduce is difficult because it is difficult to impose, *a priori*, an expectation of the runtime of the job, as opposed to systems which require and are designed around a specific latency target, e.g. multi-tier web-request processing systems such as J2EE-based web services servicing web clients, so that there are no easy Service Level Objectives (SLOs) that can be imposed to identify when the system is failing. This renders it difficult to apply existing failure diagnosis techniques, which assume that an SLO has been violated, and seek the root-cause of that violation (§9.2).

While aborts of individual map or reduce tasks in MapReduce programs can be tolerated by Hadoop's fault-tolerance mechanism of re-executing failed tasks, being able to diagnose these faults provides a window of opportunity for terminating these tasks earlier if they have been diagnosed to be faulty but have not yet been aborted, reducing the time taken for the job to complete overall. In addition, these faults that we target comprise a significant 67% of reported bugs in Hadoop.

SALSA's state-machine views of Hadoop's behavior can be used to diagnose these categories of faults based on our diagnostic hypothesis (§4.2.1), borne out by observation (§7.1), that Hadoop's slave nodes tend to behave similarly from the perspective of the runtime of each of the states in their state-machine view. This hypothesis enables us to detect nodes with states whose durations differ significantly from other nodes, and diagnose them as being faulty.

2.2 Thesis Statement

State-machine views of MapReduce logs enable the extraction of distributed, causal, control- and data-flow that facilitate problem diagnosis and visualization.

Our main approach is to build a novel abstraction of MapReduce behavior by utilizing information found in Hadoop's natively generated system activity logs. The key building blocks of this abstraction are distributed control-flows, distributed data-flows, and conjoined distributed data- and control-flows, which we term *Job-Centric Data-Flows* (JCDF) (§4.1.2). We then present applications of these abstractions in automatically diagnosing failures in Hadoop, and in visualizing Hadoop's behavior to aid users in debugging performance problems.

2.2.1 Hypothesis

We hypothesize that the coarse-grained control- and data-flow views of the execution of a MapReduce system, as provided by our proposed novel abstraction of MapReduce behavior, enables the diagnosis of performance problems in MapReduce systems.

We validate this hypothesis in §7.1 for the automated diagnosis of synthetically-injected performance problems, and we validate the hypothesis for the detecting of problems using visualizations of these coarse-grained views in real-world environments in §7.2.

2.2.2 Goals

The goals of this work are:

1. To expose MapReduce-specific behavior that results from the MapReduce framework's automatic execution and affects Hadoop's program performance, such as Maps and Reduces are executed and on which nodes, and where data inputs and outputs flow from/to, and from/to which Maps and Reduces.

2. To construct an abstraction of MapReduce behavior that accounts for both the coarse-grained behavior of user Map and Reduce code, as well as the automatic behaviors of the MapReduce framework outside of the control of user code. The abstraction must take into account both aspects of behavior inside and outside of the influence of user code to allow the two to be reasoned about in unison.

3. 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.

4. To automatically diagnose performance problems with low false-positives. Our automated diagnosis algorithm based on SALSA's state-machine views (§4.2) must have a low false-positive rate, and a low false-negative rate (defined in §7.1.3), indicting nodes if and only if they are truly the source of the fault, in order for the diagnosis to be useful to end-users.

5. To remain transparent to Hadoop, without requiring additional instrumentation of Hadoop, by either administrators to the framework, or by users in their own MapReduce code. This greatly eases the use of our techniques for diagnosing problems and visualizing the behavior of Hadoop MapReduce programs running on commodity, default installations of Hadoop.

2.2.3 Non-Goals

The focus of our abstractions and visualizations are on exposing MapReduce-specific aspects of user program behavior, rather than behavior within individual Maps and Reduces. Thus, the execution specifics and correctness of code within a user's Map or Reduce is outside of our scope, as these require finer-grained information which can be generated using the language-specific tools described in §2.1.1.

Our diagnosis algorithm (§4.2) uses coarse-grained per-task information, and together with our goal of remaining transparent to Hadoop and not requiring additional inserted instrumentation, this limits the granularity to which we can perform diagnosis, so that fine-grained diagnosis at the level of the line of code which is the root-cause of the problem is outside of our scope.

Similarly, our visualizations of MapReduce behavior does not discover the root-cause of performance problems, but aids in the process by enables users to gain useful insights that they can exploit to discover the root-cause of problems.

2.2.4 Assumptions

We assume that the logs generated by each Hadoop node contain correct information about the execution activity on that node. In particular Hadoop versions (prior to 0.21), we also

assume that timestamps across cluster nodes are synchronized, however, in newer versions with new logging statements, this assumption is no longer required, as discussed in §8.2.1. In addition, we also assume that *a priori* knowledge about the system's execution (Hadoop in the case of our work) is available for the SALSA extraction of state-machine views of the system's execution (see §4.1.1), and that this knowledge is correct.

Chapter 3

Background

3.1 MapReduce and Hadoop Architecture

3.1.1 MapReduce and Hadoop

MapReduce [DG04] is a framework that enables distributed, data-intensive, parallel applications by enabling a job described as a Map and a Reduce be decomposed into multiple copies of Map and Reduce tasks and a massive data-set into smaller partitions, such that each task processes a different partition in parallel. The framework also manages the inputs and outputs of Maps and Reduces, and transparently performs the Shuffle stage, which moves the outputs of Maps to Reduces. Hadoop is an open-source, Java implementation of MapReduce, and MapReduce programs consist of Map and Reduce tasks written as Java classes.

Hadoop uses the Hadoop Distributed File System (HDFS), an implementation of the Google Filesystem (GFS) [GGL03], to share data amongst the distributed tasks in the system. HDFS splits and stores files as fixed-size blocks (except for the last block of each file). One key difference between HDFS is that GFS supports file appends and multiple concurrent appenders per file, while HDFS currently does not support file appends at the time of this dissertation's writing.

Hadoop uses a master-slave architecture, as shown in Figure 3.1, with a unique master host and multiple slave hosts. The master host typically runs two daemons: (1) the JobTracker that schedules and manages all of the tasks belonging to a running job, and provides fault-tolerance by detecting node availability using periodic heartbeats and by re-executing failed tasks and tasks on failed nodes; and (2) the NameNode that manages

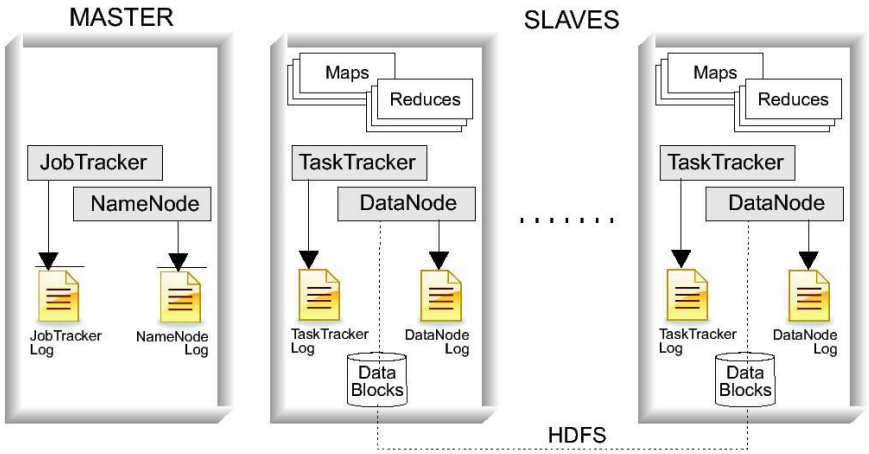

Figure 3.1: Architecture of Hadoop, showing the locations of the system logs of interest to us.

the HDFS namespace by providing a filename-to-block mapping, and regulates access to files by clients (i.e., the executing tasks). Each slave host runs two daemons: (1) the TaskTracker that launches tasks on its host, as directed by the JobTracker; the TaskTracker also tracks the progress of each task on its host; and (2) the DataNode that serves data blocks from its local disk to HDFS clients.

Each daemon on each Hadoop node (both master and slave nodes) natively generates logs which record its execution activities, as well as error messages and Java Exceptions. The logging architecture in Hadoop is described in detail next.

3.1.2 Logging in Hadoop

Hadoop uses the Java-based `log4j` logging utility to capture logs of Hadoop's execution on every host. `log4j` provides an interface to a configurable, standardized logging facility across various components of a large application. This enable programmers to easily standardize the formats of their log messages, and configure the format of log messages without modifying their source code or recompiling their application. `log4j` also enables the configuration of log sinks and provides convenient utilities such as the periodic rolling of logs, and also allows programmers to specify the severity of log messages so that application users can approximately specify the desired level of log message verbosity.

By default, Hadoop's `log4j` configuration generates a separate log for each of the daemons– the JobTracker, NameNode, TaskTracker and DataNode; these logs are stored

Hadoop source-code

```
LOG.info("LaunchTaskAction: " + t.getTaskId());
LOG.info(reduceId + " Copying " + loc.getMapTaskId()
    + " output from " + loc.getHost() + ".");
```

⇓ TaskTracker log

```
2008-08-23 17:12:32,466 INFO
    org.apache.hadoop.mapred.TaskTracker:
    LaunchTaskAction: task_0001_m_000003_0
2008-08-23 17:13:22,450 INFO
    org.apache.hadoop.mapred.TaskRunner:
    task_0001_r_000002_0 Copying
    task_0001_m_000001_0 output from fp30.pdl.cmu.local
```

Figure 3.2: `log4j`-generated TaskTracker log entries. Dependencies on task execution on local and remote hosts are captured by the TaskTracker log.

on the local filesystem of the executing daemon. In addition, Hadoop's use of `log4j` inserts the class name of the reporting class that generated the log message.

Typically, system logs (such as `syslogs`) record events in the system, as well as error messages and exceptions. However, Hadoop's log messages provide more information than typical system logs. They record the execution activities on each daemon, such as block reads and writes on DataNodes, and the beginning and end of Map and Reduce task executions on TaskTrackers. Hadoop's default `log4j` configuration generates time-stamped log entries with a specific format. Figure 3.2 shows a snippet of a TaskTracker log, and Figure 3.3 a snippet of a DataNode log.

3.2 Chukwa Log Aggregation and Analysis Framework

Next, we describe the Chukwa [BKQ+08] log aggregation framework. We implement our SALSA state-machine extraction and Mochi JCDF construction, and the accompanying visualization widgets and tools as extensions to Chukwa, which we describe in §6.1.2.

Chukwa is a framework for large-scale monitoring, log collection and aggregation for collecting logs and monitored data from large clusters, and to store them in a scalable fashion, providing automated aggregation to downsample metrics and log data collected from earlier intervals. A typical deployment of Chukwa would involve Adaptors and Agent daemons (§3.2.1) running on each monitored node, and a dedicated monitoring and collection cluster which runs Collector daemons and a Hadoop cluster, whose HDFS instance

Hadoop source-code

```
LOG.debug("Number of active connections is: "+xceiverCount);
LOG.info("Received block " + b + " from " +
    s.getInetAddress() + " and mirrored to "
    + mirrorTarget);
LOG.info("Served block " + b + " to " + s.getInetAddress());
```

⇓ DataNode log

```
2008-08-25 16:24:12,603 INFO
    org.apache.hadoop.dfs.DataNode:
    Number of active connections is: 1
2008-08-25 16:24:12,611 INFO
    org.apache.hadoop.dfs.DataNode:
    Received block blk_8410448073201003521 from
    /172.19.145.131 and mirrored to
    /172.19.145.139:50010
2008-08-25 16:24:13,855 INFO
    org.apache.hadoop.dfs.DataNode:
    Served block blk_2709732651136341108 to
    /172.19.145.131
```

Figure 3.3: `log4j`-generated DataNode log. Local and remote data dependencies are captured.

provides the storage back-end for Chukwa-collected logs. Log aggregation and analysis tasks are then executed as MapReduce jobs on this Hadoop cluster. We distinguish the clusters involved as the Chukwa Hadoop cluster, which stores monitored data and runs log aggregation jobs, and the monitored Hadoop cluster, which Chukwa monitors, but does not run on. The same Hadoop cluster that is being monitored can also be used to monitor itself. However, this requires the Hadoop JobTracker to support multiple job queues to avoid having the Demux (§3.2.2) MapReduce job hog the job queue with Demux jobs submitted faster than they can be completed.

3.2.1 Monitoring, Log Collection, and Log Aggregation

The Chukwa monitoring and log collection architecture consists of three components: the Adaptor, the Agent, and the Collector. The Adaptor is a per-source driver which interfaces with the actual data source on the monitored node; for instance, Chukwa ships with Adaptors which perform a periodic `tail` on plain text files to collect new log lines, and Adaptors which query the `proc` interface of Linux hosts for OS metrics. The Agent is a per-host daemon process which manages Adaptors by sending new metrics and log data to the Collectors using HTTP Post. The Collector is a daemon which writes received met-

ric data to the HDFS instance on the Chukwa Hadoop cluster, and the Collector manages HDFS's inefficiency with small files by collating log and metrics data in open files until they are sufficiently large before the files are closed. These Collector-written files are known as data sink files, and they contain raw log data, with each record corresponding to a single item of sampling data (i.e. single log message, or single sample of numerical metrics), and tagged with a record type to select the appropriate post-processor for that record. Agents send data to Collectors periodically in the time-scale of seconds, while Collectors close their data sink files, making them available for post-processing, in the time-scale of minutes.

3.2.2 Log Processing and Analysis

The main log data post-processing phase currently available in Chukwa is the Demux (short for Demultiplexer), which is a MapReduce job for Hadoop written in Java. The Demux demultiplexes the different types of log and metric data in the data sink files, and for each record, invokes the appropriate Mapper and Reducer classes based on the record type tagged with the record by the source Adaptor. Chukwa ships with a collection of commonly used post-processors for data such as OS metrics from `proc` and Hadoop's logs to split log messages into timestamps, logging severity level, and log message[1]. This semi-structured data is then currently loaded into a MySQL database for visualization.

Chukwa currently does not provide any additional log-processing and analysis tools in addition to the simple loading of metrics into structured storage.

In the context of Chukwa-collected log data, SALSA and Mochi abstractions (state-machine views, Job-Centric Data-Flows) can be generated by processing the output of the Demux phase. SALSA and Mochi inject semantic knowledge of Hadoop to build rich views of Hadoop's behavior based on the data collected by Chukwa. In §6.1.2 we implement SALSA and Mochi view construction as MapReduce jobs which directly leverage data collected by Chukwa, and can be run on the Chukwa Hadoop cluster to process the Chukwa-collected data to generate the SALSA and Mochi views.

[1]SALSA (§4.1.1) can leverage this semi-structured data to generate state-machine views, although this semi-structured representation merely tokenizes the log messages, and falls far short of SALSA's richer view.

Chapter 4

Approach

We describe our overall approach to characterizing the behavior of Hadoop MapReduce programs for performance debugging and failure diagnosis by: (i) constructing abstract views of MapReduce behavior from Hadoop's logs, (ii) diagnosing failures from these abstract views, and (iii) visualizing the behavior of MapReduce programs from our abstract views.

This overall approach is as summarized in Figure 4.1.

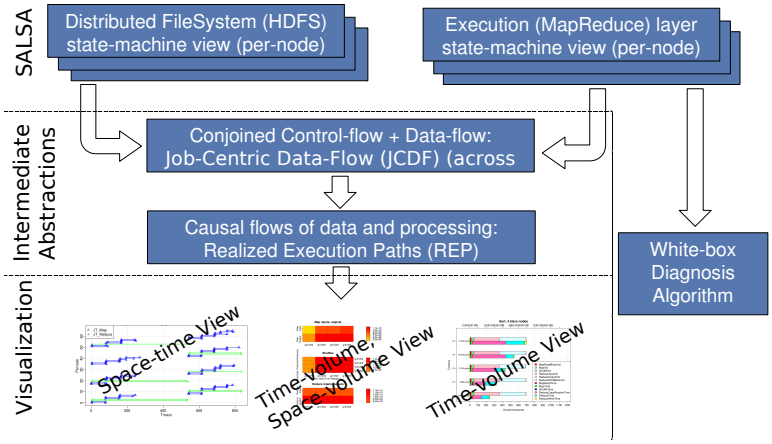

Figure 4.1: Overall approach to MapReduce program performance debugging and failure diagnosing using Hadoop's logs.

4.1 Abstract State-Machine Views

4.1.1 SALSA: Node-Local State-Machine Views

SALSA is a general technique for analyzing logs which describe system execution in terms of control-flows and data-flows in the system, and for extracting these potentially distributed[1] control-flows and data-flows from the system's logs, given *a priori* knowledge about the structure of the system's execution. Such *a priori* knowledge comprises the possible states of execution in the system, the orders in which they execute, and the tokens in log statements corresponding to the execution of these states. SALSA does not infer the correct state-machine model of a system's execution (see §8.2.3 for a discussion on how our work differs from formal verification using state-machine models). Rather, given the state-machine model of a system's execution, SALSA extracts the runtime properties (duration of state execution, data-flows between states) of the state-machine as the system executes to describe the system's control-flows and data-flows. Concretely, control-flows refer to the orders in which execution activities occur, and the times at which they begin and end (and consequently the duration for which the system was executing the particular activity); data-flows can be considered to be implicit control-flows, and they refer to explicit data-items being transmitted from one execution activity to another.

Illustrative Example

To describe SALSA's high-level operation, consider a distributed system with many producers, $P1, P2, ...$, and many consumers, $C1, C2, ...$ Many producers and consumers can be running on any host at any point in time. Consider one execution trace of two tasks, $P1$ and $C1$ on a host X (and task $P2$ on host Y) as captured by a sequence of time-stamped log entries at host X:

```
[t1] Begin Task P1
[t2] Begin Task C1
[t3] Task P1 does some work
[t4] Task C1 waits for data from P1 and P2
[t5] Task P1 produces data
[t6] Task C1 consumes data from P1 on host X
[t7] Task P1 ends
[t8] Task C1 consumes data from P2 on host Y
[t9] Task C1 ends
   :
```

[1] If the control-flows and data-flows are distributed in nature and pass from one node to another in a distributed system, the unified control- and data-flow can be reconstructed as a Job-Centric Data-Flow (§4.1.2)

From the log, it is clear that the executions (control-flows) of $P1$ and $C1$ interleave on host X. It is also clear that the log captures a data-flow for $C1$ with $P1$ and $P2$.

SALSA interprets this log of events/activities as a sequence of *states*. For example, SALSA considers the period $[t1, t6]$ to represent the duration of state $P1$ (where a state has well-defined entry and exit points corresponding to the start and the end, respectively, of task $P1$). Other states that can be derived from this log include the state $C1$, the data-consume state for $C1$ (the period during which $C1$ is consuming data from its producers, $P1$ and $P2$), etc. Based on these derived state-machines (in this case, one for $P1$ and another for $C1$), SALSA can derive interesting statistics, such as the durations of states. SALSA can then compare these statistics and the sequences of states across hosts in the system, specifically to construct Job-Centric Data-Flows (§4.1.2).

In addition, SALSA can extract data-flow models, e.g., the fact that $P1$ depends on data from its local host, X, as well as a remote host, Y. The data-flow model can be useful to visualize and examine any data-flow bottlenecks or dependencies that can cause failures to escalate across hosts.

Control-flow

Generalizing from the above example, considering each processing activity of interest in the target system as a state, SALSA is able to infer the execution of the state if two (timestamped) log messages appear for the state –one indicating the beginning the state's execution, and one indicating the end of its execution. To build a state-machine view, SALSA requires prior knowledge (expert input by a knowledgeable user) about the possible states of execution and the orders of execution of the different types of states (i.e. which states of execution are possible, and what is the normal sequence of their execution).

Also, if multiple concurrent threads of execution in the system are possible, and each thread executes its own independent state machine, then each state also requires a unique identifier of its thread of execution. Hence, SALSA needs a thread identifier (for concurrent systems), and two distinct types of log messages, one indicating the beginning and one indicating the end, to identify the execution of each type of state in the state-machine view. When control passes from one state to another along the same causal flow, this thread identifier can also take the form of a unique identifier shared by the preceding and subsequent state in the sequence, e.g. if control always flows from state S_A to state S_B, then as long as S_A and S_B both emit log messages with the same identifier along the same thread, and this identifier is unique among all pairs of $S_A \rightarrow S_B$ transitions, then the state-machine can be reconstructed. This is illustrated in Figure 4.2, where there are two concurrent threads of execution, each with its unique thread identifier emitted in the log message, and these two

concurrent threads of execution logged in the same same log can be disambiguated given the thread identifiers.

```
[T= 0] (Thread 1) State A (ID 1) Begin
[T= 0] (Thread 2) State A (ID 2) Begin
[T=10] (Thread 2) State A (ID 2) End
[T=10] (Thread 2) State B (ID 1) Begin
[T=12] (Thread 1) State A (ID 1) End
[T=12] (Thread 1) State B (ID 2) Begin
[T=15] (Thread 2) State B (ID 1) End
[T=16] (Thread 1) State B (ID 2) End
```

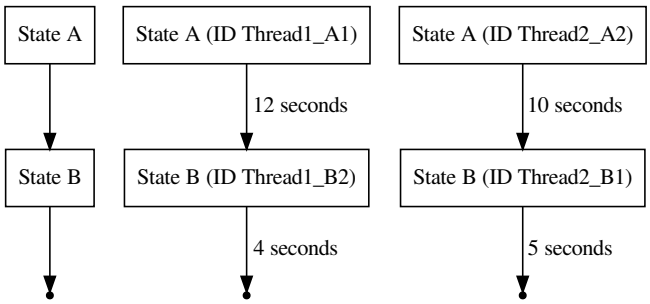

Figure 4.2: Illustration of (i) abstract state-machine on left, and two realized instantiations of the abstract state-machine; (ii) concurrent threads of execution in log (above) shown as two disambiguated state-machines on right.

In effect, SALSA's value lies in extracting all actual realized executions, or instantiations, of an abstract, "template" state-machine representation of its behavior, and filling in actual latencies along edges in the state-machine. This is as illustrated in Figure 4.2. While this precludes SALSA from detecting wrong or invalid sequences of execution, such techniques belong in the realm of formal software verification, while our target failures (§2.1.3) are performance faults rather than semantically wrong executions. We discuss this issue further in §8.2.3.

Data-flow

For each explicit data-flow, i.e. transfer of an explicit data item from one processing activity to another, SALSA requires knowledge about the source and destination hosts, source and destination processing activities, the volume of data transmitted, and the duration taken for the transfer of that data item, to fully define the distributed data-flow that occurred.

This information can either be in the form of prior expert knowledge, explicitly encoded in log messages, or inferred via external mechanisms. For instance, if the hostnames of source and destination hosts are unavailable, but information about the source and destination processing activities, as well as the hosts they executed on, are available, then the source and destination hostnames can be inferred.

Also, the information can be logged on different nodes, as long as they can be causally correlated. For instance, all of the information can be recorded on only the log of the source host of the transaction. However, if information is logged across multiple hosts (i.e. partially logged on source and partially logged on destination hosts), then it must be possible for the information to be correlated in an unambiguous way. We describe the limitations if this requirement is not met, and how we worked around this limitation in particular Hadoop versions, in §5.1.2.

Node-local State-Machine Views

In [TPK+08], SALSA extracted node-local views of the state- machine views of the execution on each node in a distributed system (as was demonstrated with Hadoop). Each node's log was independently analyzed to extract the states that executed on that particular node's control-flow, and distributed control-flows, where control passed from one node to another, were captured on the particular node that they were logged, e.g. only on the source where control originated from if the distributed flow was logged on the source node. Similarly, each node's log was analyzed to extract distributed data-flows, as viewed from the node on which they were logged.

Hence, SALSA's log analysis extracts node-local state-machine views of a distributed control- and data-flow, and extracts the control-flows and data-flows separately, and on a per-node basis. While these views show the activity on each node, they do not, on their own, reflect the full causality of processing in the system. Instead, they form the building blocks for the full distributed unified control-flow and data-flow, which we describe next in §4.1.2.

4.1.2 Job-Centric Data-Flows: Global System View

Next, we describe the full causality of processing in a MapReduce system, i.e. all of the data and processing pathways that a MapReduce program goes through. We call this abstraction a Job-Centric Data-Flow, and each JCDF is an abstraction of the execution of a single program.

The abstraction of these processing pathways is built on SALSA's node-local state-machine views. We call this view of all the data and processing pathways a Job-Centric Data-Flow, because we consider both the processing that makes up the job, and the data-flows necessary for the input of data to and output of data from the job, in the context of the actual job processing.

Intuition

In a data-intensive programming model such as MapReduce, both processing activities, such as Maps and Reduces in MapReduce, and the explicit movement of data items, such as the reading of inputs from and writing of outputs to the distributed filesystem (HDFS in the case of Hadoop) constitute significant consumption of system resources in terms of both time taken for data processing and data movement, and storage in terms of volatile storage (i.e main memory) on processing nodes and persistent storage (i.e. disk drives) on storage nodes.

In addition, when such programming models are realized in a distributed fashion across multiple nodes, the spatial characteristics of the processing also become important, because imbalances in data volumes or processing loads on nodes can lead to inefficient program execution. For instance, in the case of nodes with homogeneous processing capabilities, if a particular node needs to process more data than another, then the overall job completion time can be reduced by redistributing the load more evenly.

Hence, the abstraction of a Job-Centric Data-Flow aims to capture the overall execution of a data-intensive application such as a MapReduce application by representing the program in terms of the times taken to process data, the volumes of data processed, and the spatial characteristics of where data is processed, and how much.

JCDF: Processing Graph

First, we introduce the JCDF concretely as a directed graph, with processing items forming its vertices. These processing items can be data items (i.e. explicit data items such

as a block in HDFS in Hadoop) or control items (i.e. processing states as in SALSA's state-machine views). This reflects the property of data-intensive applications that both data and control items lie on the critical path [HPG02] of processing. These vertices reflect only logical divisions into data items or control items, and do not reflect physical host boundaries on which the items were stored (data items) or executed (control items). Nonetheless, this information is available and we store this information with the meta-data associated with each vertex, and we leave abstractions that explicitly account for these host boundaries to future work.

Next, the directed edges in the JCDF represent causality from one processing item to the next. For instance, in MapReduce (see §5.1.2 for the complete example), when a Map reads a block from HDFS as its input, in the JCDF, we represent this relationship with a vertex representing the Map as it is a control item, a vertex representing the input data block as it is a data item, and we add a directed edge from the input data block to the Map as the Map depends on data from the input block.

Then, we annotate each edge with two values: the first is the volume of data processed along that edge, i.e. the size of the input block read by the Map in our example above, and the second is the time taken along that edge, i.e. the time taken for the Map to read the input block. Alternatively, the JCDF can also be thought of as two directed graphs, with the first having all edges with weights representing the volumes of data processed, and the second having all edges representing the times taken for the processing.

Hence, in a system with a large number of interdependencies between processing stages, the JCDF for that program would be a complex, dense, directed graph. We describe the specific JCDF for Hadoop MapReduce applications in §5.1.2.

4.1.3 Realized Execution Paths: Causal Flows

Finally, each path beginning from a vertex with zero in-degree and ending at a vertex with zero out-degree in the Job-Centric Data-Flow directed graph represents a single thread of causality in the system. Hence, these are end-to-end paths in the JCDF graph, and we call these single threads of executions Realized Execution Paths (REP), because they represent a single persistent flow through the system. Note that each REP can begin its execution at different points in time. The REP in the directed graph with edge weights denoting processing times, that has the longest path length, is then the critical path [HPG02] of processing in the system. By definition, the REPs in a given JCDF are simply all possible paths in the JCDF, and these can be extracted using any standard graph algorithm such as Depth-First Search.

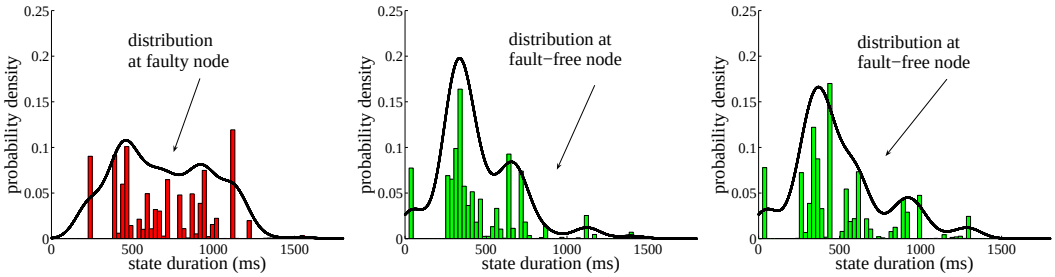

Figure 4.3: Visual illustration of the intuition behind comparing probability distributions of durations of the `WriteBlock` state across DataNodes on the slave nodes.

4.2 Diagnosis

Next, we provide an overview of our approach to automatically diagnosing the faulty node responsible for causing a performance fault, in which the execution of the MapReduce system is slower and completes in more time than it would otherwise without the presence of the fault.

Our main data-source for diagnosing faults consists of the durations of the node-local views of the states on each node extracted using SALSA as described in §4.1.1. Given the durations of all instances of a particular state, the algorithm then returns the node(s) that are responsible for a performance fault in the system; if this set of nodes returned is empty, this indicates that no performance fault is diagnosed in the system.

4.2.1 Intuition and Diagnostic Hypothesis

For all occurrences of a particular state (e.g. all Maps) in the execution of the system, we can compute a histogram of the durations of that state that executed on each node, for each node in the system. We observed that the histograms of durations of a given state tend to be similar across different nodes in the system in the absence of faulty conditions. When a fault is introduced or is present on a small subset of nodes, then the histogram of durations on these faulty nodes tend to differ from those of other nodes with no faults present, as we show in Figure 4.3.

Thus, we hypothesize that failures can be diagnosed by comparing the probability distributions of the durations (as estimated from their histograms) for a given state across hosts, assuming that a failure affects fewer than $\frac{n}{2}$ hosts in a cluster of n slave hosts.

4.2.2 Synopsis of Algorithm

The diagnosis algorithm can be implemented as an incremental one–this is done by considering durations of states within particular windows of time, and comparing the durations of only states that occur in the same window. The algorithm can thus also be implemented as an offline one by setting the window size to be as large as the duration of the entire experiment.

The algorithm consists of two stages. The first stage involves constructing a histogram of durations of a given state on each node; one challenge of this stage of the algorithm is to deal with potentially sparse data when there are few occurrences of the state in the window. The second stage involves comparing histograms across nodes. This involves computing a distance measure for each node's histogram, to the histograms of every other node in the system. The distance measure is then thresholded, and a node is indicted as faulty if its distance measure to more than $\frac{n-1}{2}$ of the other nodes in the system is greater than the threshold. [2]

4.3 Visualization of MapReduce Behavior

Next, we describe the high-level ideas behind our visualization of MapReduce program behavior based on our abstract state-machine views described in §4.1.

4.3.1 Aspects of Behavior

Our visualizations expose subsets of aspects of MapReduce behavior that affect program performance (in terms of job completion times). These aspects of behavior are exactly those described in the intuition of the Job-Centric Data-Flow (JCDF) abstraction (§4.1.2), and they are the times taken for each processing item (time), the sizes of data processed, i.e. either input sizes to control-items or data-sizes of data-items (volume), and the location, i.e. the particular node in a large cluster, where the processing occurred (space). Hence, our visualizations expose MapReduce behavior along combinations of the dimensions of time, space, and volume of the processing items, as well as the causality of the

[2] Alternatively, a global histogram of all state occurrences on every node can be computed, and nodes with distance measures to the global histogram exceeding a particular threshold can be indicted; this yields a comparison cost of $O(n)$ for an n-node cluster, although the global histogram construction would cost $O(n)$ as well.

processing items. In each visualization, we picked some dimensions along which to aggregate program behavior, and some dimensions along which to expose program behavior, to present different views which can give rise to different insights about program behavior.

4.3.2 Aggregations

Next, we describe the dimensions along which we chose to show MapReduce behavior, and dimensions along which we chose to aggregate behavior, and we describe the insights about behavior that the choices of dimensions to aggregate along and dimensions to show can yield. We describe the intuitions behind each of our visualizations, and describe them concretely in §5.3

These choices of dimensions for aggregating and showing behavior resulted in the three visualizations described in §5.3, namely the "Swimlanes", "MIROS" (Map Inputs, Reduce Outputs, and Shuffles), and *REP* plots.

First, in the "Swimlanes" visualization, we choose to show behavior along the dimensions of time and space, i.e. when in (wall-clock) time states were being executed, and on which nodes they were being executed. We choose to omit showing behavior along the volume dimension in our first visualization to keep the visualizations simple. Also, showing behavior only along the dimensions of time and space allows users to think about program execution in traditional imperative programming models, which focus on how long programs take to execute each segment of code. Hence, this view of MapReduce program behavior most closely resembles that of traditional profilers, albeit at a coarser granularity of Maps and Reduces rather than individual function calls, to allow users to think in terms of MapReduce abstractions.

Second, in the "MIROS" visualization, we choose to show behavior aggregated across time for the duration of an entire job, and we show the volumes of data when plotted against the host on which states (Maps, Reduces, Shuffles) ran, aggregating on a per-host basis. Aggregating in time provides a useful summary to users as it allows them to consider the overall equality of data volumes processed across hosts. Aggregating in space on a per-host basis is useful because the network bandwidth between hosts is significantly smaller than the bandwidth within a host (i.e. the system and memory buses) so that data volumes moved across hosts would cost significantly more in terms of latency than data volumes moved within hosts.

Finally, in the *REP* visualization, we show behaviors on a per-flow basis for each causal flow, as corresponding to each end-to-end *REP* path. We aggregate across flows by clustering similar flows before showing their behavior to enable scalable visualization. For

each cluster of similar flows, we show volumes of each stage of execution, and the times taken for each stage of execution. We omit the spatial information in this visualization as the main focus is on the data volumes processed and times taken along each causal flow, and each flow can span multiple hosts.

Chapter 5

Methodology

5.1 Abstracting MapReduce Behavior

Next, we describe the specific instantiations of the SALSA state-machine view of node-local execution, the Job-Centric Data-Flow (JCDF) directed graph view of global execution, and the Realized Execution Path (REP) causal flows, for data-intensive applications, when applied to MapReduce systems, and specifically to Hadoop and its Hadoop Distributed Filesystem (HDFS).

5.1.1 SALSA: Hadoop's Node-Local State-Machines

We describe Hadoop's state-machines of execution when SALSA is applied to abstract Hadoop's execution. These state-machine views are of local executions on each node, and we derive one state-machine for the execution of the TaskTracker daemons, and one for the execution of the DataNode daemons. The state-machine view of the TaskTracker daemon encodes both the control-flow of Map and Reduce executions, and the data-flow from Maps to Reduces, while the state-machine view of the DataNode daemon encodes the data-flow between DataNodes and HDFS clients, which can either be other DataNodes (e.g. replication), Maps or Reduces running on TaskTrackers, or command-line clients (which we exclude from consideration in this work). We describe these state-machines in detail next.

Control-Flows

The main states of execution of control items in a Hadoop MapReduce program are Maps and Reduces. In addition, control passes from Maps to Reduces via the Shuffle (also known as ReduceCopy, named for Reduces copying the outputs of Maps) stage, which is transparent to user programs and is executed by the framework to copy the outputs of Maps to the relevant Reduces where they are needed. However, control can pass from a single Map to multiple Reduces when a Map's output is needed by multiple Reduces, and can pass from multiple Maps to a single Reduce, when the Reduce depends on the outputs of multiple Maps for its input.

In addition, the Reduce state of execution can be further broken down into two finer-grained states: the first, which we call ShuffleWait (or ReduceCopyReceive), is the time the Reduce spends waiting for the outputs of all the Maps it depends on to be copied. Each Reduce needs to wait for all Maps to be completed before it can begin execution, because any Map could potentially generate an output key that the particular Reduce is responsible for processing. Hence, the ShuffleWait state represents a synchronization point in the control-flow during which it waits for all Maps to complete. The ShuffleWait state within the Reduce state is managed by the framework and transparent to user code, while the second state is the user-written Reduce code, which we call the Reducer (to be distinguished from the broader, coarser-grained Reduce state).

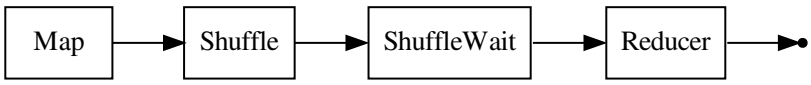

Figure 5.1: States in the state-machine view of Hadoop's control-flow

Hence, the control-flow in Hadoop's MapReduce programs extracted by SALSA consists of, in order of their occurrence in time, the Map, Shuffle, ShuffleWait, and Reducer states, along each independent thread of execution. In addition, the causality along each thread of execution (i.e. which Maps pass control to which Reduces) is preserved by means of unique identifiers of Maps and Reduces emitted in log messages indicating the beginning and end of each state, together with the data-flows (described next) described in the Shuffle log messages, which indicate the source Maps and destination Reduces by their unique identifiers. These states form the state-machine view of Hadoop's control-flow. This is as illustrated in Figure 5.1.

Data-Flows

There are two types of data-flows in Hadoop MapReduce programs: the first being the movement of data blocks written to and read from HDFS to and from DataNodes, and the second being the movement of Map outputs to Reduces to be input to Reduces.

In the HDFS data-flows, the two main states of execution of the DataNodes are block reads (ReadBlock) and block writes (WriteBlock) between DataNodes, and HDFS clients, which can be Maps and Reduces, or other Datanodes. We further distinguish between local and remote block reads (i.e. whether the HDFS client is on the same host as the DataNode), and replicated block writes (due to HDFS triplication of stored blocks). These states exactly encode the data-flows at the HDFS layer. These data-flows are captured by means of the source and destination hosts taking part in the transaction, as recorded in Hadoop's log messages for each block read and write.

The data-flows between Maps and Reduces in TaskTrackers occurs at the Shuffle stage. These data-flows are captured in the source Map ID and destination Reduce ID recorded along with each log message for each Shuffle.

5.1.2 Job-Centric Data-Flows

Next, we describe concretely the Job-Centric Data-Flow for Hadoop MapReduce programs, and briefly describe how it is constructed from the SALSA- extracted control-flows and data-flows of MapReduce execution as described in §5.1.1.

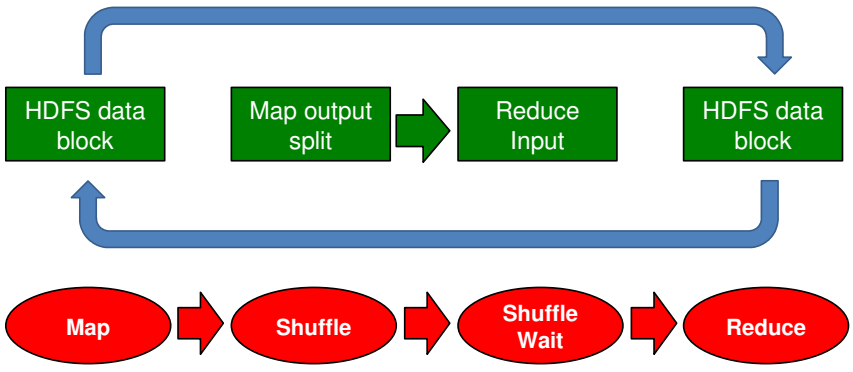

Figure 5.2: Control-items (ellipses in red) and data-items (boxes in green) of states in the state-machine view of Hadoop's execution

First, the control-items in Hadoop's MapReduce JCDF are the Map, Shuffle, Shuffle-Wait, and Reducer states, and the data-items in the JCDF are the ReadBlock and Write-Block states, and the Map-Output items which are the inputs to the Shuffle state, and the Reduce-Input items, which are the outputs from the Shuffle state. These are as shown in Figure 5.2.

Second, the JCDF is formed by effectively "stitching" together control- and data-items, by identifying the input and output data-items of each control-item. Concretely: (i) the ReadBlock data-item states are the inputs to the Map control-item; (ii) the Map-Output data-items are the outputs of the Map control-item; (iii) the Map-Output data-items are also the inputs to the Shuffle control-item; (iv) the Shuffle control-item passes control directly to the ShuffleWait state to block while waiting for all Shuffles to be received; (v) the Reduce-Input data-items are the outputs from the ShuffleWait control-item; (vi) the Reduce-Input data-items are the inputs to the Reducer control-item; (vii) the WriteBlock data-items are the outputs from the Reducer control-item. This is as illustrated in Figure 5.3.

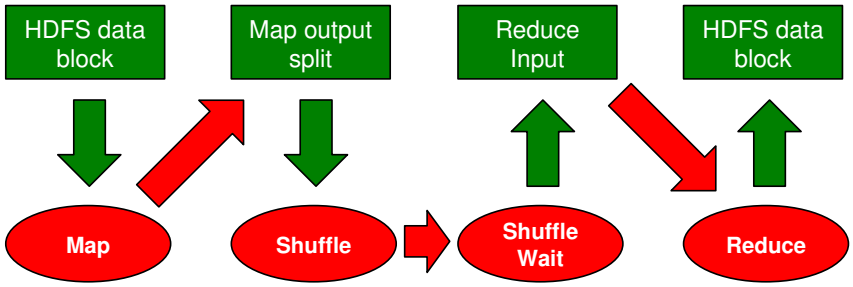

Figure 5.3: Job-Centric Data-Flow formed by "stitching" together control-items (ellipses in red) and data-items (boxes in green) of Hadoop's execution states. The directed edges show the direction of the full causality of execution in a MapReduce program.

Vertices are created for each control- and data-item, and edges are added between control- and data-items as corresponding to which specific data-items are inputs to and outputs from which control-items, as described. The determination of which data-items are inputs to/outputs from each control- item is as follows. These are clearly marked as the source Map/destination Reduce for Shuffles and their Map-Output and Reduce-Input data-items. In the case of Maps and their input ReadBlock data-item states, at present, we correlate in time the destination host of a ReadBlock state with the Maps that are executing on that same host; hence, if a ReadBlock occurs with block b_i being read to host A, and Map m_j is executing on host A at the same time, then we say that the ReadBlock state

of b_i is an input to the control-item Map m_j. Note that multiple Maps can be executing on a given host at the same time; in this case, we attribute the ReadBlock of block b_i to every Map executing on host A at the same time; this captures a superset of the true causal paths[1]. Similarly, we correlate in time the source host of a WriteBlock state with the Reduces that are executing on the same host. This does not result in any spurious causal paths being generated. However, the current Hadoop DataNode logging has been augmented in version 0.21 to include the client ID of the HDFS client in WriteBlock and ReadBlock states, which includes the Map ID or Reduce ID of the writer or reader of the block, which would completely eliminate this issue of spurious causal edges generated.

Hence, this forms the directed JCDF graph of full control- and data-flows in a MapReduce program in Hadoop, with the vertices representing the ShuffleWait state potentially having out-degree > 1 when the output of a Map is read by multiple Reduces, and the vertices representing the Reducer state potentially having out-degree > 1 when a Reducer writes to multiple data-blocks in HDFS. The vertices representing Map states typically have in-degree 1 because Hadoop creates one Map for each input data-block (if no spurious edges are generated); they also have out-degree 1 because each Map logically creates one output item. Each Shuffle state corresponds to exactly one Map, as it serves the output of exactly that Map, so that vertices representing Shuffle states always have in-degree of 1. Also, each Reducer has exactly one ShuffleWait state, so that the ShuffleWait vertex always has an out-degree of 1.

5.1.3 Realized Execution Paths

Finally, the Realized Execution Paths (*REP*) are simply paths that begin from vertices with in-degree 0 in the JCDF, and that end on vertices with out-degree 0. Hence, in the case of Hadoop MapReduce programs, *REP*s begin on ReadBlock states, and end on WriteBlock states, traversing the following states in each path: ReadBlock, Map, Map-Output, Shuffle, Reduce-Input, ShuffleWait, Reducer, WriteBlock. Then, the edges are each weighted with either the duration taken for that transition, or with the volume of data processed or moved for that transition. These edge values are summarized in Table 5.1.

Each *REP* represents a single causal flow through the system, and these can be recovered from the JCDF by performing a Depth-First Search on the JCDF. Paths comprising the volume-weighted edges represent the total processing load along that causal flow, while

[1]While we capture a superset of the true causal paths, and as a result also capture spurious paths, we never miss any causal path that is present; we believe that the value of capturing causal paths in a large distributed system such as MapReduce, in the applications it has (§10.2.1) far outweigh the spurious causal paths generated

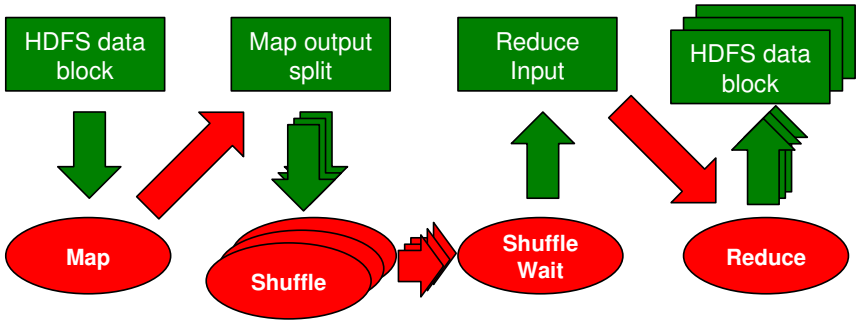

Figure 5.4: Illustration of a Realized Execution Path. The multiple copies of a particular directed edge and of a particular vertex show the vertices with states with in-degree or out-degree greater than 1, and each Realized Execution Path is formed by picking one of the multiple edges in each group of edges.

paths comprising the time-weighted edges show the time taken by that causal flow. However, the total weight along paths with time-weighted edges do not represent wall-clock time elapsed during the execution of that causal flow, because the edges from the Shuffle states to the ShuffleWait states represent a synchronization point, so that the time on that edge subsumes some of the processing times of Maps completed earlier. We discuss possible alternative abstractions for the *REP* in §10.2.2 that take the synchronization points into account.

5.2 Diagnosis

First, for a given state on each node, probability density functions (PDFs) of the distributions, $distrib_i$'s, of durations at each node i are estimated from their histograms using a kernel density estimation with a Gaussian kernel [Was04] to smooth the discrete boundaries in histograms.

In order to keep the distributions relevant to the most recent states observed, we imposed an exponential decay to the empirical distributions $distrib_i$'s. Each new sample s with duration d would then be added to the distribution with a weight of 1. We noticed that there were lull periods during which particular types of states would not be observed. A naive exponential decay of $e^{-\lambda\Delta t}$ would result in excessive decay of the distributions during the lull periods. States that are observed immediately after the lull periods would thus have large weights relative to the total weight of the distributions, and thus effectively result in the distributions collapsing about the newly observed states. To prevent this

Source Vertex	Destination Vertex	Duration Edge	Volume Edge
ReadBlock	Map	Time to read block	Size of block read
Map	Map-Output	Time to run Map	Size of total Map input processed
Map-Output	Shuffle	Time to transfer data to Reduce	Size of data shuffled
Shuffle	ShuffleWait	Time to transfer data to Reduce	Size of data shuffled
ShuffleWait	Reduce-Input	Time spent waiting for all Shuffles	Size of all Map outputs collected
Reduce-Input	Reducer	Time spent performing Reducer	Size of input processed by Reducer
Reducer	WriteBlock	Time to write block	Size of block written

Table 5.1: Edge weights in *REP*s for Hadoop MapReduce programs extracted from the JCDF for each causal flow.

unwanted scenario, we instead used an exponential decay of $e^{-\lambda \frac{lastUpdate_i - t}{\alpha(lastUpdate_i - t)+1}}$, where $lastUpdate$ is the time of the last observed state, and t is the time of the most recent observation. Thus, the rate of decay slows down during lull periods, and in the limit where $lastUpdate - t \rightarrow 0$, the rate of decay approaches the naive exponential decay rate.

The difference between these distributions from each pair of nodes is then computed as the pair-wise distance between their estimated PDFs. The distance used was the square root of the Jensen-Shannon divergence, a symmetric version of the Kullback-Leibler divergence [ES03], a commonly-used distance metric in information theory to compare PDFs.

Then, we constructed the matrix $distMatrix$, where $distMatrix(i, j)$ is the distance between the estimated distributions on nodes i and j. The entries in $distMatrix$ are compared to a $threshold_p$. Each $distMatrix(i, j) > threshold_p$ indicates a potential problem at nodes i, j, and a node is indicted if at least half of its entries $distMatrix(i, j)$ exceed $threshold_p$. The pseudocode is presented as Algorithms 1, 2 in Appendix A.1.

5.3 Visualization

We describe our three visualizations of MapReduce behavior based on the SALSA extracted state-machine view and the Job-Centric Data-Flows, without discussing actual experimental data or drawing any conclusions from the visualizations (although the visualizations are based on real experimental data). We describe the actual workloads and case studies in §7.2.

5.3.1 "Swimlanes": Task progress in time and space.

In the "Swimlanes" visualization, we show task progress as it unfolds in time, and we also show how tasks running on different nodes progress in time. Hence, the swimlanes plots show MapReduce behavior in time and space, and omit the volume dimension of behavior.

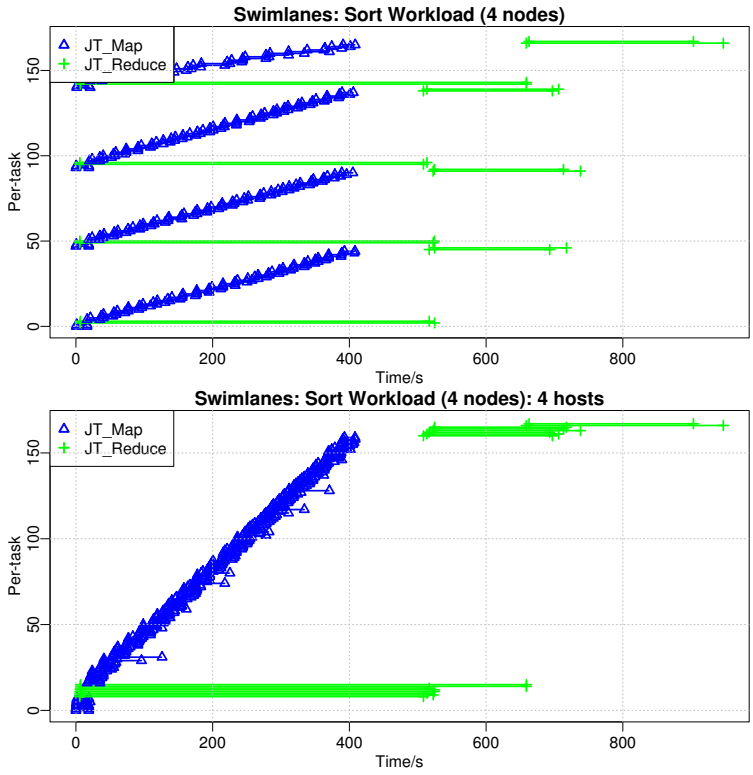

Figure 5.5: *Swimlanes* plot for the Sort workload. Top plot shows tasks grouped by host, bottom plot shows tasks grouped by start time.

Consider Figure 5.5 for an illustration of the basic properties of the Swimlanes visualization. In this plot, the x-axis denotes wall-clock time elapsed since the beginning of the job, and each horizontal line corresponds to the execution of a state (e.g., Map, Reduce) running in the marked time interval (hence the number of horizontal lines corresponds exactly to the total number of tasks in the job as represented in the plot). For instance, for a horizontal line corresponding to a Reduce task (green line, with a '+' markers) marked from time 0 to approximately 650, this illustrates a Reduce task running from $t = 0$ to $t = 650$ during the job. The top and bottom plots are of the same run of a MapReduce job, and illustrate two different ways of visualizing the same job. The top plot groups tasks according to the host/node on which they ran, and sorts tasks within each group by the time at which they started. The bottom plot merely sorts tasks by the time at which they started. Each plot can show one or more jobs, and one job can be distinguished from the next by identifying Map tasks that begin execution after any Reduce task has completed, such as in Figure 5.7, which plots the execution of two jobs, one from $t = 0$ to $t = 90$, and one from $t = 90$ to $t = 160$.

Figures 5.6, 7.3, 7.4 and 5.7 are all variants on the basic swimlanes plot, and provide different levels of detail and different levels of aggregation. Figure 5.6 shows a detailed view with all states across all nodes–this is useful for showing the detailed behavior of Hadoop, but the large number of states shown can result in an overwhelming amount of information. Figures 7.3, 7.4 and 5.7 show only Maps and Reduces, as MapReduce programs spend most of their processing time in these states. This allows users to focus their attention on these key components of the behavior of MapReduce programs. Figure 7.4 groups states (Maps and Reduces) by nodes, aggregating these states together on a per-node basis so that nodes with persistently long tasks, or that are not running tasks, can be identified. Figure 5.7 shows the *Swimlanes* plot for a job executing on a 49-slave-node cluster, and demonstrates the scalability of the *Swimlanes* visualization and that it can effectively present information even about large clusters. In general, *Swimlanes* are useful for capturing dynamic Hadoop execution, showing where the job and nodes spend their time.

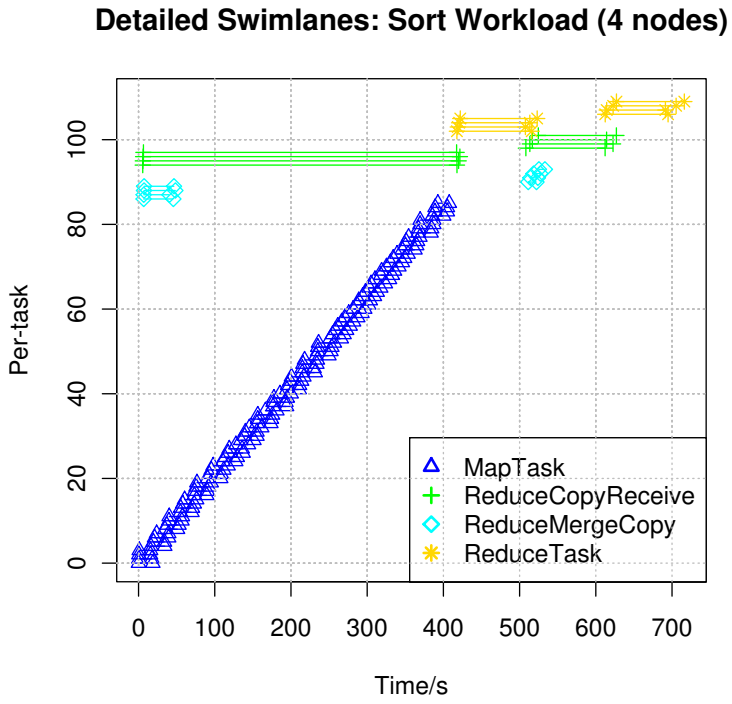

Figure 5.6: *Swimlanes*: detailed states: Sort workload

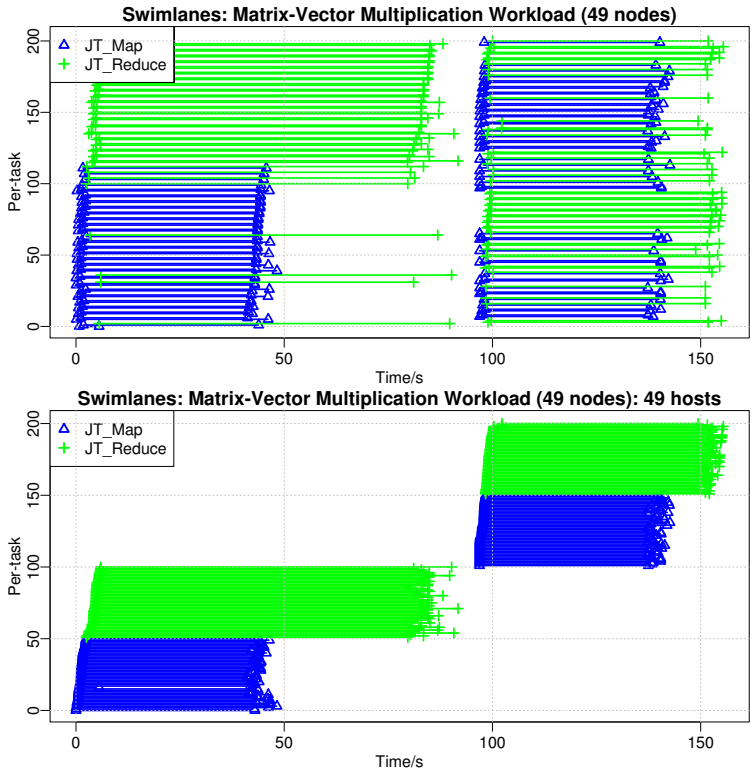

Figure 5.7: *Swimlanes* plot for 49-node job for the Matrix-Vector Multiplication; top plot: tasks sorted by node; bottom plot: tasks sorted by time.

5.3.2 "MIROS" plots: Data-flows in space.

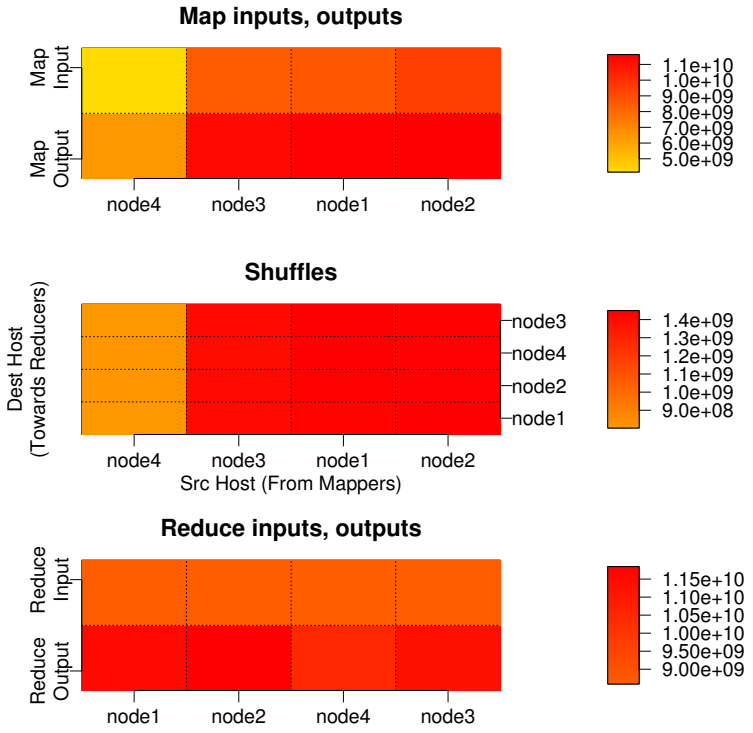

Figure 5.8: *MIROS*: Sort workload; (volumes in bytes)

MIROS (Map Inputs, Reduce Outputs, Shuffles, Figure 5.8) visualizations show input data volumes into all Maps and output data volumes out of all Reduces on each node, and between Maps and Reduces on nodes. Each visualization consists of three heatmap plots, with each heatmap being a matrix of values, and the value represented by the intensity of the colour in the cell. These three heatmap plots illustrate the following values: the first shows input and output data volumes of Maps, the second shows input and output data volumes out of Shuffles that move Map outputs from each pair of source/destination host pair, and the third shows input and output data volumes out of Reduces. These volumes are aggregated over the a single MapReduce program, and over nodes. MIROS is useful in highlighting skewed data flows that can result in bottlenecks when a particular node needs to process more data than other nodes.

5.3.3 REP: Volume-duration correlations.

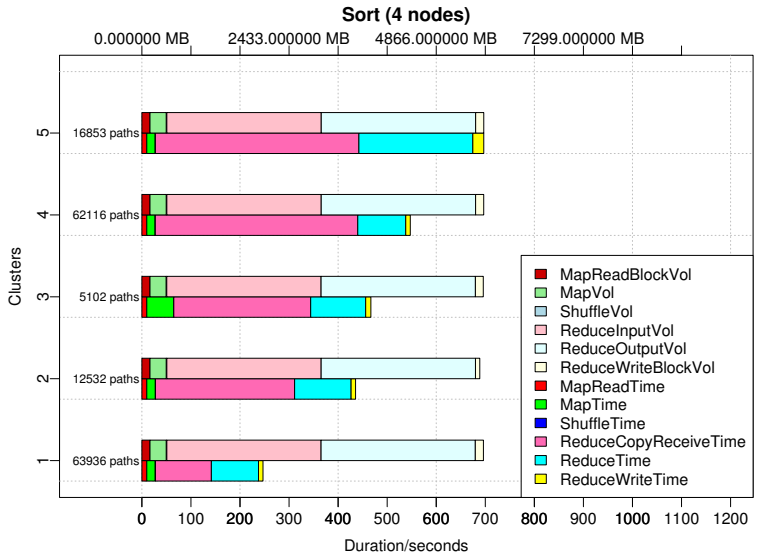

Figure 5.9: *REP* plot for Sort workload

For each flow of causality in a given Realized Execution Path (REP), we show the time taken for a causal flow, and the volume of inputs and outputs, along that flow (Figure 5.9). Each REP is broken down into time spent and volume processed in each state. We group similar paths for scalable visualization using the K-Means clustering algorithm. For each group, the top bar shows volumes processed in each processing (control- or data-item), and the bottom bar shows durations taken for that processing item. This visualization is useful in (i) checking that states that process larger volumes should take longer, and (ii) in tracing problems back to any previous stage or data that might have affected it.

Chapter 6

Implementation

Next, we describe key architectural features and design choices of our implementation of the log-based extraction of MapReduce behavior, our diagnosis algorithm, and our visualizations of MapReduce behavior.

Our work is implemented as a tool-chain which takes Hadoop's natively generated logs (from the TaskTracker and DataNode daemons) as its input, and generates a diagnosis outcome (whether a fault is present, and if so, which node(s) are responsible), and visualizations of behavior of the MapReduce program. The tool-chain first processes Hadoop's logs to produce intermediate abstract representations (as described in §5.1) of MapReduce behavior, and we describe the implementation of these intermediate representations as well. Our diagnosis algorithm and visualizations then take these intermediate representations as inputs to produce their final output. The overall structure of the components of our tool-chain loosely mirrors our approach, as illustrated in Figure 4.1.

6.1 Extracting Abstract Views of MapReduce Behavior

First, we describe how we build abstract views of MapReduce behavior from Hadoop's natively-generated TaskTracker and DataNode logs, and we describe the intermediate representations generated of these abstract views.

6.1.1 Intermediate Representation of MapReduce Behavior

Node-Local State-Machine View

For each node, we represent a state of execution as a tuple of fields which describe the state. Every state is described by its state name (e.g. Map, Reducer, WriteBlock, ReadBlock), a unique identifier for the state (e.g. Map identifier, Reduce identifier), the start and end times of the state, and the host the state was executed on. In addition, tuples of certain state types are augmented with additional information, such as the source/ destination host other than the state itself (e.g. destination host in ReadBlock).

The execution of the TaskTracker is fully described by the Map, Shuffle, ShuffleWait, and Reducer states. The execution of the DataNode is fully described by the ReadBlock, and WriteBlock states (which additional can be local or remote variants depending on the source/destination addresses).

The unique identifier of the Map and Reducer states are naturally their Hadoop internal Map and Reduce identifiers since these are unique, and the unique identifier of the Shuffle state is the concatenation of the identifier of the source Map and the identifier of the destination Reduce, since this pair is unique. The unique identifier of the ShuffleWait state is the same as that of the Reduce state, since each Reducer has exactly one ShuffleWait state[1].

The unique identifier of the ReadBlock state is the concatenation of the internal Hadoop unique block identifier and the hostname of the receiving host, while the unique identifier of the WriteBlock state is the concatenation of the internal Hadoop unique block identifier and the hostname of the writing host.

Hence, the state-machine view of each node can be compactly represented in a simple relational database or as line records in a delimited text file.

Job-Centric Data-Flow

The Job-Centric Data-Flow is a directed graph; in our implementation, the graph is stored as an edge list and a list of vertices with additional attributes, as the graph can be potentially too large for storing an adjacency matrix to be infeasible, and has significantly fewer edges than vertices. The list of vertices is simply the list of states from the node-local state-machine view, and the key additional information in the JCDF representation is the

[1]Effectively, then, a unique identifier for each state can be obtained by concatenating the state name with its unique identifier.

edge-list which correlates execution explicitly across the control-items and data-items in the node-local states.

Realized Execution Paths

The Realized Execution Paths are simply stored as a list of paths through the Job-Centric Data-Flow directed graph, and each path is stored as a list of vertices along the path, with the time-weights and volume-weights for easy computation of clusters of *REP*s.

6.1.2 Extracting Intermediate Representations

Next, we describe the implementation of the state-extraction from Hadoop's logs to build the intermediate representations of MapReduce behavior.

The first stage involves parsing the TaskTracker and DataNode logs of each host to extract the states executed on each node in both the TaskTracker and DataNode on each host. This involves identifying log tokens associated with the beginning and end of each state, and extracting parts of log messages that map to the fields in the tuple identifying each state.

Each state requires two log messages to fully define it, one to indicate the beginning and one to indicate the end of that state's execution. Hence, the log parsing involves maintaining a list of previously-seen log messages identifying the beginning of a state, and matching log messages identifying the end of those states to create the tuple for that state. The unique identifiers for each state are used to ensure that the beginning and end of the execution of each instance of a particular type of state can be unambiguously identified.

The second stage involves constructing the Job-Centric Data-Flow directed graph from the individual state tuples extracted in the log parsing stage. The states created in the log parsing stage correspond exactly to the vertices in the JCDF graph, and this graph construction stage involves adding edges between the vertices by matching identifiers. Particular identifiers are matched between every type of state-pairs, such as between the ReadBlock and Map states, as described in detail in §5.1.2. This is essentially a search problem, in which the source vertex of an edge is first selected, and then we search for the destination vertex based on the identifier information encoded in both the source and destination vertices; for instance, in connecting Maps to Shuffles to ShuffleWaits, the Map identifier is matched with the Map-source part of the identifier in the Shuffle identifier, and the Reduce-destination identifier part in the Shuffle identifier is matched with the ShuffleWait identifier, and the internal representation's vertices are then listed as pairs to

create edges in the edge list.

The third stage involves extracting all *REP*s from the JCDF directed graph and simply involves a simple depth-first search traversal beginning from every vertex with zero in-degree, and the volume-weights and time-weights are recorded along with the list of paths generated to enable easy clustering based on the times and volumes along stages in the *REP*.

Stand-alone Operation

In our first prototype of the log-parsing, JCDF construction, and *REP* extraction, we implemented all three components in C++. The log-parsing was implemented in 14 KLOC of C++, while the JCDF construction and *REP* extraction were implemented in 3.4 KLOC of C++.

The log-parsing module was separated into two main components. The first provided an interface to raw log files by maintaining a file descriptor to maintain a file pointer in the log; this allows for incremental processing of logs for online, on-demand extraction of states for online operation (described later); this interface manages basic splitting of log messages into timestamps, logging class, severity level, and the actual log message, and allows time-based sampling of log messages–given a time interval, this module returns the relevant log messages within the time interval. The second component is log-type specific, and provides the main SALSA parsing logic for converting tokens to states, remembering previously-seen tokens corresponding to the beginning of states, and for mapping tokens in log messages to state tuple values. Finally, the `TimeSeries` class provides the external interface to the log- parser, and allows the returning of various types of aggregated, summarized, or detailed views of the extracted state-machine.

We chose C++ for our implementation due to the large memory requirements of generating large JCDF graphs, and to obtain the fastest possible speed due to the high computational complexity of extracting all *REP*s from the potentially large JCDF.

In addition, we chose to implement our log-parsing in C++ so that our log-parsing functionality would also be available as a C++ library for inclusion in the ASDF [BKK+08] Automated System for Diagnosing Failures for real-time extraction and parsing of log-data. C++ was the language of choice for the low latency design requirement of ASDF.

However, this implementation of our MapReduce behavior abstraction executes only on a single host and does not scale well with large numbers of nodes.

MapReduce Implementation

We also implemented our log-parsing and state-machine construction as Java Hadoop MapReduce programs for execution as part of the log analysis pipeline in the Chukwa [BKQ+08] log collection and aggregation framework. A second motivation of implementing our log analysis tools and state-machine construction as MapReduce programs was to improve the scalability of our tool-chain.

We describe our choices of key and value types for the Mappers and Reducers of our MapReduce implementations of each stage of the abstract MapReduce view construction, as this constitutes the key design point of MapReduce programs. In MapReduce programs, Maps specify an input key type and an input record type, and generate output keys of an intermediate output key type, and output records of an intermediate output value type. Reduces then read input keys of the intermediate key/value type, and generate output of a possibly different output key type and output value type. Each intermediate value with the same intermediate key will be sent to the same Reduce, and the Reduce receives, for each intermediate key, a list of values associated with that key.

Our MapReduce tool-chain consists of: (i) a MapReduce program for constructing state-machine views for each node for all nodes, (ii) a series of MapReduce programs for constructing the JCDF, and *REP*s jointly. Our tool-chain takes as its input the output from the Demux phase of Chukwa's log post-processing, in which each log message is stored as a `ChukwaRecord`, which consists of a series of key/value pairs, with string values, and a series of meta-data key/value pairs. Chukwa performs rudimentary log-parsing by splitting the log message into timestamp, logging component, severity level, and log message.

SALSA

As described, the first part of our tool-chain consists of a single MapReduce program which reads raw logs as generated by Chukwa in the form of `SequenceFiles`[2] of `ChukwaRecords`, with each record containing a Chukwa-processed log message, and produces a `SequenceFile` of `ChukwaRecords`, with each `ChukwaRecord` containing a single tuple defining a single state.

Instead of parsing TaskTracker and DataNode logs, our MapReduce state- machine construction uses the more compact data sources of Hadoop's JobTracker job history logs, which record Maps and Reduces and their completion times and input and output data volumes, and ClientTrace log messages in the DataNode and TaskTracker logs which specif-

[2] `org.apache.hadoop.io.SequenceFile`, a Hadoop MapReduce library class

ically track inter-node communication, as introduced in Hadoop 0.19.

This SALSA state-machine builder [3] contains a collection of Mapper classes, each of which processes records from a particular log type (e.g. job history logs, DataNode ClientTrace logs) to produce a common intermediate state-machine state entry, which can describe either the start or end of a state, and which stores the fields for the state tuple. To ensure that the same Reducer processes both the start and end entries of a given state instance, both the state entries are given the same intermediate key, and the key is also chosen such that it is unique to the start and end entries of the particular state instance. The Reducer then generates a common state-machine view by writing out standardized tuples of states regardless of whether the states are of data-items of the DataNode's processing or of control-items of the TaskTracker's processing.

Job-Centric Data-Flows and Realized Execution Paths

We relegate discussion of a possible implementation strategy for the extraction of Job-Centric Data-Flows and Realized Execution Paths to §8.1 as the JCDF extraction has not been fully implemented as a MapReduce program at the time of writing.

6.2 White-Box Diagnosis

A prototype of our white-box diagnosis algorithm has been implemented in MATLAB, and currently performs offline white-box diagnosis by taking as its input the tuples of states generated by our log-parsing phase, and generating a diagnosis outcome (a vector of binary values, indicating whether a fault is present) for each node of whether a fault is present at that node. If no nodes are indicated as having a fault present, that can be taken to indicate there is no fault present in the system.

Although our current implementation of the diagnosis algorithm is offline, it can produce incremental output. It accepts as a configuration parameter a window size, in which case it performs diagnosis incrementally for each window of time, and returns a set of diagnostic outcomes for each window.

In addition, we have also implemented visualizations of the diagnosis outcomes of the incremental diagnosis as a heatmap, with 'cool' colors such as blue indicating high peer similarity and 'hot' colors such as red indicating high peer dissimilarity, indicating the presence of a fault. An example of such a visualization is in Figure 6.1, in which a CPU

[3] `org.apache.hadoop.chukwa.analysis.fsm.FSMBuilder`

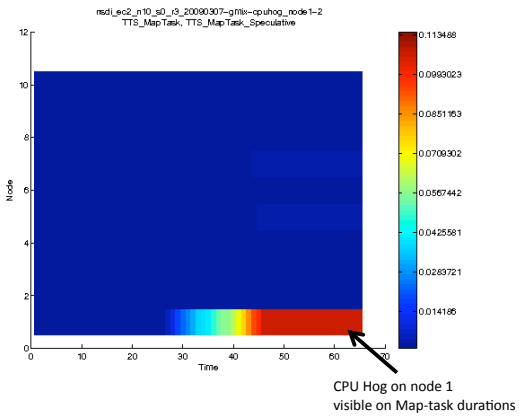

Figure 6.1: Visualization of time-series of diagnosis outcomes from the white-box diagnosis algorithm; the x-axis shows time in seconds, while the y-axis shows the diagnosis outcome for each node for each point in time; cool colors indicate high peer similarity with all other nodes; warm colors indicate low peer similarity with all other nodes and indicates the presence of a fault on the given node.

Hog (see §7.1.2 for description of fault injection) was injected in Node 1, resulting in high peer dissimilarity of Node 1 relative to the other nodes in the cluster. We defer a full evaluation of the diagnosis algorithm to §7.1.

An earlier version of the white-box diagnosis algorithm has also been previously implemented in C++ as an analysis module for the ASDF [BKK+08] Automated System for Diagnosing Failures which took as its inputs the state-machine states extracted online using the log-parsing library.

6.3 Visualization

We implemented both offline and online visualizations of our Hadoop and MapReduce behavior visualizations, with the offline version being both a prototype, as well as to support batched operation to process large amounts of post-mortem log data, and with the online version being targeted as system administrators and Hadoop users for online diagnosis during the execution of a job itself.

6.3.1 Offline Operation

The offline visualizations were implemented as scripts for the GNU R [R D08] statistical and graphing package, and these scripts took as their inputs comma-separated value (CSV) files containing the state tuples of the state-machine view, and generated graphs as image files.

The *Swimlanes* charts were plotted on Cartesian coordinates by specifying task lines as X-coordinate extents and assigning tasks unique Y-coordinates. The *MIROS* charts were plotted using the standard `heatmap` command in GNU R, while the *REP* volume-duration correlation charts were plotted as standard bar charts, while normalizing the scales of total causal flow durations to total causal flow volumes to enable comparison of the two (i.e. the Y-axis scaled the maximum-duration flow to be represented using the same length as the maximum-volume flow).

In addition, the plotting of the *Swimlanes* chart has also been implemented using a simple Python parser for job history logs generated by the JobTracker (one log file per job containing all counters and task start and end times), which generates a `gnuplot`-compatible data file, and the chart is plotted using our custom `gnuplot` command file.

6.3.2 Online Console – Chukwa HICC

Finally, we have also implemented our visualizations as widgets for the Chukwa Hadoop Infrastructure Care Center (HICC) online monitoring web application, as shown in Figure 6.2. Chukwa provides a Metrics Data Loader (MDL) component which loads Chukwa-generated data from HDFS into a relational database (Chukwa HICC uses MySQL at time of writing) so that web-based rendering of collected metrics does not need to incur the performance overhead of reading through large, sequential-access files from HDFS. The MDL also provides additional features which improve the scalability of data storage for online rendering and visualization, while preserving the low latency of the rendering. These include the automated downsampling of historical data and maintaining time-based pointers to data (i.e. tables are named according to the time period they store data for to improve speed of retrieving data given the time period of the data).

First, we wrote data dictionary entries for loading the state-machine tuples from the `SequenceFiles` of `ChukwaRecords` into the relational database automatically using the MDL. Then, we explored two methods of implementing the visualization widgets.

In the first, JavaScript-based visualizations, we implemented our visualizations using JavaScript libraries. Finally, the data plotted by the JavaScript libraries was supplied by

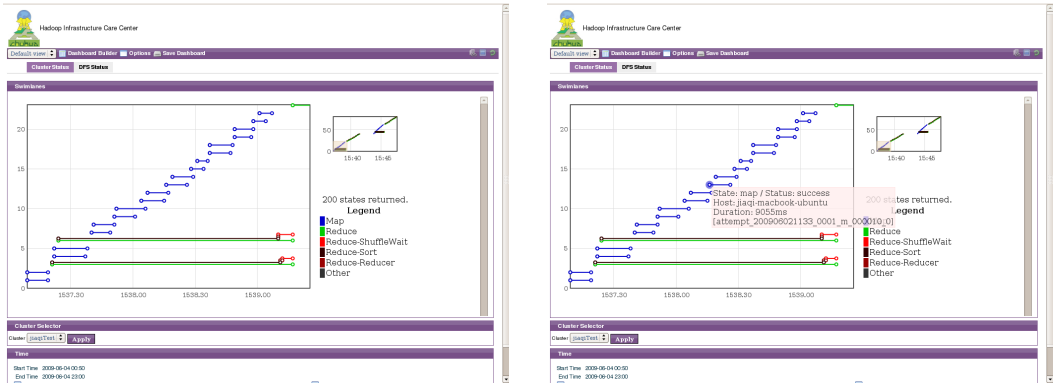

Figure 6.2: Screenshot of *Swimlanes* interactive visualization widget for Chukwa HICC. Screenshot on right shows mouseover details for state.

accessing the relational database using Java Server Pages (JSP) and transforming the data into JavaScript data structures that were fed to the JavaScript visualization libraries.

At the time of writing, the *Swimlanes* visualization has been fully implemented for the Chukwa HICC, using the Flot [Lau09] JavaScript plotting library, and using JSP to retrieve the state tuples from MySQL and formatting the plotting coordinates as dynamically-generated JavaScript data structures that are then passed to Flot.

The key advantage of the JavaScript visualization is that it allows a high degree of interactivity, as users can select regions of the *Swimlanes* plot to zoom in on and out of, and obtain task identifiers and details via mouse-overs. However, the key disadvantage is that this technique does not scale well to large jobs with complex *Swimlanes*, e.g. tasks with more than 1000 states, as the web browser becomes unresponsive and the user experience is greatly deteriorated. This prompted us to explore the option of rendering the *Swimlanes* offline and returning the user a statically-generated image.

In the second visualization, we used the Prefuse [HCL05] Java visualization toolkit to generate the *Swimlanes* chart offline, and we returned an image to the front-end to be

displayed to the user. This greatly improved user-responsiveness of the web browser, although the user still suffers from some latency in waiting for the image to be generated offline. In this implementation, both the data retrieval and image generation were completely implemented in Java, while the web interface merely acts as a skeleton to call the Java class for generating the image.

Chapter 7

Evaluation

7.1 Diagnosis of Synthetic Faults

First, we evaluate the ability of our white-box diagnosis algorithm to diagnose performance faults using state-machine statistics, specifically, the durations of different types of states in each job. Our evaluation consisted of injecting a performance fault and evaluating if our algorithm was able to correctly indict the node we injected the fault on.

7.1.1 Testbed and Workload

We analyzed Hadoop's DataNode and TaskTracker logs from Hadoop 0.18.3 running on 10- and 50-node (where cluster size indicates number of Slave nodes, with an additional Master node not in the count) clusters on Large instances on Amazon's EC2. Each node had the equivalent of 7.5 GB of RAM and two dual-core CPUs, running amd64 Debian/GNU Linux 4.0. Each experiment consisted of one run of the `GridMix` benchmark. `GridMix` is a multi-workload benchmark that models the mixture of job types (namely generating data, sorting data, scanning data for retrieval, and generating inverted indices) used at industrial Hadoop installations such as Yahoo!, and simulates the use of a shared Hadoop cluster by multiple users by staggering MapReduce job submissions in a manner that mimics observed data-access patterns in actual user jobs in enterprise deployments. The `GridMix` benchmark has been used in the real-world to validate performance across different clusters and Hadoop versions. We scaled down the size of the dataset to 2MB of compressed data for our 10-node clusters and 200MB for our 50-nod clusters to ensure timely completion of experiments.

[Source] Reported Failure	[Fault Name] Fault Injected
[Hadoop users' mailing list, Sep 13 2007] CPU bottleneck resulted from running master and slave daemons on same machine	[CPUHog] Emulate a CPU-intensive task that consumes 70% CPU utilization
[Hadoop users' mailing list, Sep 26 2007] Excessive messages logged to file during startup	[DiskHog] Sequential disk workload wrote 20GB of data to filesystem
[HADOOP-2956] Degraded network connectivity between DataNodes results in long block transfer times	[PacketLoss5/50] $5\%, 50\%$ packet losses by dropping all incoming/outcoming packets with probabilities of $0.01, 0.05, 0.5$
[HADOOP-1036] Hang at TaskTracker due to an unhandled exception from a task terminating unexpectedly. The offending TaskTracker sends heartbeats although the task has terminated.	[HANG-1036] Revert to older version and trigger bug by throwing NullPointerException
[HADOOP-1152] Reduces at TaskTrackers hang due to a race condition when a file is deleted between a rename and an attempt to call getLength() on it.	[HANG-1152] Simulated the race by flagging a renamed file as being flushed to disk and throwing exceptions in the filesystem code
[HADOOP-2080] Reduces at TaskTrackers hang due to a miscalculated checksum.	[HANG-2080] Simulated by miscomputing checksum to trigger a hang at reducer

Table 7.1: Injected faults, and the reported failures that they simulate. HADOOP-xxxx represents a Hadoop bug database entry.

7.1.2 Injected Faults

We injected one fault on one Slave node in each cluster to validate the ability of our algorithms at diagnosing each fault. The faults cover various classes of representative real-world Hadoop problems as reported by Hadoop users and developers in: (i) the Hadoop issue tracker [Apa06] from October 1, 2006 to December 1, 2007, and (ii) 40 postings from the Hadoop users' mailing list from September to November 2007. We describe our results for the injection of the seven specific faults listed in Table 7.1.

7.1.3 Results

We evaluate the ability of our white-box diagnosis algorithm to effectively diagnose the injected fault by using the true-positive and false-positive rates [Faw06] of the diagnosis outcome across all runs of the experiment for each injected fault, for each cluster-size. For each injected fault, we separately evaluated the ability of our algorithm to diagnose the fault using the durations of Map states, and using the durations of Reduce states. A true-positive is a node with a fault injected which was (correctly) indicted as faulty, while

a false-positive is a node with no fault injected but which was (wrongly) indicted as faulty. Hence, the true-positive ratio (TP) and false-positive ratio (FP) can be computed as follows:

$$TP = \frac{\text{\# faulty nodes correctly indicted}}{\text{\# nodes with injected faults}}$$
$$FP = \frac{\text{\# nodes without faults incorrectly indicted}}{\text{\# nodes without injected faults}}$$

We report results across 20 runs for each of the injected faults on 10-node cluster experiments, and 6 runs for each of the injected faults on 50-node cluster experiments, with half the experiment runs having Speculative Execution enabled, and half without. The diagnosis results were similar across both types of experiments with and without Speculative Execution enabled. Figures 7.1, 7.2 show the TP and FP rates of the white-box algorithm for each injected fault, for 10-node and 50-node clusters, using Map durations, and Reduce durations, respectively. The bars above the zero line represent the TP rates, and the bars below the zero line respresent the FP rates for each fault. We report the results for the diagnosis algorithm when used with durations of Map states, and when used with durations of Reduce states, separately, and we describe the efficacy of diagnosis for each injected fault.

Map Durations

The white-box diagnosis algorithm, using comparisons of the durations of Map states, was able to diagnose the CPU and Disk resource hog faults successfully, with TP ratios of 1.0, detecting all instances of nodes with the injected fault, and low FP ratios, misdiagnosing non-faulty nodes less than 15% of the time. It was also able to diagnose the HANG-1036 fault successfully, as this was a hang in the Map state. However, HANG-1152 and HANG-2080 were not diagnosed successfully, with low TP rates, as these were hangs in the Reduce state and generally did not affect Map state durations significantly; however, HANG-1152 and HANG-2080 were generally detected with higher TP rates on 50-node clusters than on 10-node clusters, and we speculate that this is due to the larger number of Maps in the jobs running on the larger 50-node cluster, resulting in improved diagnosis. The PacketLoss-50 fault was not successfully diagnosed on 10-node clusters, but was diagnosed (high TP ratio of ≈ 1.0), albeit not successfully localized (relatively high FP ratio of nearly 0.2), because the high rate of packet loss resulted in correlated fault manifestations, causing both the node with the injected packet loss and the other nodes

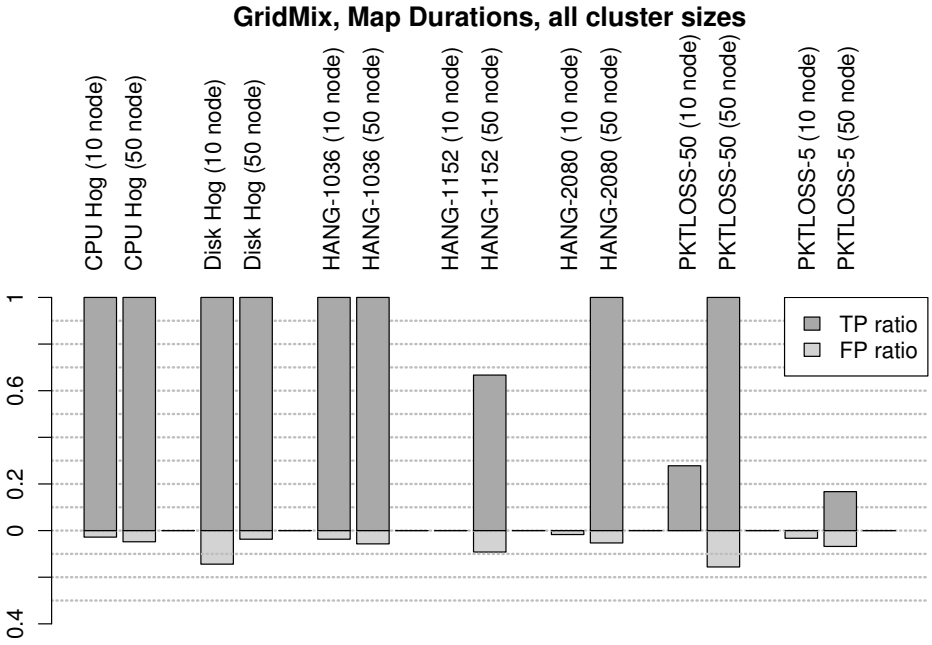

Figure 7.1: True-positive and False-positive ratios of diagnosis of white-box diagnosis algorithm using Map durations.

communicating with it to exhibit symptoms of the fault. The PacketLoss-5 fault was not successfully diagnosed, with low TP ratios, and we believe this is because TCP, which is used in Hadoop's network communications, are able to tolerate this rate of packet loss using its reliability mechanisms such as retransmissions. a 5% rate of packet loss

Reduce Durations

The white-box diagnosis algorithm, using comparisons of the durations of Reduce states, was not successful at diagnosing the CPU and Disk resource hog faults, with low TP ratios. This was because most of the time spent in the Reduce state was dominated by the time spent waiting for Maps to complete, in the ShuffleWait sub-state (see §7.2.2 for discussion of the ShuffleWait), so that the increased duration of the Reduce state computation that was affected by the resource hogs was insignificant as compared to the overall Reduce duration. The HANG-1036 fault was also not diagnosed successfully using Reduce durations, as HANG-1036 is a hang of the processing in the Map state. HANG-1152 and HANG-2080

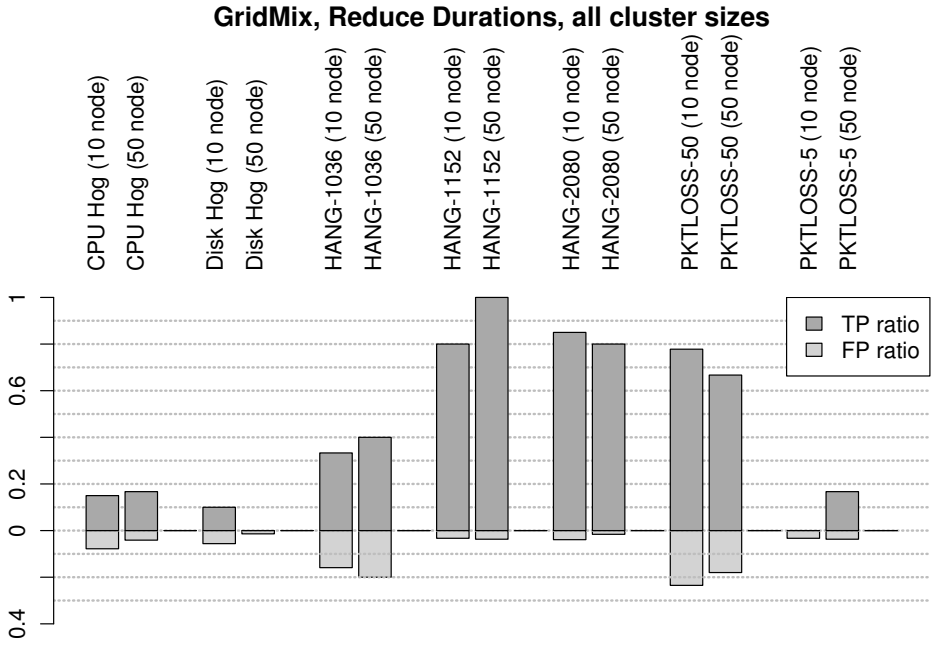

Figure 7.2: True-positive and False-positive ratios of diagnosis of white-box diagnosis algorithm using Reduce durations.

were successfully diagnosed, with TP ratios of > 0.8 and low FP ratios of < 0.1, as these were hangs in the Reduce state, and hence were naturally detected using durations of the Reduce state. PacketLoss-50 was again diagnosed but not localized, with high TP ratios of > 0.5 but high FP ratios of > 0.2 as well. Similarly with Map durations, PacketLoss-5 was not diagnosed, with low TP ratios, and again, we believe this is due to Hadoop's use of TCP, and TCP's ability to tolerate a 5% packet loss using its retransmissions and reliable delivery mechanisms.

Summarizing the results, the use of the white-box diagnosis algorithm in comparing Map durations was effective for diagnosing resource faults (CPU hog, Disk hog), and hangs in the Map state, while the use of the algorithm in comparing Reduce durations was effective for diagnosing hangs in the Reduce state but not resource faults nor hangs in the Map state; packet losses were generally difficult to diagnose because of the correlated nature of its fault manifestations. Thus, diagnosis using the white-box algorithm can potentially isolate the root-cause of a fault to the state of processing (Map or Reduce), in addition to isolating the faulty node.

7.2 Performance Debugging in the Wild

Next, we demonstrate the applications of our visualizations for performance debugging on production environment clusters, using case studies of various user workloads from actual users, as well as from synthetic traces consisting of benchmark workloads.

7.2.1 Testbed and Workloads

All traces used in our evaluation were collected from the Yahoo! M45 [Yah07] production cluster, which researchers from Carnegie Mellon have had access to for running research workloads such as large-scale machine learning algorithms, large-scale graph mining, text and web mining, natural language processing, machine translation, and data-intensive file-system applications. The M45 cluster has over 400 nodes, with approximately 4000 processors, 3 TB of memory, and 1.5 PB of disk capacity. The cluster runs Hadoop 0.18.3 at time of writing, and uses Hadoop on Demand (HOD) to provision virtual Hadoop JobTracker/TaskTracker clusters over the large physical cluster, and over the single monolithic HDFS instance.

The examples in § 7.2.2, § 7.2.3 involve 5-node clusters (4-slave, 1-master), and the example in § 7.2.4 is from a 25-node cluster. We omit case-studies involving more slave nodes as it would be difficult to present these visualizations in print without the benefit of dynamic rescaling and zooming of large, high-resolution images on-screen.

7.2.2 Understanding Hadoop Job Structure

Figure 7.3 shows the *Swimlanes* plots from the Sort and RandomWriter benchmark workloads (part of the Hadoop distribution), respectively. RandomWriter writes random key/-value pairs to HDFS and has only Maps, while Sort reads key/value pairs in Maps, and aggregates, sorts, and outputs them in Reduces. From these visualizations, we see that RandomWriter has only Maps, while the Reduces in Sort take significantly longer than the Maps, showing most of the work occurs in the Reduces. In addition, we can observe a common behavior in MapReduce programs, where tasks execute in "waves" as the number of tasks awaiting execution is larger than the number of available execution "slots" on slave nodes-here, the Reduces execute in roughly two waves. The *REP* plot in Figure 5.9 shows that a significant fraction ($\approx \frac{2}{3}$) of the time along the critical paths (Cluster 5) is spent waiting for Map outputs to be shuffled to the Reduces, suggesting this is a bottleneck.

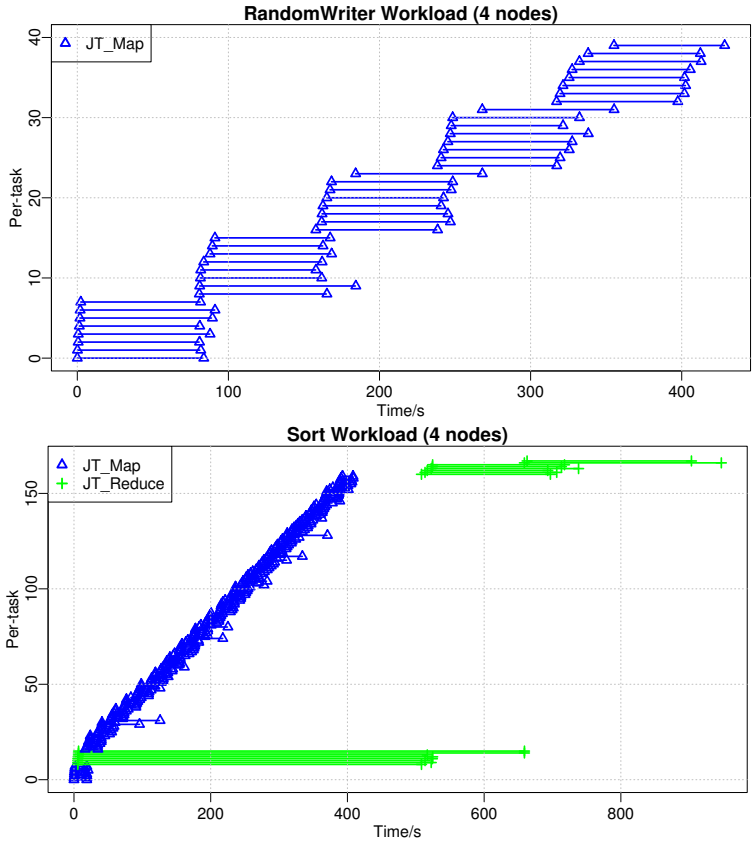

Figure 7.3: Summarized *Swimlanes* plot for RandomWriter (top) and Sort (bottom)

7.2.3 Performance Optimization

Figure 7.4 shows the *Swimlanes* from the Matrix-Vector Multiplication job of the HADI [KTA+08] graph-mining application for Hadoop. These experiments were run by CMU users, whose log data we performed post-mortem analysis on. This workload contains two MapReduce programs, as seen from the two batches of Maps and Reduces. Before optimization, the second node and first node do not run any Reduce in the first and second jobs respectively. The number of Reduces was then increased to twice the number of slave nodes, after which every node ran two Reduces (the maximum concurrent permitted), and the job completed 13.5% faster.

This insight was very quickly gleaned from the *Swimlanes* plots alone, and it was easy to verify that the program was more efficient from the subsequent plot after the change in configuration.

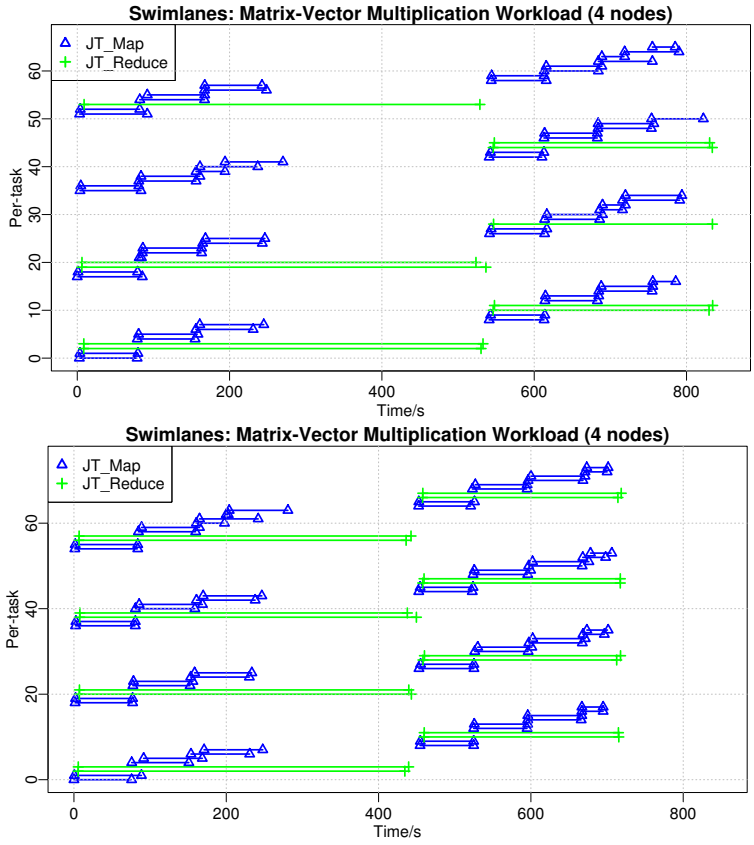

Figure 7.4: Matrix-vector Multiplication before optimization (above), and after optimization (below)

In addition, we studied the time spent in the ReceiveCopyWait stage where the Reduce waits while receiving Map outputs, and Figure 7.5 shows the target program also suffers from significant overhead in this stage, and our next optimization aim is to reduce this overhead.

7.2.4 Hadoop Misconfiguration

We ran a no-op ("Sleep") Hadoop job, with 2400 idle Maps and Reduces which sleep for 100ms, to characterize idle Hadoop behavior, and found tasks with unusually long durations. On inspection of the *Swimlanes*, we found delayed tasks ran for 3 minutes (Figure 7.6). We traced this problem to a delayed socket call in Hadoop, and found a fix

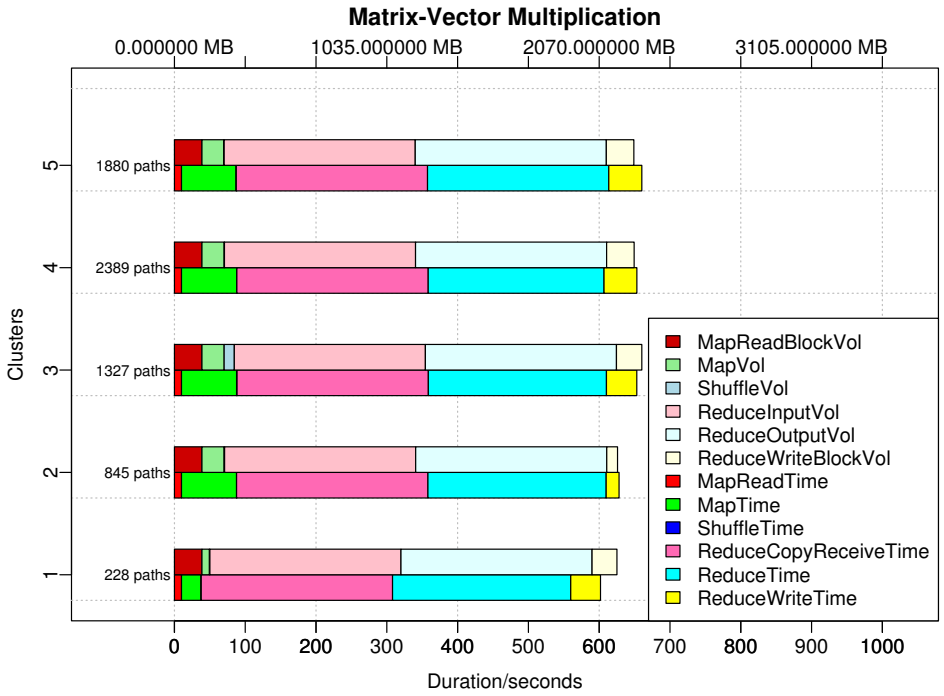

Figure 7.5: *REP* plot for Matrix-Vector Multiplication

described at [soc04]. We resolved this issue by forcing Java to use IPv4 through a JVM option, and Sleep ran in 270, instead of 520, seconds.

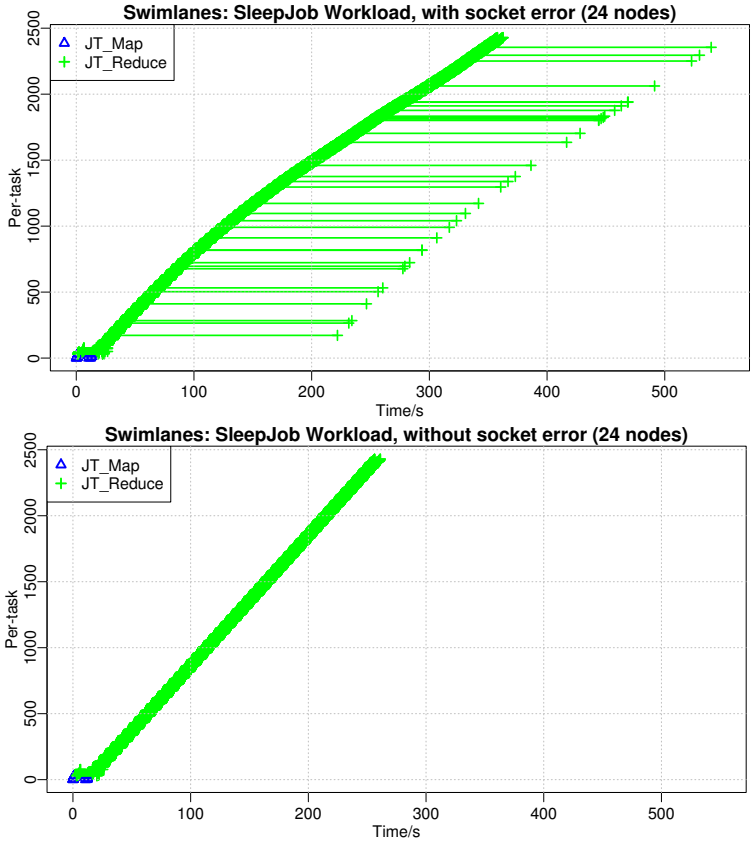

Figure 7.6: SleepJob with delayed socket creation (above), and without (below)

Chapter 8

Discussion

8.1 Implementation Notes

8.1.1 MapReduce Implementation of Job-Centric Data-Flow Extraction

We describe a proposed design for implementing the Job-Centric Data-Flow construction and corresponding Realized Execution Path extraction as MapReduce programs as part of our MapReduce implementation of our tool-chain, part of which is described in §6.1.2.

The Job-Centric Data-Flows can be constructed by performing the equivalent of a relational join[1] on the states generated by the state-machine builder. This directly generates the Realized Execution Paths by generating the cross-product of all control-items with all data-items, subject to actual causality within the system. These joins exactly relate the data-items with the relevant control-items as described in §5.1.2 as "stitching" the items.

The basic component of the JCDF construction is a MapReduce program which takes as its input a source tuple state type and a destination tuple state type, and joins states of the source type with states of the destination type based on a "join" identifier in a similar fashion to relational joins, where the source type is the state whose vertices in the JCDF are the source nodes of the edges in the graph, and the destination type is the state whose vertices are the destination nodes of the edges in the graph. Concretely, the MapReduce

[1] Arguably this can be implemented using tools which provide relational operators on top of HDFS-stored data such as HBase, Hive, or Pig, but a key design principle was for our tool-chain to work with only a plain Hadoop installation and Chukwa, and at the time of implementation, Chukwa support for running Pig jobs on `ChukwaRecord`s was not available.

program takes as its inputs source and destination state names, fields in the tuple to use as the primary key for each of the source state and destination state, and a field in the tuple to use as the key to join on.

The basic component is then run once for each pair of state types to join on; since there are seven pairs of state types to join, the JCDF consists of a pipeline of seven MapReduce jobs, each of which feeds its output to the input of the next job in the pipeline.

The intermediate Map output consists of keys with the value of the field to join on for tuples of the source state, and keys with the value of the field primary key of the destination state, since the joins will be of the source state on the destination state. The output values then consist of the results of the join, i.e., the cross-product of the source states with the destination states, and the output keys are a concatenation of the source key and the destination key to product a unique key for the output record. Hence, the *REP*s are incrementally constructed and each output record at the end of the pipeline of MapReduce jobs is an *REP* causality flow.

8.2 Lessons for Logging

8.2.1 Log Statement Evolution and System Diagnosability

We began this work in June 2007, with Hadoop 0.4.0, as distributed with Nutch 0.8 [Apa07d]. Our log parsing and analysis tools have been modified over the course of Hadoop's development, and have been tested and used successfully with Hadoop 0.4.0, 0.12.x, 0.14.x, 0.15.x, 0.17.x, 0.18.x, 0.19.x, and 0.20.x. However, this has required modifications to our parsing tools and analysis logic at various stages to evolve our analysis with the changes in log statements, which are typically not maintained as public and/or stable interfaces, especially in open-source software. However, there is currently a drive by the Hadoop project community to define the public and stable interfaces of Hadoop formally [RCL+09].

Some of these modifications have been minor, e.g. typographical errors in log messages, while some have required significant changes in the logic of our analysis. While some changes have hindered our analysis and rendered less information available to us, or required us to obtain the same information using alternate means, the general trend has been for logging to be more helpful to diagnosis.

In particular, as described in §5.1.2, the introduction of new `ClientTrace` log messages in the DataNode and TaskTracker logs which exactly log the data-flows as we de-

scribe, along with data volumes, durations, and communication endpoints [Xu09, TD09], which will be introduced in Hadoop 0.21.x, enables us to exactly extract complete causal paths.

In addition, the JobTracker-generated Job History logs for each job provide the information necessary for listing all the states in the control-flow of the execution of a MapReduce program, although this information is insufficient for construction the state-machine as information about Shuffles is not present in the Job History logs.

Hence, logging in a system, especially a distributed one, should be co-designed with the system itself to enhance the diagnosability of the system [TPK+08], and our work presents an example of log-enabled diagnosis and the potential of this approach to a semantically-rich, yet lightweight approach to diagnosis and performance characterization. Our use of log information provides a guide as to the types of information that would be useful to log, especially in a large-scale, distributed system, as well as the levels of abstraction at which information about the system would be useful.

8.2.2 SALSA for Log Compression

Next, we evaluate the size of our post-processed views to show the scalability of our approach based on our evaluation in §7.2. Table 8.1 lists the sizes of the raw logs generated by Hadoop in each of our experiments, and the sizes of the state-machine views generated by SALSA. Then, these views are correlated to form the JCDF and *REP* distributed views, and these sizes are listed next. The DataNode log sizes are the same regardless of workload because M45 uses a static shared HDFS instance with private TaskTracker clusters using Hadoop- on-Demand, so every user accesses the same HDFS instance. DataNode log sizes are dependent on the cluster workload on any given day, hence the sizes are reported as a range over the days our experiments were performed.

To summarize, we achieve significant size reduction in TaskTracker execution logs and minor reduction in DataNode logs, while the JCDF and *REP* sizes are dependent on job complexity. Sort is a deceptively simple workload-it involves a dense $M \times R$ exchange of data between M Maps and R Reduces, creating an extremely dense graph, and represents the worst-case complexity. That the 50-node Matrix-Vector Multiplication has four times fewer *REP* paths than the 5-node Sort workload suggests that real-world MapReduce programs are not likely to approach the worst-case complexity. The possibly large number of *REP* paths suggests that the Depth-first Search component of our path extraction has greatest room for optimization.

	Random Writer	Sort	Matrix-Vector Multiply	
Cluster size	5	5	5	50
TaskTracker				
Log File Size	428 KB	6.9 MB	4.9 MB	46 MB
Log Lines	2108	42,487	31,726	314,767
Parsed File Size	6 KB	567 KB	116 KB	3.5 MB
Parsed Lines	44	2634	556	15,582
DataNode				
Log File Size	204 ± 52 MB			
Log Lines	$1,374,382 \pm 296,660$			
Parsed File Size	155 ± 34 MB			
Parsed Lines	$1,252,264 \pm 180,546$			
JCDF, *REP*				
JCDF Vertices	N/A	3129	279	5624
JCDF Edges	N/A	5853	470	10670
REP Paths	N/A	160,540	6670	41,801

Table 8.1: Space savings from parsed log views

8.2.3 Formal Software Verification and State-Machines

Finite-State Automata have been used largely for formal verification of software systems using Model Checking techniques as pioneered by [CES86], with the correct behavior of software systems being defined as finite-state machines and model checking used to create a logic of actual program executions as possible in executable code, and to verify program executions against a definition of correct behavior. These model checking techniques rendered the use of formal logics to represent programs scalable, and the applications of formal software verification have ranged from verifying mission-critical applications to security systems. These applications use finite state-machines as a definition of correct behavior, and build state-machines of the execution of program code, and verify the two against each other.

Our use of the finite state-machine abstraction differs from that of formal software verification; while both formal software verification as well as our work uses a state-machine as a definition of correct program behavior, we do not attempt to verify the correctness of the implementation of the specified behavior. Rather, we assume the correct implementation of the behavior according to the state-machine (modulo interrupted executions, e.g. a crash in the Map state resulting in the rest of the state-machine not being executed),

and capture the performance properties of the system using the state-machine model as an abstraction, and augmenting the state-machine abstraction to have edges encode the execution times and volumes of data processed of each state (§4.1.2).

While it may be worthwhile to formally verify the correctness of the implementation of Hadoop and MapReduce programs, we believe that Hadoop is sufficiently mature that such verification is unnecessary, and that greater value lies in characterizing and evaluating the performance of MapReduce programs, and we have used finite state-machines as they are a convenient abstraction for informally reasoning about the performance characteristics of MapReduce programs, rather than in a formal sense, as in the domain of software model checking.

Chapter 9

Related Work

9.1 Log Analysis

9.1.1 Event-based Analysis

Many studies of system logs treat them as sources of failure events. Log analysis of system errors typically involves classifying log messages based on the preset severity level of the reported error, and on tokens and their positions in the text of the message [OS07] [LZS+06]. More sophisticated analysis has included the study of the statistical properties of reported failure events to localize and predict faults [OS08] [LZS+06] [HCSA07] and mining patterns from multiple log events [HMP02].

Our treatment of system logs differs from such techniques that treat logs as purely a source of events: we impose additional semantics on the log events of interest, to identify durations in which the system is performing a specific activity. This provides context of the temporal state of the system that a purely event-based treatment of logs would miss, and this context alludes to the operational context suggested in [OS07], albeit at the level of the control-flow context of the application rather than a managerial one. Also, since our approach takes log semantics into consideration, we can produce views of the data that can be intuitively understood. However, we note that our analysis is amenable only to logs that capture both normal system activity events and errors.

9.1.2 Request Tracing

Our view of system logs as providing a control-flow perspective of system execution, when coupled with log messages which have unique identifiers for the relevant request or processing task, allows us to extract request-flow views of the system. Much work has been done to extract request-flow views of systems, and these request flow views have then been used to diagnose and debug performance problems in distributed systems [BDIM04] [AMW+03]. However, [BDIM04] used instrumentation in the application and middleware to track requests and explicitly monitor the states that the system goes through, while [AMW+03] extracted causal flows from messages in a distributed system using J2EE instrumentation developed by [CKF+02]. Our work differs from these request-flow tracing techniques in that we can causally extract request flows of the system without added instrumentation given system logs, as described in § 4.1.1.

9.1.3 Log-Analysis Tools

Splunk [Spl05] treats logs as searchable text indexes, and generates visualizations of the log; Splunk treats logs similarly to other log-analysis techniques, considering each log entry as an event. There exist commercial open-source [Apa07a] tools for visualizing the data in logs based on standardized logging mechanisms, such as `log4j` [Apa07c]. To the best of our knowledge, none of these tools derive the control-flow, data-flow and state-machine views that SALSA does. SALSA represents a particular approach to processing logs specific to Hadoop and MapReduce-style processing systems, and can be implemented as post-processing modules for these logs types in these log-management systems.

9.1.4 State-Machine Extraction from Logs

[JCUY05] automatically inferred state-machine views of execution from textual logs emitted by a multi-tier J2EE web-transaction processing system, by identifying co-occurring log statements, similar to our principle of identifying for each state a start and end log token to identify the start and end respectively of the execution of a given state. However, [JCUY05] required significant amounts of training data from known-good fault-free runs to characterize normal traces in order to detect failed runs. Also, the use of [JCUY05] is to detect runs with state-machines with different shapes from normal runs, and thus targets incorrect execution, which is different from our use of the state-machine abstraction to encode performance characteristics. Also, the execution of a MapReduce program tends to be more complex and to generate much denser execution graphs than in the systems

examined in [JCUY05].

[LMP06] constructed fine-grained finite-state automata which modeled the execution of programs at the granularity of individual program statements, and introduced a technique for modeling the interaction between variables and method invocations. These state-machine models used are at a much finer granularity than our use of the state-machine abstraction at the coarser granularity of high-level logical blocks of execution (i.e. Maps and Reduces rather than individual program statements). Hence, such techniques, when used with MapReduce programs, would quickly run into scalability issues as discussed in §2.1.1. Also, [LMP06] is an extension of 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.

[MP08] examines textual logs of multi-tier J2EE web- transaction processing systems as well, and also automatically extracts legal behaviors and encodes interactions between program components using state-machine models. Similarly to [JCUY05], [MP08] uses a supervised learning approach, and requires input data from known-good runs in order to identify state-machine executions that are faulty. Also, [MP08] detects state-machines with shapes that are different from known-good runs.

Hence, the current tools that diagnose failures based on state-machine models extracted from logs differ from our work in that they detect state-machines that vary in shape and identify correctness failures, and the state-machine abstractions used are formal ones that do not encode any additional information other than the state-transition itself in their edges, whereas our work uses the state-machine abstraction informally, and encodes volume and time data on edges in order to enable the debugging of performance problems, which cannot be directly detected by these techniques.

9.2 Distributed Tracing and Failure Diagnosis

Recent tools developed to trace distributed program execution have focused on building instrumentation that can trace causal paths [BDIM04], assert causal relationships across disparate components [KJ08] and networks [FPK+07]. They produce fine-grained views at the language rather than MapReduce level of abstraction. Our work correlates system views from an existing instrumentation point (Hadoop system logs) to build views at a higher level of abstraction for MapReduce. Other techniques which use distributed execution traces [AMW+03, CKF+02, KF05] for diagnosis and debugging operate at the language level, as they worked with systems without a limited programming model unlike MapReduce, whereas we generate views at the higher-level MapReduce 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. [KF05, CKF+02, AMW+03, CZG+05] all assume the availability of SLO violations. However, as we explain in §2.1.3, such SLOs are not readily available in MapReduce systems because MapReduce programs are designed to be large batch jobs with potentially long runtimes.

[CKF+02] then goes on to identify the components responsible for causing a failure in a request where multiple processing components in the request generated error messages, and [KF05] also analyzes shapes of request paths in terms of components traversed in processing the request to identify anomalous paths; SALSA considers only "normal" path shapes in MapReduce processing and does not consider degenerate paths since the processing paths in MapReduce are determined by the framework rather than user programs, as compared to J2EE applications in which programmers can create arbitrary flows between J2EE components; also, SALSA focuses on latencies and volumes transmitted between processing (control-items and data-items) path elements whereas [KF05] considers path shapes.

[AMW+03] infers causal paths at the application layer from messages in the underlying messaging layer e.g. Remote Procedural Call (RPC) messages by interpositioning between the messaging layer and the application and capturing messages; our work differs in that we do not require interpositioning and are completely transparent to the system and we can perform post-mortem analysis even on uninstrumented Hadoop systems because we leverage Hadoop's natively generated logs. This suggests that logging that generates messages of a particular type can be a cheap and effective means of enhancing the diagnosability of a system. Also, [AMW+03] considers only latencies along path components, whereas we introduce the notion of considering data volumes along path components as well, a critical feature for data-intensive computing applications/systems such as MapReduce. However, our approach is designed specifically for MapReduce systems, and while it is likely to apply to other data-intensive systems, [AMW+03] is more general and would likely apply to a larger variety of systems.

In the Abacus project, [APGG00] performs dynamic placement of functions of an data-intensive application which manipulates large datasets that are stored on a cluster. They abstracted resource usage in the cluster using a data-flow graph, and this graph is then populated at runtime by monitoring the amount of bytes moved between application-level objects. Thus, like our work, [APGG00] also considers the volume dimension of

program behavior, although we unify all three dimensions of space, time, and volume, of program behavior in our Realized Execution Paths, while [APGG00] only considers the space-volume dimension.

9.3 Diagnosis for MapReduce

[KZKS08] collected trace events in Hadoop's execution generated by custom instrumentation-- these are akin to language-level views; while they provide summarized statistics of these events, their abstractions do not account for the volume dimension which we provide (§4.1.2), and they do not provide correlation with the MapReduce level of abstraction. [XHF+08] only showed how outlier events can be identified in DataNode logs; we utilize information from the TaskTracker logs as well, and we build a complete abstraction of all execution events.

[PTK+09] used black-box operating-system-level performance counters for diagnosis, as compared to the white-box application specific metrics we used for diagnosis; however, the peer-similarity hypothesis and approach is fundamentally similar to our diagnostic approach. [Pan09] used a combination of white-box and black-box metrics for diagnosis, putting together multiple algorithms, each using a different source of metrics for diagnosis, and imposed a supervised learning process over these algorithms to achieve finer-grained root-cause diagnosis of previously-known faults; the white-box diagnosis algorithm we present here is a component algorithm in the BliMeE framework in [Pan09].

In previous work, we also explored correlating white-box Hadoop log events with statistical changepoints in black-box operating system metrics in the BlackSheep approach [TN08], and we explored the hypothesis of correlation between activity in the privileged operating system-mode and unprivileged user-mode in fault-free conditions in the RAMS approach [TN08].

9.4 Visualization Tools

Artemis [CBG08] provides a pluggable framework for distributed log collection, data analysis, and visualization. We have presented specific MapReduce abstractions and ways to build them, and our techniques can be implemented as Artemis plugins. The "machine usage data" plots in [CBG08] resemble *Swimlanes*; *REP* shows both data and computational dependencies, while the critical path analysis in [CBG08] considers only computation.

[BFB+05], visualized web server access patterns and the output of anomaly detection algorithms, while we showed system execution patterns.

Other frameworks have also been built for managing the visualization and summarizing of the large amounts of data from large clusters [MMKN08, LBST08]. However, these frameworks collect general system metrics such as operating system counters, and deal with the problem of scalable visualization and presentation of general system data in a way that enhances operator understanding. This differs from our work in that 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, and we do not tackle the problem of visualization of general systems. Our tools can instead be used as specific plugins for these general frameworks.

In addition, a number of tools [TSS96, CTF+97, TSS98, WBS+00] were previously built for inserting instrumentation in, collecting trace data from, and visualizing metrics in distributed supercomputing applications built using Application Programming Interfaces (APIs) such as the Message Passing Interface (MPI) [GLS99] and the Parallel Virtual Machine (PVM) [GBD+94]. MPI and PVM provided much more low-level interfaces than MapReduce to programmers, and consequently allowed significantly more general programs, but these programs were also consequently harder to write because the MapReduce abstraction simplified programming by hiding the low-level communications. MPI and PVM enabled much finer grained control over the parallelization of the program, allowing programmers to program at the level of threads. As a result, these visualization frameworks exposed much lower-level details than our tools expose for MapReduce; this is simply an artefact of the differences between MapReduce and PVM and MPI. Also, such tools are not suitable for MapReduce programs since they present fine-grained details about program behavior that MapReduce programmers would not have control over as they have been abstracted away by the MapReduce model where they had been exposed in the MPI/PVM model.

"Area Charts"–Aggregate task count time-series

The authors of Hadoop have also released a simple parser for Hadoop's job history logs which produces a time-series of aggregate task counts per unit time, i.e. the number of Maps, Reduces, Shuffles (which we termed ShuffleWait) and Merges (a sub-state in the Reduce which pre-sorts data before the Reduce), and they have demonstrated what we call "Area Charts" [MO09] (to distinguish them from our *Swimlanes* charts in §5.3.1). A key benefit of our *Swimlanes* over the Area Charts are that we expose the exact scheduling behavior on tasks, and for a given Map, Reduce, or any other state, we can tell the exact

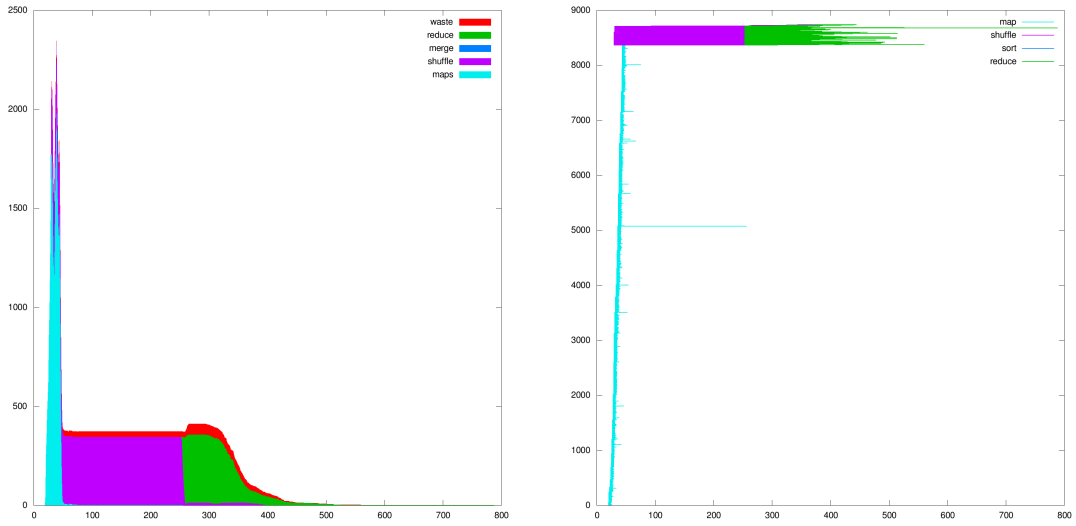

Figure 9.1: Comparison of the "Area Chart" (left) and *Swimlanes* chart of the same job.

start and end times immediately, whereas in the Area Charts, only counts are shown, as we demonstrate using an Area Chart and a *Swimlanes* chart for the same job in Figure 9.1, where the Map with long duration is shown more clearly in the *Swimlanes* chart, and the exact start-time of this Map with a relatively long duration is also shown clearly.

Chapter 10

Conclusion and Future Work

10.1 Conclusion

In conclusion, we have presented a log-based approach to characterizing and understanding the behavior of MapReduce programs and the Hadoop open-source implementation of Hadoop, for diagnosing failures and debugging performance problems in MapReduce programs. We have presented a novel abstraction of MapReduce program behavior based on Hadoop's logs, that takes into account both control-flows and data-flows in the program in a unified view called a Job-Centric Data-Flow, which is a directed graph of control- and data-items in Hadoop's execution, based on state-machine views of each Hadoop node's execution. We were then able to extract Realized Execution Paths from the Job-Centric Data-Flow, which are flows of causality annotated with the volumes of data processed and execution times of various data and control items along the path. We also presented a diagnosis algorithm based on the state-machine views of Hadoop's execution on each node, and using the durations of Maps and Reduces in programs, we were able to successfully diagnose resource hogs and hangs in Maps using the former, and hangs in Reduces using the latter. We also presented three visualizations of Hadoop's MapReduce program behavior in combinations of the dimensions of space, time, and volume, called *Swimlanes*, *MIROS*, and *REP* Volume-Duration correlation plots, as well as real-world use-cases of these visualizations in the production Yahoo! M45 cluster where we were able to optimize user MapReduce programs in a non-intrusive fashion.

10.2 Future Work

10.2.1 Causal Backtrace

We intend to augment the Realized Execution Paths to enable full causal backtracing of Maps and Reduces to the input data blocks and offsets in the input files. With the current *REP*s, for a given Reduce, it is possible to idenfity all Maps whose output it depends on, and to identify all data blocks that each Map read as an input. However, these data blocks are exposed in *REP*s by their block identifiers, but the reverse mapping from block identifier to file name and file offset in HDFS is currently not accessible to users and is only accessible as a data structure in the NameNode. We intend to explore ways of achieving this reverse mapping so that for a given Map or Reduce, it is possible to identify the data that the Map or Reduce depends on. Then, for instance, in the case of a crashed or hung Map or Reduce, it is possible for programmers to isolate the testing of the program to its behavior on the data contained in the specific data contained in the input file offset.

Also, this would enable us to analyze the performance of the Partition, which in MapReduce identifies which intermediate Map output keys are assigned to which Reduces, which is commonly identified as a common cause of data-skew and its resulting performance degradation in MapReduce programs. This would allow users to quantitatively observe the equity of key distribution of their Partition function to aid them in design better Partition functions.

10.2.2 Alternative *REP* Representations

We intend to explore other models and abstractions to represent the Realized Execution Paths which take into account the fact that the ShuffleWait state represents a synchronization barrier in the system. We hope to find models in which time can be encoded in terms of wall-clock time rather than durations that are not conforming to wall-clock time in our current representation (see discussion in §5.1.3). Possible alternative representations include tools from the project management domain, such as Gantt charts, which take into account scheduling artifacts and allow encoding of earliest/latest start/end times of tasks, as has already been used for MPI systems [WB02].

10.2.3 Anomaly Detection for *REP*

We intend to explore the detection of anomalous *REP* flows, to identify outlier flows that might not be visible due to the clustering performed on the *REP* flows prior to visualizing them. These outliers could correspond to flows which are causing performance problems, and together with the causal backtrace described in §10.2.1, would enable fine-grained debugging.

Appendix A

Appendix

A.1 Diagnosis Algorithm

Algorithm 1 Algorithm for exponentially-decayed histogram construction of state durations.

procedure SALSA-FINGERPOINT($prior$, $threshold_p$, $threshold_h$)
  for all i, initialize $distrib_i \leftarrow prior$
  for all i, initialize $lastUpdate_i \leftarrow 0$
  initialize $t \leftarrow 0$
  while job in progress **do**
    for all node i **do**
      while have new sample s with duration d **do**
        if $1 - CDF(distrib_i, d) > 1 - threshold_h$ **then**
          indict s
        end if
        $distrib_i \leftarrow distrib_i \times e^{-\lambda \frac{lastUpdate_i - t}{\alpha(lastUpdate_i - t)+1}}$
        add d to $distrib_i$ with weight 1
        $lastUpdate \leftarrow t$
      end while
    end for
    $COMPARE - HISTO(distrib, threshold_p)$
    $t \leftarrow t + 1$
  end while
end procedure

Algorithm 2 Algorithm for comparing histograms of state durations between states. Note: $JSD(distrib_i, distrib_j)$ is the Jensen-Shannon divergence between the distributions of the states' durations at nodes i and j.

procedure COMPARE-HISTO($distrib$, $threshold_p$)
  for all node pair i,j **do**
    $distMatrix(i, j) \leftarrow \sqrt{JSD(distrib_i, distrib_j)}$
  end for
  for all node i **do**
    if $count_j(distMatrix(i, j) > threshold_p) > \frac{1}{2} \times \#nodes$ **then**
      raise alarm at node i
      if 20 consecutive alarms raised **then**
        indict node i
      end if
    end if
  end for
end procedure

Bibliography

[Ama09] Amazon Web Services LLC. Amazon Elastic Compute Cloud, 2009. `http://aws.amazon.com/ec2/`. 1.1

[AMW+03] 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*, pages 74–89, Bolton Landing, NY, Oct 2003. 1.3, 9.1.2, 9.2

[Apa06] Apache Software Foundation. Apache's JIRA issue tracker, 2006. `https://issues.apache.org/jira`. 2.1.3, 7.1.2

[Apa07a] Apache Software Foundation. Chainsaw, 2007. `http://logging.apache.org/chainsaw`. 9.1.3

[Apa07b] Apache Software Foundation. Hadoop, 2007. `http://hadoop.apache.org/core`. 1.1

[Apa07c] Apache Software Foundation. Log4j, 2007. `http://logging.apache.org/log4j`. 9.1.3

[Apa07d] Apache Software Foundation. Nutch, 2007. `http://lucene.apache.org/nutch`. 8.2.1

[Apa08] Apache Software Foundation. Hadoop Users' Mailing List, 2008. `http://mail-archives.apache.org/mod_mbox/hadoop-core-user`. 2.1.2

[APGG00] K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamic function placement for data-intensive cluster computing. In *USENIX Annual Technical Conference*, San Diego, CA, Jun 2000. 9.2

[BDIM04] 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. 9.1.2, 9.2

[BFB+05] P. Bodik, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M. Jordan, and D. Patterson. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *ICAC*, 2005. 9.4

[BKK+08] K. Bare, M. Kasick, S. Kavulya, E. Marinelli, X. Pan, J. Tan, R. Gandhi, and P. Narasimhan. ASDF: Automated online fingerpointing for Hadoop. Technical Report CMU-PDL-08-104, Carnegie Mellon University PDL, May 2008. 6.1.2, 6.2

[BKQ+08] 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. 3.2, 6.1.2

[Bro09] J. Brodkin. Cloud Computing, Demystified (NetworkWorld), May 2009. `http://bit.ly/17p4Kd`. 1.1

[CBG08] G. Cretu-Ciocarlie, M. Budiu, and M. Goldszmidt. Hunting for problems with artemis. In *USENIX Workshop on Analysis of System Logs*, 2008. 9.4

[CES86] E. Clarke, E. Emerson, and P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986. 8.2.3

[CKF+02] 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. 1.3, 9.1.2, 9.2

[CTF+97] C. Carothers, B. Topol, R. Fujimoto, J. Stasko, and V. Sunderam. Visualizing parallel simulations in network computing environments: a case study. In *WSC '97: Proceedings of the 29th conference on Winter simulation*, pages 110–117, Atlanta, GA, 1997. 9.4

[CZG+05] 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, pages 105–118, Brighton, United Kingdom, Oct 2005. 1.3, 9.2

[DG04] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 137–150, San Francisco, CA, Dec 2004. 1.1, 3.1.1

[ES03] D. M. Endres and J. E. Schindelin. A new metric for probability distributions. *Information Theory, IEEE Transactions on*, 49(7):1858–1860, 2003. 5.2

[Fac08] Facebook. Engineering @ facebook's notes: Hadoop, Jun 2008. `http://www.facebook.com/note.php?note_id=16121578919`. 1.3

[Faw06] T. Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27:861–874, 2006. 7.1.3

[FPK+07] 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. 2.1.1, 9.2

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1st edition, Nov 1994. 1.1, 9.4

[GGL03] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *ACM Symposium on Operating Systems Principles*, pages 29 – 43, Lake George, NY, Oct 2003. 1.1, 3.1.1

[GLS99] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999. 1.1, 9.4

[HCL05] J. Heer, S. Card, and J. Landay. prefuse: A Toolkit for Interactive Information Visualization. In *SIGCHI Conference on Human Factors in Computing Systems (CHI)*, Florence, Italy, Apr 2005. 6.3.2

[HCSA07] Chengdu Huang, Ira Cohen, Julie Symons, and Tarek Abdelzaher. Achieving scalable automated diagnosis of distributed systems performance problems, 2007. 9.1.1

[HMP02] Joseph L. Hellerstein, Sheng Ma, and Chang-Shing Perng. Discovering actionable patterns in event data. *IBM Systems Journal*, 41(3):475–493, 2002. 9.1.1

[HPG02] J. Hennessy, D. Patterson, and D. Goldberg. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, May 2002. 4.1.2, 4.1.3

[JCUY05] 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. 9.1.4

[KF05] 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*, 16(5):1027– 1041, Sep 2005. 1.3, 9.2

[KJ08] Eric Koskinen and John Jannotti. Borderpatrol: isolating events for black-box tracing. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 191–203, New York, NY, USA, 2008. ACM. 9.2

[KTA+08] U. Kang, C. Tsourakakis, A.P. 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. 7.2.3

[KZKS08] A. Konwinski, M. Zaharia, R. Katz, and I. Stoica. X-tracing Hadoop. *Hadoop Summit*, Mar 2008. 9.3

[Lau09] O. Laursen. flot, 2009. `http://code.google.com/p/flot/`. 6.3.2

[LBST08] 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. 9.4

[LMP06] D. Lorenzoli, L. Mariani, and M. Pezze. Inferring State-based Behavior Models. In *Workshop on Dynamic Analysis (WODA)*, Shanghai, China, May 2006. 9.1.4

[LZS+06] Yinglung Liang, Yanyong Zhang, Anand Sivasubramaniam, Morris Jette, and Ramendra K. Sahoo. BlueGene/L failure analysis and prediction models. In *IEEE Conference on Dependable Systems and Networks*, pages 425–434, Philadelphia, PA, 2006. 9.1.1

[MMKN08] 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. 9.4

[MO09] A. Murthy and O. O'Malley. (Untitled parse script), 2009. `http://bit.ly/1KNFs`. 9.4

[MP08] 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. 9.1.4

[Mur08] Arun Murthy. Hadoop MapReduce - Tuning and Debugging, 2008. `http://tinyurl.com/c9eau2`. 1.2, 2.1.1

[Net08] Yahoo! Developer Network. Yahoo! launches world's largest hadoop production application (hadoop and distributed computing at yahoo!), Feb 2008. `http://developer.yahoo.net/blogs/hadoop/2008/02/yahoo-worlds-largest-production-hadoop.html`. 1.3

[OS07] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *IEEE Conference on Dependable Systems and Networks*, pages 575–584, Edinburgh, UK, June 2007. 9.1.1

[OS08] A. Oliner and J. Stearley. Bad words: Finding faults in Spirit's syslogs. In *8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008)*, pages 765–770, Lyon, France, May 2008. 9.1.1

[Pan09] X. Pan. The Blind Men and the Elephant: Piecing Together Hadoop for Diagnosis. Technical Report CMU-CS-09-135, Carnegie Mellon University Master's Thesis, May 2009. 9.3

[PTK+09] 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. 9.3

[R D08] R Development Core Team. R: A language and environment for statistical computing, 2008. `http://www.R-project.org`. 6.3.1

[RCL+09] S. Radia, D. Cutting, S. Loughran, J. Homan, and J. Tan. Hadoop 1.0 Interface Classification, 2009. http://issues.apache.org/jira/browse/HADOOP-5073. 8.2.1

[soc04] Creating socket in java takes 3 minutes, 2004. http://tinyurl.com/d5p3qr. 7.2.4

[Spl05] Splunk Inc. Splunk: The it search company, 2005. http://www.splunk.com. 9.1.3

[TD09] J. Tan and C. Douglas. Add ReduceID to Shuffle ClientTrace, 2009. http://issues.apache.org/jira/browse/MAPREDUCE-479. 8.2.1

[TN08] J. Tan and P. Narasimhan. RAMS and BlackSheep: Inferring white-box application behavior using black-box techniques. Technical Report CMU-PDL-08-103, Carnegie Mellon University PDL, May 2008. 9.3

[TPK+08] 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. 1.5, 4.1.1, 8.2.1

[TPK+09] 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. 1.5

[TSS96] Brad Topol, John T. Stasko, and Vaidy Sunderam. Monitoring and visualization in cluster environments. Technical Report GIT-CC-96-10, Georgia Institute of Technology, 1996. 9.4

[TSS98] B. Topol, J. Stasko, and S. Sunderam. PVaniM: A Tool for Visualizatoin in Network Computing Environments. *Concurrency: Practice and Experience*, 10(14):1197–1222, 1998. 9.4

[TSS+06] 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.*, 34(1):3–14, 2006. 2.1.1

[Was04] L. Wasserman. *All of Statistics: A Concise Course in Statistical Inference*. Springer, 1st edition, Sep 2004. 5.2

[WB02] C. Wu and A. Bolmarcich. Gantt Chart visualization for MPI and Apache multi-dimensional trace files. In *International Conference on Parallel and Distributed Systems (ICPADS)*, Chungli, Taiwan, Dec 2002. 10.2.2

[WBS+00] C. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *ACM/IEEE Conference on Supercomputing*, Dallas, TX, Nov 2000. 9.4

[XHF+08] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Mining console logs for large-scale system problem detection. In *Workshop on Tackling Systems Problems using Machine Learning*, Dec 2008. 9.3

[Xu09] L. Xu. Add I/O Duration Time in ClientTrace, 2009. `http://issues.apache.org/jira/browse/HADOOP-5625`. 8.2.1

[Yah07] Yahoo! Inc. Yahoo! reaches for the stars with M45 supercomputing project, 2007. `http://research.yahoo.com/node/1884`. 7.2.1