Mochi: Visual Log-Analysis Based Tools for Debugging Hadoop

Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, Priya Narasimhan

CMU-PDL-09-103
May 2009

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Mochi, a new visual, log-analysis based debugging tool correlates Hadoop's behavior in space, time and volume, and extracts a causal, unified control- and data-flow model of Hadoop across the nodes of a cluster. Mochi's analysis produces visualizations of Hadoop's behavior using which users can reason about and debug performance issues. We provide examples of Mochi's value in revealing a Hadoop job's structure, in optimizing real-world workloads, and in identifying anomalous Hadoop behavior, on the Yahoo! M45 Hadoop cluster.

Acknowledgements: The authors would like to acknowledge Christos Faloutsos and U Kang for discussions on the HADI Hadoop workload and for providing log data. This research was partially funded by the Defence Science & Technology Agency, Singapore, via the DSTA Overseas Scholarship, and sponsored in part by the National Science Foundation, via CAREER grant CCR-0238381 and grant CNS-0326453.

Keywords: Visualization, Log analysis, Performance Debugging, Hadoop, MapReduce

1 Introduction

MapReduce (MR) [7] is a programming paradigm and framework introduced by Google for data-intensive cloud computing on commodity clusters. Hadoop [9], an open-source Java implementation of MapReduce, is used by Yahoo! and Facebook. MapReduce has also been adopted as the infrastructure in Amazon's pay-as-you-use EC2 cloud computing infrastructure. Debugging the performance of Hadoop programs is difficult because of their scale and distributed nature. Hadoop can be debugged by examining the local (node-specific) logs of its execution. These logs can be large, and must be manually stitched across nodes to debug system-wide problems. Hadoop provides a `LocalJobRunner` that runs jobs in isolation for debugging, but this does not support debugging of large jobs across many nodes.

Current Java debugging/profiling tools (`jstack`, `hprof`) target programming abstractions to help debug local code-level errors rather than distributed problems across multiple nodes [16]. In addition, these tools do not provide insights at the higher level of abstraction (e.g. Maps and Reduces) that is more natural to MR users and programmers. Similarly, path-tracing tools [8] for distributed systems produce fine-grained views at the language rather than at the MR abstraction.

Our survey of the Hadoop users' mailing-list indicates that the most frequent performance-related questions are indeed at the level of MR abstractions. We examined the 3400 posts on this mailing list over a 6-month period from 10/2008 to 4/2009, and classified the 30-odd explicit performance-related posts[1] (some posts had multiple categories) in Table1. These posts focused on MR-specific aspects of Hadoop program behavior. The primary response to these posts involved suggestions to use Java profilers. However, these posts involved dynamic MR-specific behavior, such as relationships in time (e.g., the order of execution), space (which tasks ran on which nodes), and the volumes of data in various program stages. This motivated us to *extract and analyze time-, space- and volume-related Hadoop behavior*.

The MR framework affects program performance at the macro-scale through task scheduling and data distribution. This macro behavior is hard to infer from low-level 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 MR-specific abstractions. This motivated us to *capture Hadoop distributed data- and execution-related behavior that impacts MR performance*. 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. To the best of our knowledge, Mochi, the approach and implementation that we present here, is the first debugging tool for Hadoop to extract (from Hadoop's own logs) both control- and data-flow views, and to then jointly analyze and visualize these extracted views in a distributed, causal manner. We provide concrete examples where Mochi has been helpful in assisting us and other Hadoop users in understanding Hadoop's behavior and unearthing problems.

2 Problem Statement

Our previously developed log-analysis tool, SALSA [17], extracted various statistics (e.g., durations of Map and Reduce tasks) of system behavior from Hadoop's logs on individual nodes. Mochi aims to go beyond SALSA, to (i) correlate Hadoop's behavior in space, time and volume, and (ii) extract causal, end-to-end, distributed Hadoop behavior that factors in both computation and data across the nodes of the cluster. From our interactions with real Hadoop users (of the Yahoo! M45 [11] cluster), a third need has emerged: to provide helpful visualizations of Hadoop's behavior so that users can reason about and debug performance

[1] As expected of mailing-lists, most of the 3400 posts were from users learning about and configuring Hadoop (note that misconfigurations can also lead to performance problems). We filtered out, and focused on, the 30-odd posts that *explicitly* raised a performance issue.

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 mappers complete before reducers can run? What is the performance impact of setting X? What are the execution times of program parts?	50%

Table 1: Common queries on users' mailing list

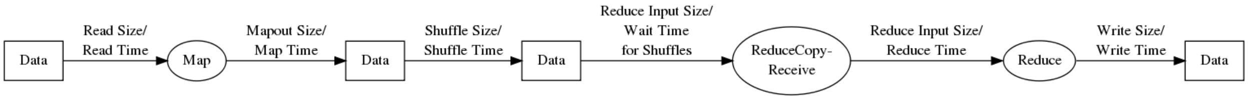

Figure 1: Single instance of a Realized Execution Path (REP) showing vertices and edge annotations

issues themselves.

Goals. Mochi's goals are:

To expose MapReduce-specific behavior that results from the MR framework's automatic execution, that affects Hadoop program performance but is either not visible/exposed to user-written Map and Reduce code. Examples include when Maps/Reduces are executed and on which nodes, from/to where data inputs/outputs flow and from which Maps/Reduces. Existing Java profilers do not capture such information.

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

To be transparent so that Mochi does not require any modifications to Hadoop, or to the way that Hadoop users write/compile/load their programs today. This also makes Mochi amenable to deployment in production Hadoop clusters, as is our objective.

Non-goals. Our focus is on exposing MR-specific aspects of user programs rather than behavior within individual Maps and Reduces. Thus, the execution specifics or correctness of code within a Map/Reduce is outside our scope. Also, Mochi does not discover the root-cause of performance problems, but aids in the process through useful visualizations and analysis that Hadoop users can exploit.

3 Mochi's Approach

MapReduce programs, or jobs, consist of Map tasks followed by Reduce tasks; multiple identical but distinct instances of Map or Reduce tasks operate on distinct segments of data in parallel across the nodes of a cluster. The framework has a single master node (running the NameNode and JobTracker daemons) that schedules Maps and Reduces on multiple slave nodes. The framework also manages the inputs and outputs of Maps, Reduces, and Shuffles (intermediate execution stages that move Map outputs to Reduces). Hadoop provides a distributed filesystem (HDFS) that implements the Google Filesystem [10]. Each slave node runs a TaskTracker (execution) and a DataNode (HDFS-related) daemon. Hadoop programs read and write data from HDFS. Each Hadoop node generates native logs that record the local execution of tasks and local accesses to data.

3.1 Mochi's Log Analysis

Mochi constructs cluster-wide views of the execution of MR programs from Hadoop-generated system logs. Mochi builds on our log-analysis capabilities to extract local (node-centric) Hadoop execution views. Mochi then correlates these views across nodes, and also between HDFS and the execution layer, to construct

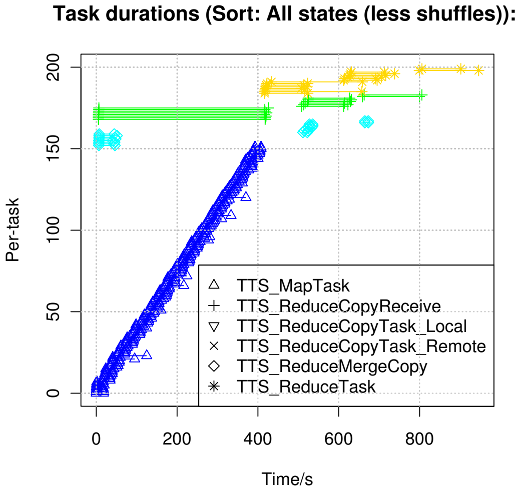

Figure 2: *Swimlanes*: detailed states: Sort workload

a unique end-to-end representation that we call a *Job-Centric Data-flow (JCDF)*, which is a distributed, causal, conjoined control- and data-flow.

Mochi parses Hadoop's logs[2] based on our knowledge of Hadoop's state-machine-like execution. In its log analysis, Mochi extracts (i) a time-stamped, cross-node, control-flow model by seeking string-tokens that identify TaskTracker-log messages signaling the starts and ends of activities (e.g., Map, Reduce), and (ii) a time-stamped, cross-node, data-flow model by seeking string-tokens that identify DataNode-log messages signaling the movement/access of data blocks, and by correlating these accesses with Maps and Reduces running at the same time. Mochi assumes that clocks are synchronized across nodes using NTP, as is common in production Hadoop clusters.

Mochi then correlates the execution of the TaskTrackers and DataNodes in time (e.g. co-occurring Maps and block reads in HDFS) to identify when data was read from or written to HDFS. This completes the causal path of the data being read from HDFS, processed in the Hadoop framework, and written to HDFS, creating a JCDF, which is a directed graph with vertices representing processing stages and data items, and edges annotated with durations and volumes (Figure 1). Finally we extract all Realized Execution Paths (REPs) from the JCDF graph–unique paths from a parent node to a leaf node– using a depth-first search. Each REP is a distinct end-to-end, causal flow in the system.

Thus, Mochi automatically generates, and then correlates, the cross-node data- and control-flow models of Hadoop's behavior, resulting in a unified, causal, cluster-wide execution+data-flow model.

3.2 Mochi's Visualization

Mochi's distributed data- and control-flows capture MR programs in three dimensions: space (nodes), time (durations, times, sequences of execution), and volume (of data processed). We use Mochi's analysis to drive visualizations that combine these dimensions at various aggregation levels. In this section, we describe the form of these visualizations, 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 §4.

[2] Mochi uses SALSA [17] to parse TaskTracker and DataNode logs. We can extend this to parse NameNode and JobTracker logs as well.

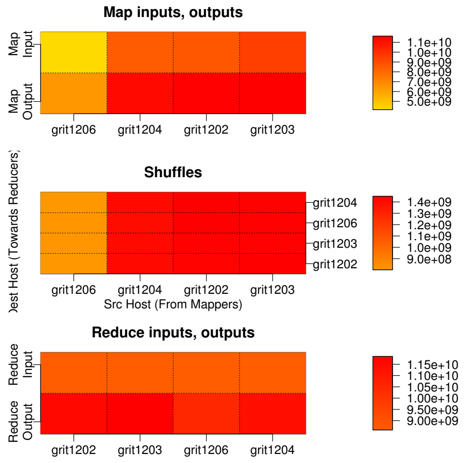

Figure 3: *MIROS*: Sort workload; graph shows volumes in bytes

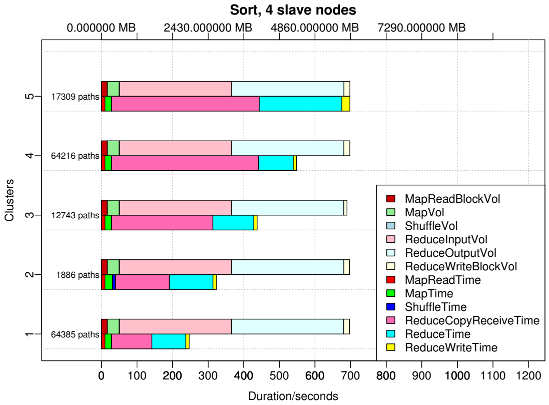

Figure 4: *REP* plot for Sort workload

"Swimlanes": Task progress in time and space. In such a visualization, the x-axis represents wall-clock time, and each horizontal line corresponds to an execution state (e.g., Map, Reduce) running in the marked time interval. Figure 2 shows a sample detailed view with all states across all nodes. Figure 5 shows a sample summarized view (Maps and Reduces only) collapsed across nodes, while Figure 6 shows summarized views with tasks grouped by nodes. *Swimlanes* is useful in capturing dynamic Hadoop execution, showing where the Hadoop job and nodes spend their time.

"MIROS" plots: Data-flows in space. MIROS (Map Inputs, Reduce Outputs, Shuffles, Figure 3) visualizations show data volumes into all Maps and out of all Reduces on each node, and between Maps and Reduces on nodes. These volumes are aggregated over the program's run and over nodes. MIROS is useful in highlighting skewed data flows that can create bottlenecks.

REP: Volume-duration correlations. For each REP flow, we show the time taken for a causal flow, and the volume of inputs and outputs, along that flow (Figure 4). Each REP is broken down into time spent and volume processed in each state. We used a *k*-Means algorithm to group similar paths for scalable visualization. For each group, the top bar shows volumes, and the bottom bar shows durations. 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.

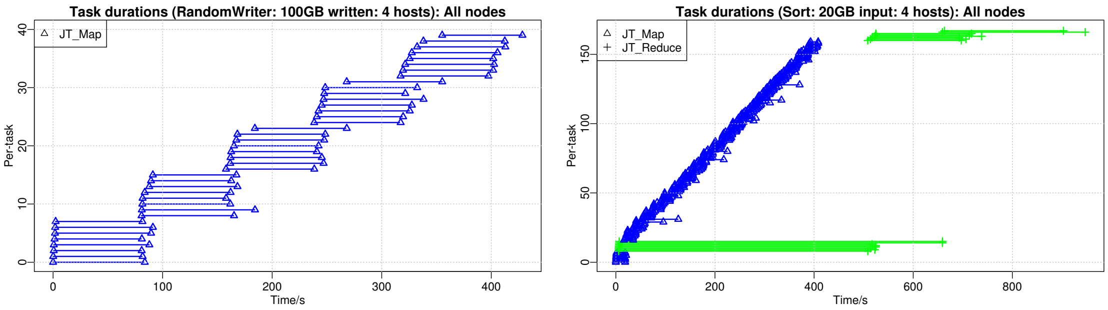

Figure 5: Summarized *Swimlanes* plot for RandomWriter (top) and Sort (bottom)

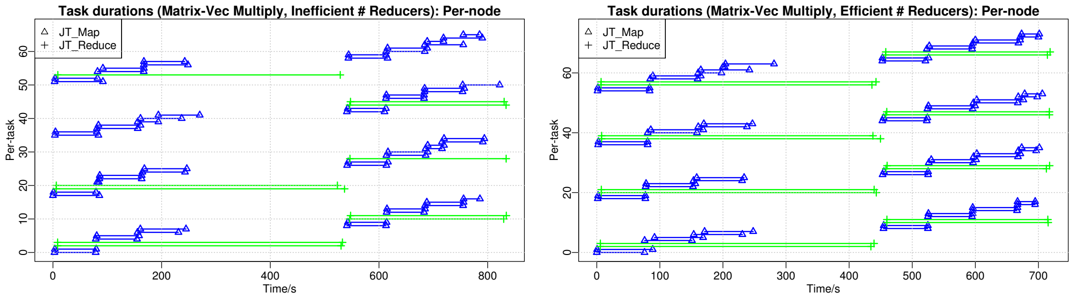

Figure 6: Matrix-vector Multiplication before optimization (above), and after optimization (below)

4 Examples of Mochi's Value

We demonstrate the use of Mochi's visualizations (using mainly *Swimlanes* due to space constraints). All of the data is derived from log traces from the Yahoo! M45 [11] production cluster. The examples in § 4.1, § 4.2 involve 5-node clusters (4-slave, 1-master), and the example in § 4.3 is from a 25-node cluster. Mochi's analysis and visualizations have run on real-world data from 300-node Hadoop production clusters, but we omit these results for lack of space; furthermore, at that scale, Mochi's interactive visualization (zooming in/out and targeted inspection) is of more benefit, rather than a static one.

4.1 Understanding Hadoop Job Structure

Figure 5 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. The *REP* plot in Figure 4 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.

4.2 Finding Opportunities for Optimization

Figure 6 shows the *Swimlanes* from the Matrix-Vector Multiplication job of the HADI [12] graph-mining application for Hadoop. This workload contains two MR 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.

4.3 Debugging: Delayed Java Socket Creation

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). We traced this problem to a delayed socket call in Hadoop, and found a fix described at [1]. We resolved this issue by forcing Java to use IPv4 through a JVM option, and Sleep ran in 270, instead of 520, seconds.

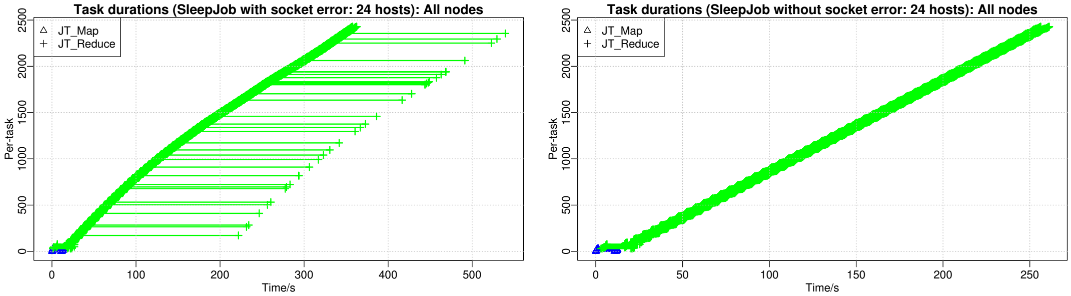

Figure 7: SleepJob with delayed socket creation (above), and without (below)

5 Related Work

Distributed tracing and failure diagnosis. Recent tools developed to trace distributed program execution have focused on building instrumentation that can trace causal paths [3], assert causal relationships across disparate components [15] and networks [8]. They produce fine-grained views at the language rather than MR 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 MR. Other techniques which use distributed execution traces [2, 5, 13] for diagnosis and debugging operate at the language level, as they worked with systems without a limited programming model unlike MR, whereas we generate views at the higher-level MR abstraction.

Diagnosis for MR. [14] 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 (§3.2), and they do not provide correlation with the MR level of abstraction. [18] 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.

Visualization tools. Artemis [6] provides a unified framework for distributed log collection, data analysis, and visualization. Artemis is a pluggable framework, while we have presented specific abstractions and ways to build them, and our techniques can be implemented as Artemis plugins. The "machine usage data" plots in [6] resemble our *Swimlanes*, although our *REP* plots show both data and computational dependencies, while the critical path analysis in [6] considers only computation bottlenecks. [4], visualized web server access patterns and the output of anomaly detection algorithms, while we showed system execution patterns.

6 Conclusion and Future Work

Mochi extracts and visualizes information about MR programs at the MR-level abstraction, based on Hadoop's system logs. We show how Mochi's analysis produces a distributed, causal, control+data-flow model of Hadoop's behavior, and then show the use of the resulting visualizations for understanding and debugging the performance of Hadoop jobs in the Yahoo! M45 production cluster. Our preliminary data shows that our condensed views of the Hadoop logs result in a 100-fold reduction in the original size of the logs. We intend to implement our (currently) offline Mochi analysis and visualization to run online, to evaluate the resulting performance overheads and benefits. We also intend to support the regression testing of Hadoop programs against new Hadoop versions, and debugging of more problems, e.g. misconfigurations.

References

[1] Creating socket in java takes 3 minutes, 2004. http://www.jaredoberhaus.com/tech_notes/2004/04/creating

[2] 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.

[3] 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.

[4] P. Bodik, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M. Jordan, and booktitle = ICAC year = 2005 D. Patterson, title = Combining Visualization and Statistical Analysis to Improve Operator Confidence and Efficiency for Failure Detection and Localization.

[5] 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.

[6] G. Cretu-Ciocarlie, M. Budiu, and M. Goldszmidt. Hunting for problems with artemis. In *WASL*, 2008.

[7] 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.

[8] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. X-Trace: A pervasive network tracing framework. In *USENIX Symposium on Networked Systems Design and Implementation*, Cambridge, MA, Apr 2007.

[9] The Apache Software Foundation. Hadoop, 2007. http://hadoop.apache.org/core.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

[16] Arun Murthy. Hadoop MapReduce - Tuning and Debugging, 2008. http://tinyurl.com/c9eau2.

[17] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. Salsa: Analyzing logs as state machines. In *Workshop on Analysis of System Logs*, San Diego, CA, Dec 2008.

[18] 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.