RAMS and BlackSheep: Inferring white-box application behavior using black-box techniques

Jiaqi Tan

May 2008

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Submitted in partial fulfillment of the requirements for the Senior Honors Thesis program in the School of Computer Science at Carnegie Mellon University

This research was made possible by the guidance of Priya Narasimhan

Abstract

A significant challenge in developing automated problem-diagnosis tools for distributed systems is the ability of these tools to differentiate between changes in system behavior due to workload changes from those due to faults. To address this challenge, current, typically white-box, techniques extract semantically-rich knowledge about the target application through fairly invasive, high-overhead instrumentation. We propose and explore two scalable, low-overhead, non-invasive techniques to infer semantics about target distributed systems, in a black-box manner, to facilitate problem diagnosis. RAMS applies statistical analysis on hardware performance counters to predict whether a given node in a distributed system is faulty, while BlackSheep corroborates multiple system metrics with application-level logs to determine whether a given node is faulty. In addition, we have developed and demonstrated a novel technique to extract, from existing application-level logs, semantically-rich behavior that is immediately amenable to analysis and synthesis with other numerical, black-box metrics. We have evaluated the efficacy of RAMS and BlackSheep in diagnosing real-world problems in the Hadoop distributed parallel programming system.

Contents

List of Figures

1 Introduction

Finding the location and root cause of a failure in a distributed system is an inherently difficult problem. Execution paths span multiple machines and can be arbitrarily complex. As a result, a fault may manifest itself as an error many execution modules down the execution path, before the error manifests itself as a failure, making the fault difficult to trace. Fault localization—tracing a system failure to the site of its initial manifestation as an error—requires either a characterization of externally observable correct system states, so that system states outside of this set are marked as erroneous, or a direct characterization of erroneous states. On the other hand, root-cause analysis—tracing a system error to its fault—requires detecting when software behavior deviates from the programmers' intentions. This requires knowledge of the semantics of the program, which is not present in the program.

We propose two new techniques for identifying the location and inferring the root-cause of a failure in a distributed system. These techniques attempt to infer semantically-rich white-box software behavior using black-box techniques. These techniques are designed to work in an online, scalable fashion that is amenable to use on production systems. They aim to address problem diagnosis on distributed systems with long-lived jobs, few user-initiated requests, and complex execution paths. While we do not immediately implement an online solution in this work, our approach has been carefully designed to ensure that the algorithms used are amenable to being run online with reasonable computation cost. We achieve this by using *a priori* knowledge of both distributed systems in general, and the deployed software, to build two classes of inference models. These models allow for white-box information of varying granularity about the phase of execution of software to be inferred from black-box information. In addition, our techniques require only intra-node information within a given node, so that these techniques are immediately scalable to distributed systems containing arbitrarily many nodes.

We designed and investigated the efficacy of two black-box techniques. *RAMS* (Regression Autocorrelation for detecting Malfunctioning nodeS) attempts to perform fault localization for Hadoop [25] by inferring coarse-grained white-box information about application behavior (i.e. whether the target system is malfunctioning) from black-box hardware performance counters. *BlackSheep* uses black-box techniques to corroborate black-box operating system-reported metrics and white-box application-level logs, for problem diagnosis in a candidate distributed system, Hadoop, with fine-grained white-box root-cause analysis.

We demonstrate the efficacy of our root-cause diagnosis technique on Hadoop, the open-source implementation of the Map/Reduce distributed parallel programming runtime environment and distributed filesystem [25], and further demonstrate the applicability of our technique where current problem diagnosis techniques are not immediately applicable, on Hadoop.

2 Background

There are two broad classes of techniques for analyzing systems and software. Black-box techniques treat the software system as an enclosed, unobservable entity that cannot be modified. We classify information sources that do not reveal the execution path inside software components as black-box, while we classify techniques that neither require source code nor machine code modification as black-box techniques. White-box information sources provide views into the internals and execution path of the software system. We classify information sources that provide knowledge of the original source code or execution path structure of the software, such as knowledge of the order of function calls, as white-box, while we classify techniques that require any form of source code modification as white-box techniques. While white-box information is a much wealthier source of information than black-box sources, there is typically an inherent trade-off between the richness of information that can be extracted from software, and the cost of gathering that information in terms of runtime overheads and ease of deployment. Black-box techniques are easy to use at existing software installations and typically involve setting up external software monitors that record general system state, but provide limited information. White-box techniques may involve significant initial programmer effort to insert source code such as assertions (which are only as good as the correctness of the assertions, creating a dual problem), and providing a fine granularity of information about control flow may have involve high overheads as large numbers of probe-and-record instructions will be needed.

It would appear that white-box techniques are necessary to trace a software error to the fault that is its root-cause. Faults occur when the software behavior deviates from the intentions of its programmers, and programmer intentions are reflected in the execution path at the granularity of control flow through functions. Current techniques have danced this tightrope of the inherent tension between instrumentation overheads and the amount of information that can be extracted, to try to find a good leverage on the smallest possible information source from which they can extract maximum diagnostic value .

Major black-box techniques have included Pinpoint, which instrumented the J2EE middleware platform to trace message flows between software components, to associate particular groups of components with erroneous transactions, and to find anomalous control flow paths [5]. Cohen et. al.'s work has focused on using clustering on black-box system metrics, and building informative summaries of metrics to reduce the amount of information that must be exchanged among the nodes of a distributed system to minimize bandwidth use [6], but can only detect the location but not the root-cause of anomalous behavior. Magpie correlated resource usage information from operating system-provided resource accounting facilities with output from application event logs to build causal paths of applications on a single node using clustering (that is extensible to multiple nodes, albeit at possibly high cost when tracing execution flows across large distributed systems) [2].

Major white-box techniques have included Pip, which relied on programmer-written expectations of correct behavior, and recorded alarms of anomalous behavior raised from within the software itself [16], but Pip is only as good as the programmer-written expectations it uses. Triage works on stand-alone (non-distributed) software to uncover the faulty source code behavior or system environment feature which caused a crash by using a re-execution framework combined with a trial-and-error automation of the intuitive human troubleshooting process [18], but this method is an after-the-fact technique that relies on the system being down to allow such root-cause discovery (rather than online diagnosis).

Current techniques which allow for root-cause analysis, such as Pip and Triage, require too much programmer input, which precludes the discovery of bugs that programmers are unaware of. Both Pip and Triage do not allow for runtime prognostics to be made for detecting errors before they have resulted in failures. Both Pip and Triage also require access to program source code, which may not always be feasible, especially at commercial production sites. Even black-box techniques such as Pinpoint are not necessarily suited to production sites, because Pinpoint requires a modified middleware, which production sites may not allow due to various concerns such as security, while techniques such as Cohen et al.'s work do not allow for root-cause analysis although it is amenable to deployment at production sites.

The goal of this work is to develop techniques for problem diagnosis on distributed software systems deployed in production environments. Production environments typically deploy commercial or otherwise third-party software packages for which source code is often not available. Production environments also typically have strict requirements on availability and quality of service—production systems strive to achieve maximum throughput and minimum latencies on servicing requests at a minimum cost. Production environments will generally prohibit modifying even program binaries for security and privacy concerns. Hence, intrusive and high-overhead white-box techniques are not amenable to our goal. Instead, we infer and extract white-box information using black-box techniques, to perform root-cause analysis in addition to fault localization.

2.1 Scalable Problem Diagnosis

The difficulty of finding the location and root cause of failures in distributed systems is further complicated by the fact that execution can take place on arbitrarily many systems, leading to an explosion in the volume of trace data gathered. Again, there is a trade-off between gleaning more information by combining trace data across systems to obtain a system-wide view, and incurring higher bandwidth and processing costs of transmitting large amounts of data across a network and processing it. This work studies one extreme of this trade-off, and uses only node-local information for problem diagnosis in a distributed system. We restrict ourselves to using only information available on a single node for diagnosis on the node, to push the boundaries of the efficacy of using only local information.

2.2 Problem Diagnosis using Multiple Data Sources

Our key to pushing the boundaries of using only node-local information for problem diagnosis is in the synthesizing of multiple data sources on the same node in a meaningful manner to gain additional information for problem diagnosis. To this end, we have examined various local information sources available at various levels of each node , such as hardware performance counters, operating system-reported metrics such as processor, memory, disk and network bandwidth utilization, and application-reported information such as logs. We make initial efforts at synthesizing this information for further analysis. This meaningful use of many data sources (as opposed to mathematically collapsing all the data for analysis using machine learning algorithms [6]), distinguishes our approach from current work which mostly use few (one or two) information sources [16] [5]. By preserving the meaning in the information sources, we are able to assist human operators by highlighting possible root-causes of the failure, in addition to localizing the fault.

2.3 Application Logs as a White-box Data Source

In addition, this work presents what is (to the best of the author's knowledge) a novel use of application activity logs, using application activity logs from the Hadoop distributed parallel programming platform as a case study. Since application-level log entries are programmer-reported statements of application behavior, they can be seen as a source of white-box information that provdes semantically-rich details about the activity of the application. Most current work on the use of application logs focuses on text-mining web access logs to analyze traffic patterns [21], and on text-mining error and access logs to discover pertinent features [17] [12]. However, the data mined from these logs using text-mining techniques is typically in an unstructured, multinomial (but not ordinal) form that cannot be immediately combined with operating system-reported metrics and performance counters, which are typically numerical, ordinal values, for analysis. The closest relative to our approach of interpreting event logs as a time-series of ordinal values is [10], although the author examined error logs rather than activity logs, and examined logs from a hardware source. The distinguishing features of our approach to application-level logs, as demonstrated through application activity logs from Hadoop, are: (i) our use of simple parsing instead of text-mining, (ii) inferring high-level white-box states of application execution, and (iii) generating structured data with a fixed number of descriptive numerical variables, each with ordinal values (counts of states), and that (iv) the structured, ordinal data we generate is immediately amenable to a larger range of analysis and machine learning algorithms.

We have built an online parser library for the application activity logs of the various components of the Hadoop platform that reports (i) significant white-box application events in the lifecycle of Hadoop, and (ii) the instantaneous workload/behavior state of Hadoop. The ability of our parsing algorithm to extract semantically-rich information about application behavior useful for problem diagnosis can also provide insight into how application logging can be designed to aid problem diagnosis of the application.

2.4 Problem Diagnosis for Hadoop

Distributed systems, such as Hadoop, and other Map/Reduce-type distributed parallel processing systems, are designed for batch processing of large datasets [25] [7], and are not amenable to problem diagnosis using most existing techniques.

These distributed systems see much fewer user-initiated requests, so that there are much fewer runs on which techniques such as Cohen's work, Magpie, and Pinpoint can perform clustering for learning the correct behavior of the system. Cohen et al.'s work, Magpie, Pinpoint, and Pip all assume the availability of large numbers of short-lived user-initiated requests, so that each of these requests can be used as a sample for clustering to determine which requests are anomalous. This model is well-suited to the vast majority of traditional multi-tier web-based applications, with common tiers being a web-server front-end, an application server tier, and a database back-end, but not to Hadoop.

Also, Hadoop has uninteresting execution paths through its components, as it implements a node-based processing model in which every node performs the same computation, rather than a path-based processing model in which each node along the processing path performs specialized processing. Thus, there is only one type of execution component (the TaskTracker), such that path-based techniques such as Pinpoint's Probabilistic Context-Free Grammars and Pip will have limited leverage from analyzing paths of execution flows for problem diagnosis.

Hence another key objective of our work is to use node-local, path-agnostic techniques for problem diagnosis on Map/Reduce-type distributed parallel processing systems for which current problem diagnosis techniques are not effective.

2.5 Hadoop failure scenarios

We studied 9 months of data (October 2006 to July 2007) from the bug database [23] of Hadoop, an open-source implementation of the MapReduce distributed parallel programming model, which motivated the characteristics of our target system, to identify common failure manifestations of faults. We found that the majority of faults manifested as process hangs and resource exhaustions. Out of 23 bugs surveyed from the Hadoop bug database, 11 resulted in process hangs in which no forward progress was made, 3 resulted in excessive CPU usage that slowed nodes down, 2 resulted in out of memory errors, while 7 resulted in application-level Java exceptions being thrown. Hence, we focused our fault-injection and problem-diagnosis efforts on detecting process hangs and memory leaks (in which objects that were allocated but which the system failed to dereference failed to be garbage collected, leading to out of memory errors).

3 Approach

3.1 Target System

Hadoop, an open-source implementation of Google's Map/Reduce infrastructure, handles a workload of long-running jobs that aim to process large datasets. Hadoop's master-slave architecture has a few ($O(1)$ in the number of slave nodes) master nodes coordinating many slave nodes which all have the same functionality. Master nodes provide two types of functionality in two separate daemons: the NameNode serves as the directory service for the Hadoop Distributed Filesystem (HDFS, a block-replicated filesystem that implements the Google Filesystem

(GFS) [8]), providing the mapping from named files to the slaves on which the individual fixed-size blocks are stored, while the JobTracker serves as the coordinator for Map/Reduce jobs. Similarly, slave nodes provide two types of functionality in two separate daemons: the DataNode serves as a chunk server in GFS terminology, providing actual storage of blocks, while the TaskTracker shepherds execution of tasks on slave nodes by starting up new Java Virtual Machines (JVMs) to execute tasks.

Long-running jobs are divided by the JobTracker on the master node into smaller, short-lived (relative to jobs) subtasks that are processed by the TaskTrackers on slave nodes. A job's subtasks are likely to be small relative to the job itself, in order to minimize the amount of re-computation when a node fails. We assume that the number of slave nodes can be large: any fault-tolerance techniques that warrant the remote inspection of nodes (e.g., through heartbeats) from a central/master location are likely to be costly in network bandwidth, thus the case for node-local diagnosis. We make no assumptions about the number of culprit nodes in the system, and do not currently probe further to discern which of the fingerpointed culprit nodes might be more to blame than the others. We focus our problem diagnosis efforts on node-local diagnosis on slave nodes, since these can be arbitrarily many while there are few master nodes; we first attempt to localize the fault to a single node, and make further efforts at localizing the fault to a phase of execution in the TaskTracker or DataNode on the slave node.

3.2 Manifestation-centric Problem Diagnosis: Goals and Non-goals

The actual root causes of performance problems are often difficult to diagnose without detailed application/domain knowledge. On the other hand, the manifestations of performance problems are observable errors or anomalous system activity, ultimately leading to system unavailability or unresponsiveness. Thus, our approach to problem diagnosis seeks to identify the culprit (node) of a performance problem by tracing any observed problem manifestations back to their source node. This also allows us to perform black-box problem diagnosis in a production setting, with neither access to nor modification of application source-code.

The goal of our work is to perform online problem determination: to locate, during the execution of the system, the node(s) on which a performance problem occurred, and to provide suggestions as to what the root-cause of the failure might be—these suggestions are in the form of system resource categories that are possible sources of performance issues, such as processing, memory, disk, and network resources. The eventual aim of this work is to expedite system recovery from a failure, either by aiding system administrators and operators in isolating faults and identifying recovery actions, or by providing diagnostic information for automated tools to decide the best course of action for system recovery.

In the context of the candidate failures identified in Section 2.5, the goal of our work is to flag off nodes exhibiting failure manifestations to isolate the failure, and to then provide informative metrics as suggestions as to what the root-cause of the failure might be.

Program debugging is a non-goal of our work. Our techniques are not intended to aid programmers in performing code-level analysis and extremely fine-grained localization of faults. Instead, our techniques bridge the gap between requiring access to and instrumentation of application source code for extremely fine-grained, code-level analysis, and using coarse-grained non-invasive black-box information sources by introducing (i) the use of statistical analysis to gain additional insights and enable inference about application behavior, and (ii) white-box information sources that can be accessed using black-box techniques.

3.3 Available Data Sources

The following main categories of sources of performance data about systems that can be accessed using black-box techniques, requiring no access nor modification to application source-code, have been utilized in our approach. A brief description of the type of information, the means of collection, and the cost of collecting each data source, in terms of overheads imposed, follows.

3.3.1 Hardware Performance Counters

Modern microprocessors implement performance counters to provide counts of hardware events, such as the number of unhalted cycles, or the number of cache hits and misses [22]. Hardware performance counters provide the lowest-level view of a system from the perspective of software, and provide a most fundamental (to the extent that collecting performance counter values causes minimal perturbations) view of the system closest to the ground truth of the bare metal of the system, free of artifacts as introduced by operating system or middleware-induced abstractions. The *RAMS* technique of our approach examines hardware performance counters to make inferences about the (correctness of the) behavior of the candidate application.

However, hardware performance counters can be potentially expensive to collect, as each data collection requires a context-switch into kernel mode to access the performance counter values. Nonetheless, our work uses the oprofile hardware performance counter monitoring package, which has a measured overhead of between 1-8% [29].

3.3.2 Operating System-reported Resource Metrics

The next higher level of abstraction from system hardware at which monitoring can be performed is the operating system. Major operating systems report aggregate statistics about various system resource categories—namely processing, memory, disk, network, and the virtual memory subsystem. These statistics are typically reported periodically as part of the service provided by the operating system, regardless of whether they are collected. Hence, these metrics can typically be collected in a low-overhead fashion. Specifically, Linux and many variants of Unix implement the proc filesystem, an interface through which a comprehensive array of operating system-provided information about aggregate system state and per-process state can be accessed.

The *BlackSheep* technique of our approach leverages on the low overheads of this data source and focuses on synthesizing the wide array of information available through the proc filesystem on Linux for problem diagnosis.

3.3.3 Application Logs

Many software applications, especially Internet-deployed and distributed software applications, have activity logs that describe the actions of the software application. Traditionally, application logs have provided a trace of error messages for system administrators and users to identify problems and for programmers to debug the application; application logs also sometimes provide a trace of accesses for audit trails to identify security breaches. These software applications typically have configurable levels of logging detail, so that they can be set to generate log messages about the software's actions with varying levels of verbosity. At the minimum level of verbosity, log messages are usually generated only in the event of fatal errors which caused the application to fail completely and crash, while at the maximum level of verbosity, log messages may be generated during the course of normal application execution as well, to report events.

Applications can be thought of as being in one of a finite number of high-level states, with each state corresponding to a particular mode in which a particular type of task is being executed, giving rise to a signature of that state which characterizes it. A key insight is that the normal application events as reported by detailed log messages will typically correspond to the entrance and exit of abstract application states as described above.

Hence, if an application has sufficiently few types of states, these states and the events which demarcate the entrance into/exit from abstract application states can be enumerated. Then, well-structured logs from the application can be parsed to process textual event reports to generate numerical counts of the states which the application is in. These numerical reports are more amenable to synthesis with other metrics, as they are all numerical and hence can be synthesized and treated with statistical analysis and learning as with typical numerical metrics.

3.4 Analytical Framework

Next, we provide an overview to the key ideas that the two techniques of our approach use for identifying deviant application behavior.

3.4.1 *RAMS*: an *a priori* Model of System Activity

The *RAMS* technique is based on the following hypothesized model of the local behavior of nodes in a distributed system.

The processing on slave nodes is always in one of two modes: (i) communication with the other nodes in the system, or (ii) actual data processing to compute a subtask. The user-space application (henceforth the application) invokes system calls to perform its external communication, which is recorded as operating system (OS), or kernel-space activity. Hence, under communication-intensive operations, a node's OS activity will dominate that node's application activity; conversely, under compute-intensive operations, the node's application activity will be higher.

The processing of a job's subtasks will likely involve repeated communication between nodes in the cluster—for subtasks to be dispatched to nodes, and for the results of subtasks to be returned to the dispatcher.

Consider a sufficiently long observation window on a node that encompasses both the communication phase of receiving and returning the results of the subtask as well as the computation phase to execute the subtask locally. Our hypothesis is that when a node experiences a performance problem, its processing is likely to be interrupted or to take significantly longer (possibly never returning), so that the node might not be observed (albeit indirectly) to be communicating as much with the other nodes in the system within the window of observation. Thus, we expect the system metrics that respectively characterize OS' and application's activity to be correlated in the absence of performance problems. OS activity and application activity will increase together in the same window, reflecting the external communication and the local computation required to process a subtask. However, when the node experiences a performance problem, we expect to see significantly less correlation between the node's OS' and application's activity, as either no computation nor communication are occurring, or the application is failing to return—in both cases, application activity moves independently of OS activity. An obvious side-benefit of this hypothesis (if indeed, it is borne out by experimental evidence) is that a node's observed local behavior alone ought to suffice for deciding whether that node is a culprit of a performance problem.

3.4.2 *BlackSheep*: Corroborating Application Behavior with System Activity

The *BlackSheep* technique is based on the key hypothesis that during normal, problem-free execution, the abstract state or mode of execution of the application should be approximately in line with the observed black-box metrics of the system.

We hypothesize that given normal execution, during a given mode of execution of the application, particular black-box metrics will exhibit stable patterns, such that changes in the mode of execution of the application will be followed by, possibly with a time lag, changes in the aggregate behavior of black-box aggregate system metrics.

Conversely, when there is a problem in the application, two scenarios are possible. First, there can be changes in the mode of execution of the application as reported by the application in its logs, but no changes in the aggregate behavior of system metrics, due to a failure of the application to transition to the new execution mode. Second, there can be changes in the aggregate behavior of system metrics although there was no change in the mode of execution of the application, as the change in system behavior was brought about by the transitioning of the application from its normal execution mode to a problematic one. By detecting the phase of execution at which an anomaly occurred (unexpected change in system activity, or unexpected absence of change in system activity, relative to application behavior), we can thus isolate the fault to a phase of execution in a particular component (DataNode/TaskTracker) of our target application.

Hence, the key idea behind *BlackSheep* is in quantifying changes in both the mode of execution of the application, and in black-box system metrics, and in identifying black-box system metrics whose changes co-occur with changes in the mode of execution of the application.

4 Application Log Parsing Case Study: Hadoop activity logs

There are four different types of activity logs provided by Hadoop: one for each of the four different types of daemons (NameNode, JobTracker, DataNode, TaskTracker) that provide services in Hadoop. Our initial efforts focus on the activity logs from the DataNode and TaskTracker. Hadoop uses the Apache Log4J [24] logging framework, thus emitting logs that are standardized across many other open-source software which also use this framework, suggesting that our approach is possibly portable to other applications also using Log4J (to the extent that the application developers of other applications using Log4J also provide log messages with similar semantic content as Hadoop does).

4.1 Log Overview

A snippet of log messages from the TaskTracker logs are shown in Figure 1. Log entries are timestamped, and the level of verbosity and the originating component of the log entry are stated, followed by a descriptive message. The Log4J framework used by Hadoop allows the destination of log messages to be configured; we assume that the default configuration of Log4J in Hadoop is used, so that log messages are written to plain text files.

Our Hadoop log parser parses each log message into its timestamp, the level of logging verbosity, the reporting component, and its message, and parses the log message to generate application events, from which application states are inferred, as described next.

```
2008-04-22 08:53:10,347 INFO org.apache.hadoop.mapred.TaskRunner:
task_0003_r_000000_0 Copying task_0003_m_004566_0 output from pc69.emulab.net.
2008-04-22 08:53:10,349 INFO org.apache.hadoop.mapred.TaskRunner:
task_0003_r_000000_0 Copying task_0003_m_001577_0 output from pc73.emulab.net.
2008-04-22 08:53:10,358 INFO org.apache.hadoop.mapred.TaskRunner:
task_0003_r_000000_0 done copying task_0003_m_004566_0 output from
pc69.emulab.net.
2008-04-22 08:53:10,436 INFO org.apache.hadoop.mapred.TaskRunner:
task_0003_r_000000_0 done copying task_0003_m_001577_0 output from
pc73.emulab.net.
```

Figure 1: A snippet from a TaskTracker log showing log entries which trigger $StateStartEvent$s and $StateStopEvent$s for the $ReduceCopyTask_Local$ and $ReduceCopyTask_Remote$ states.

4.2 Application Views: Events and States

4.2.1 Events and States

In order to interpret the semantic meaning of application logs (in a manner useful for problem diagnosis), we propose two orthogonal ways of viewing the high-level modes of execution of applications in general: as **events** and **states**, using Hadoop as a case-in-point: Consider each single thread of execution in an application as a deterministic finite automaton (a transition must be taken at each step, and the machine can be in at most one state) which is in exactly one DFA state at each time instant. Then, the mode of execution of the application can be viewed as **states** of the DFA, or as **events**, which are related to the transitions in the DFA, as explained next.

We define **states** in the DFA to correspond to high-level tasks (e.g. serving a block read request to a remote client in a DataNode), and **events** to be the entering and exiting of states, from which we derive transitions in the DFA to correspond to a composition of one state-entrance and one state-exit event. We define the two types of events as StateStartEvents and StateStopEvents. Then, multi-threaded applications would comprise multiple threads of execution, with one DFA representing the execution mode of each thread. The mode of execution of the application at each time instant can then be represented by (i) a vector of states in each of the DFAs, with one for each thread of execution, showing the instantaneous composite workload of the application, or as (ii) a vector of events in each of the DFAs, showing the changes that have taken place in the system at the time instance.

4.2.2 Events and States in Logs for Hadoop

A key observation about the log messages in Hadoop is that they correspond to notifications about the events as defined above. At the highest level of logging verbosity, they precisely denote the starting and stopping of each high-level task (e.g. Maps, various Reduce phases, block reads/writes served) undertaken by the DataNode and TaskTracker. Hence, our model of high-level application behavior can be directly parsed and extracted from the activity logs of Hadoop.

There are, however, exceptions: only the occurrence, and not the entrance to and exit from certain states in the DataNode and TaskTracker are reported, presumably because the tasks corresponding to these states are short-lived; we define a third event type, an InstantStateEvent, for transient states for these types of states. Events of this type, when composed with an event before it and an event after it, then corresponds to a transition into the state, followed by an immediate transition out of the state, in the context of a DFA.

A list of states for the DataNode and the TaskTracker are included in Appendix A, and each state has two associated events: one for the entrance to the state, and one for the exit from the state, and an InstantStateEvent is included for states whose execution is reported only in a transient manner.

4.3 Parsing Algorithm

The log parser implements a discrete window over the activity log. The log entries reported in each window of time under consideration are processed to return the *Events* occurring in the window. In addition, the *Events* occurring in the window are processed to return the *states* that the application is in within the window of consideration.

Log entries are read sequentially in strictly increasing chronological order, and parsed to assign an *Event* to each log entry. An *Event* may be one of { *StateStartEvent, StateStopEvent, InstantStateEvent, NoOpEvent, ErrorEvent*}, with the last two events added for log entries extraneous to our analysis that do not describe any significant change in workload (such as idle heartbeat messages, or a message indicating no useful work is being done), and for error messages, respectively. Thus, a time series of events can be immediately generated from the Hadoop activity logs (for DataNodes and TaskTrackers presently) with a single forward pass over the log entries.

In order to generate the vector of *states* that the application is in for each window, the log parser maintains internal state to remember the number of *StateStartEvent*s and *StateStopEvent*s that it has seen for each *state* at each time instance. Then, the number of threads executing each *state* is simply the difference between the number of *StateStartEvent*s and *StateStopEvent*s, plus the number of *InstantStateEvent*s seen for each *state*, within the given window.

A minor complication arises with the *StateStartEvent*s for the *ReadBlock* and *WriteBlock* states in the DataNode-—*StateStartEvent*s for most states in the logs of DataNodes are denoted by a generic message, while state-specific information is available only in *StateStopEvent*s. Hence, we make the simplifying assumption that *StateStartEvent*s and *StateStopEvent*s corresponding to the same *state* occurrence occur in FIFO order to make processing possible. This also implies that for any given *StateStartEvent*, the state it corresponds to cannot be identified before its (assumed) corresponding *StateStopEvent* is observed. *StateStartEvent*s in the DataNode logs are given an additional designation as *DeferredStateStartEvent*s to indicate that the identity of the state corresponding to the event has not been ascertained, and the window is prevented from sliding forward until the identity of the *StateStartEvent* has been resolved (by observing a corresponding *StateStopEvent*).

4.4 Parser architecture

The log parsing algorithm has been implemented as a library of C++ calls that can be easily reused in a larger software framework.

All logs are represented by a generic base class, which defines functionality common to manipulating each type of log, from which subclasses are derived and implemented for specific log types. Each log-specific subclass (e.g. DataNode, or TaskTracker) then implements its own monolithic parser to parse log entries from that particular type of log. The log base class stores a chronologically ordered list of *Events*, with the identity of the *Event* stored as an enumeration, and its associated *State* stored as a member variable. *State*s are defined by a generic base class, from which subclasses are defined for *State*s specific to each different logged component of Hadoop. The log-specific subclasses then implement functionality for processing a given list of events associated with states specific to the particular type of logged component to generate time series' of observed events and application states.

The log parser has a modular architecture, which exposes a common interface for sampling events and states from the different types of logs produced by Hadoop. A query object accepts a log object and calls on the log-specific event-processing method to generate samples of occurring *Event*s or samples of *State*s that the application is in. The query object manages the window over which sampling is performed, and manages the formating and presentation of reports of observations.

The log parser library supports on-demand, lazy parsing, and only needs to remember the latest log entry to perform processing; all information from prior log entries is summarized and stored as internal representations as lists of *Event*s and *State*s, and users of the library can explicitly request the library to clean up past reported events and states.

4.5 Offline Parser Output

In addition, the log parser provides an offline output mode, in which the log parser is provided with a sampling interval, and the parser generates a comma-separated value (CSV) file of a time series of the counts of each of the states and events for the particular node type—the number of each of the states and events for the particular node type in each sampling time interval is listed as a row in the CSV file. A visualization of these states is shown in Figures 22, 23, 24, 25.

5 *RAMS*: Statistical Tests of an *a priori* Model of System Activity

5.1 Analytical Methodology

5.1.1 Linear regression model of system activity

On each node in the system, we collect traces of the intra-node performance counter values for OS activity, os_t, and application activity, app_t (as discussed later). Consider a linear ordinary least-squares regression fitted to the time-series of the node's OS' performance counters (os_t) and the application's performance counters (app_t), with Gaussian noise u_t allowed:

$$os_t = \beta_{app} app_t + u_t$$

Concretely, (t) pairs of observed OS and application performance counter values form a window, and these are plotted as a function of each other, and a straight-line which minimizes the sum of squared errors between observed os_t and fitted os'_t is plotted through these points. Thus, for each pair (app_t, os_t), fitted values os'_t and the noise, or regression residual, $u_t = os'_t - os_t$, are generated from the regression.

5.1.2 Autocorrelation of residuals

Next, consider first-order lagged residuals, u_{t-1}, (i.e. consider the residual from the preceding pair in the time-series for each given time) and residuals u_t, from the linear regression. When a node is not experiencing problems, u_{t-1} will be independent of u_t, if the window over which regression is performed includes samples from both the communication-intensive and compute-intensive phases of the system. This is because the strong correlation between OS and application performance counters results in a strong relationship between the regressand (os_t) and regressor (app_t), so that residuals u_t reflect purely Gaussian noise and are uncorrelated.

When nodes are experiencing performance problems, there will be correlation between u_t and u_{t-1}. This is because application activity becomes increasingly uncorrelated with the OS activity, so that u_t and u_{t-1} become correlated. The regression residuals will reflect movements in the application activity counts and hence are no longer random noise, but become correlated.

This statistical condition in which residuals (u_t) become correlated with their lags (u_{t-i} for $i > 0$) is autocorrelation.

Hence, we hypothesize that autocorrelation between lagged residuals in an observation window exists on a node if and only if the node experiences performance problems in that window.

5.1.3 Autocorrelation tests for identifying anomalous nodes

The Breusch-Godfrey and Durbin-Watson [13] tests for autocorrelation were used to detect autocorrelation in the linear regressions of OS with application performance counter values for problem diagnosis. In each of these tests, the ordinary least-squares linear regression is first fitted to the window of observed OS and application performance counter values, from which secondary regressions and test statistics are computed based on the regression residuals. Each of these is a statistical test, which tests the null hypothesis that there is no autocorrelation present against the alternate hypothesis that there is autocorrelation present in the regression residuals, and returns a p-value—the probability of wrongly rejecting the null hypothesis.

Since our hypothesis is that autocorrelation is present in a regression on a given window of performance counter values if and only if the node is a problem node, the null hypothesis of these two statistical tests is exactly the (statistical) hypothesis that the node is problem-free. Thus, a smaller p-value indicates greater confidence that a problem is present in the node.

5.2 Experimental Setup and Methodology

We conducted a series of experiments to test the *RAMS* hypothesis. Our goal was to study the characteristics of the time-series of metric traces of every node's OS' and the application's activity under normal execution and under induced performance problems.

5.2.1 Setup

We deployed a 6-node (5 slave, 1 master) Hadoop 0.4.0 cluster on two 3.0GHz Xeon nodes, each running the Xen 3.1.0 hypervisor [3] hosting three unprivileged Linux 2.6.18 guests, on the Emulab [20] remote testbed. The Nutch (version 0.8.1) web-crawler [28], running on a Linux 2.6.18 guest hosted on a third 3.0GHz Xeon node over a Xen hypervisor, was used to generate workloads for the Hadoop cluster. Each iteration of the experiment involved rebooting all of the nodes in the Hadoop cluster, running a single Nutch web-crawling request, and collecting performance-counter traces over the duration of the execution. Each iteration of the experiment lasted approximately as long as the execution of the Nutch web-crawling job of 40 minutes.

5.2.2 Fault Injection

As Hadoop is written in Java, we used a JVM Tool Interface (JVM TI) agent [27] to perform load-time class bytecode-rewriting to inter-position calls to methods in our problem-injector class before the actual methods of interest. As our problem injection uses Java methods within the same virtual machine as the target application, all problem-injection activity is encompassed in the activity of the target application.

One of the two types of problem manifestations at one of two levels (high and low) of intensities was injected into three of the five slave nodes in each of the problem-induced iterations of the experiment. 72 iterations of the experiment were run, of which 27 iterations had memory leaks injected (11 high-intensity, 16 low-intensity) and 45 had process delays injected (14 high-intensity, 31 low-intensity).

Process-delay injection involved a `while` loop running for a preset duration—an infinite loop in the high intensity case, and alternation between executing the loop for one second and yielding control in the low-intensity case. The memory-leak injection involved allocating Java objects and adding them to a persistent vector, for a preset duration, in a similar manner to the process-delay injection described above.

The injected problems are representative of the manifestations of real-world performance problems recorded in the Hadoop bug database, as described in Section 2.5. As our problem diagnosis approach is a manifestation-driven one, being able to detect the identical manifestation would be a sufficient benchmark for our technique.

5.2.3 Instrumentation and Data Collection

The intra-node metrics that we gathered were Intel P4 performance-counter counts of instruction cycles collected by `oprofile` [29] with the `xenoprof` [14] Xen driver. Samples of instruction-cycle counts were taken at 10s intervals by `oprofile`, and attributed to the Linux processes whose instructions accounted for the cycle counts. In particular, we examined the counts for the Linux kernel process and the aggregate activity counters for the Java Virtual Machine (JVM) processes of the multiple Hadoop components.

5.2.4 Analysis

We analyzed the collected metrics offline after completing the experiments. For each node in each iteration of the experiment, the time-series of instruction-cycle counts for the Linux kernel (os_t) and the JVM (app_t) were fitted to the linear regression:

$$os_t = app_t + u_t$$

Next, we ran the Breusch-Godfrey and Durbin-Watson [13] tests for autocorrelation between the first-order lags of residuals (u_t, u_{t-1}), generating the *p*-values for the probability that there is no serial correlation between the first-order lagged residuals for each node. The *p*-values are used as a measure of serial correlation between the first-order lagged residuals. Then, specific *p*-value thresholds were used to identify the culprit node(s). Nodes with *p*-values below the threshold value were classified as being the culprits. Various *p*-value thresholds were used to vary the recall of the algorithm, to study how precision varied with recall (see Section 5.3 for definitions of precision and recall).

5.3 Evaluation Results

5.3.1 Statistical Characteristics of Metrics

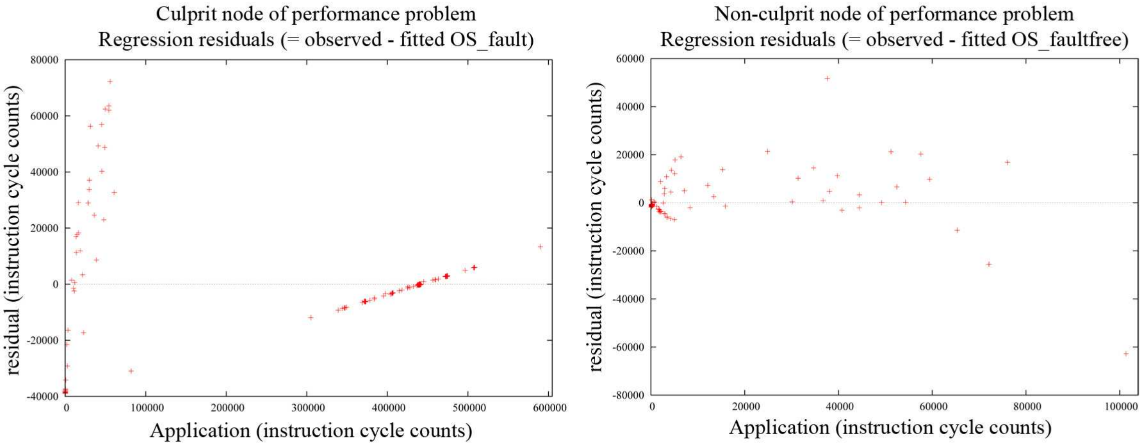

Figure 2: Regression residuals as a function of application activity for a culprit node with a low-intensity process delay (left) and a non-culprit node (right)

In the case of process delays, in nodes with injected problems (the culprit nodes), the residuals of the linear regressions of OS activity (os_t) with application activity (app_t) were strongly correlated with application activity, indicating strong autocorrelation in the (lagged) residuals. This is seen in the clear linear, non-zero relationship between the residuals and application activity in the left graph in Figure 2, while in problem-free nodes, the residuals showed no clear relationship with application activity, as seen in the right graph in Figure 2. This observation is consistent with our hypothesis. This observation was also confirmed by the statistically significant evidence of autocorrelation between the residuals in the culprit nodes, in contrast with the lack of such autocorrelation in the problem-free nodes.

However, in regressions for the experiments with injected memory leaks, there appeared to be no clear difference in the correlation patterns between the regression residuals and the application activity across the culprit and problem-free nodes.

5.3.2 Efficacy of Problem Diagnosis

Next, we examine the effectiveness of our approach at classifying culprit and problem-free (non-culprit) nodes.

Figures 3, 4 shows the performance of our initial problem-diagnosis algorithm for each type of failure, broken down by failure intensity. We quantified the efficacy of our approach using *precision* and *recall*, measures of classification effectiveness from the data-mining literature [19]. When our problem-diagnosis algorithm indicts a node, that node becomes a suspect; this is different from the node being truly guilty, i.e., a *culprit*. Precision measures the fraction of all suspects that are indeed culprits, while recall measures the fraction of culprits that our algorithm successfully indicted. We tuned the recall of our approach by varying the *p*-value threshold (Section 5.2.4), where

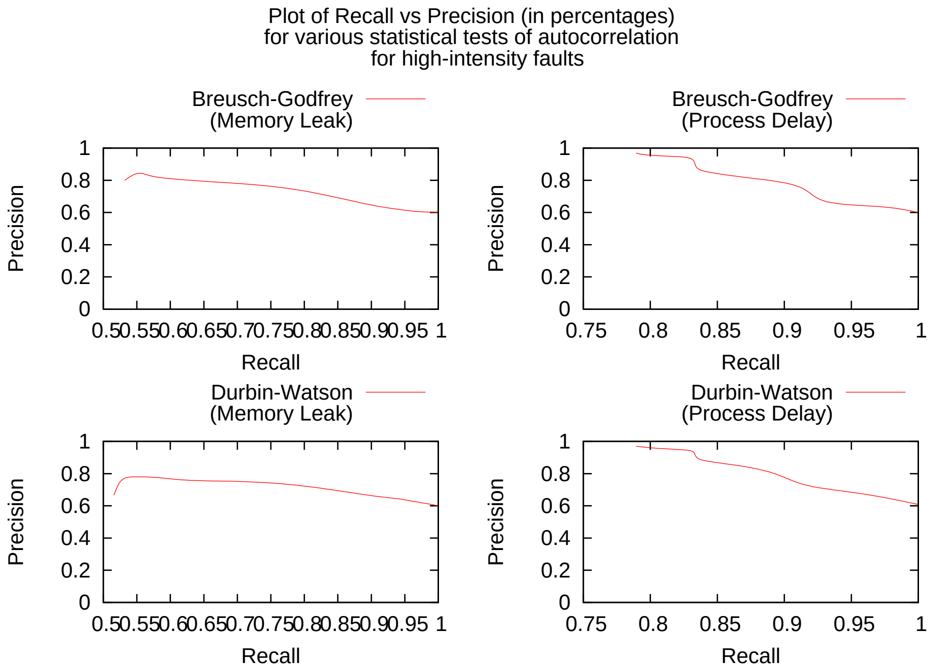

Figure 3: Overall precision as a function of recall for failure diagnosis algorithm for high-intensity failures

a *p*-value threshold of 1.0 results in our algorithm indicting all of the nodes in the system. As the number of suspects increases with more aggressive indictment (higher *p*-value thresholds), recall increases, but precision suffers. A perfect problem-diagnosis algorithm would have a precision/recall curve with a precision of 1.0 for all values of recall [11].

From Figure 3, our algorithm has some success identifying nodes suffering high-intensity problems—precision falls gradually and does not suffer a complete collapse as recall is increased, while our algorithm has some success with identifying nodes with low-intensity process-delay problems (Figure 4), but is ineffective at identifying nodes with low-intensity memory leaks, as precision collapses when recall is increased.

Perhaps a more informative statistic is *Balanced Accuracy* (BA) [11], the average of the proportion of problem-induced and problem-free nodes that were correctly classified. If problems occurred randomly, a random classifier would, in the limit, achieve a balanced accuracy of 0.5. Figures 5, 6 show the highest BA achieved by our algorithm under high- and low-intensity problems for memory leaks and process delays across all *p*-value thresholds used for each of the problem-intensity and problem-type cases shown (these are upper bounds on the efficacy of the algorithm; further work is needed to find the best single threshold value for all problem types and intensities). From Figure 5, our approach is moderately effective at identifying nodes with high-intensity problems and low-intensity process-delays, achieving a BA of greater than 0.7 using both (Breusch-Godfrey and Durbin-Watson tests) measures of autocorrelation. However, from Figure 6, our approach does only marginally better than a random classifier for nodes with low-intensity memory leaks.

In conclusion, we have shown that *RAMS* is effective at detecting both types of injected high-intensity faults, process hangs and memory leaks, and is somewhat effective at detecting low-intensity process hangs, but not much better than random at detecting low-intensity memory leaks. *RAMS* shows some promise at being able to identify anomalous nodes exhibiting process slowdowns and hangs, which would be helpful for detecting the large proportion of Hadoop bugs that manifest as process hangs (as surveyed in Section 2.5).

6 *BlackSheep*: Application-System Corroboration through Change Point Analysis of System Activity

6.1 Analytical Methodology

The fundamental idea of *BlackSheep* is that application logs and operating system-reported resource metrics (which we will refer to as resource metrics) provide orthogonal views of a system that should agree with each other at a high level during problem-free operation. Application logs provide semantically rich information about the high-level

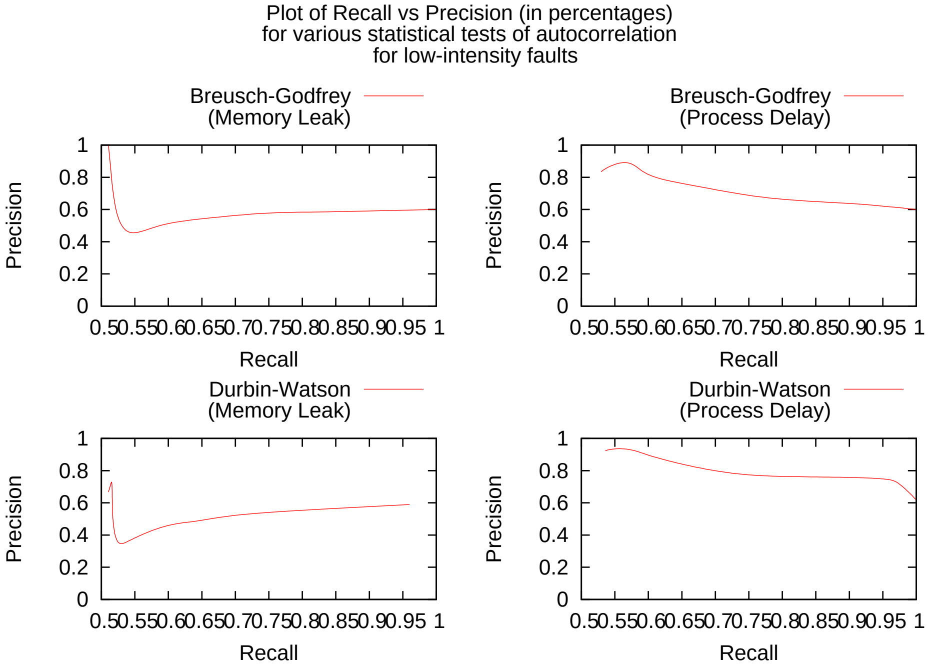

Figure 4: Overall precision as a function of recall for failure diagnosis algorithm for low-intensity failures

modes of execution of the application, while operating system metrics such as disk, memory, processor, and network utilization provide evidence of the actual behavior of the application as observed from its system-level actions. Thus, we would expect the high-level activities that the application reports itself as performing to correspond with its actual system-level actions during problem-free execution. An immediate consequence is that multiple views of the system disagreeing with each other is an indication of a problem. Then, the system resources whose metrics disagree with the view provided by application logs will provide suggestions as to which area of the application is not behaving as the high-level log information suggests the application should be.

Application logs typically contain textual information, while operating system-reported resource metrics are typically sequences of observed counts, and are not immediately comparable to determine if they agree with each other. However, this textual information in application logs can be parsed to extract counts of high-level states, each of which corresponds to a logical task performed by the application, and events, which correspond to the beginning and ending of states, as we have demonstrated for Hadoop in Section 4. These counts of high-level states and events can then be compared with operating system-reported metrics.

Change point analysis is applied to resource metrics and application state counts to compare them with each other for determining if an anomaly is present in the system.

6.1.1 Change point analysis

Operating system-reported resource metrics and counts of high-level application states are compared by computing the change points in resource metrics and application state counts, and ensuring that they occur together. We define the steady state of both the application and the system resources to be durations of absences of change points. The intuition is that when the counts of the number of occurrences of each state remain unchanged, the application is in a steady state, so that its system-level behavior as reflected by resource metrics should be steady and unchanging as well. Hence, we make the *a priori* assumption that a change in resource metrics when the application is exhibiting a steady state is anomalous, and likewise, that a change in application behavior when system resources are exhibiting a steady state is anomalous.

Change point detection is a classification problem over a time series of values. The objective is to separate the time series of values into contiguous segments, in which the underlying parameters describing the distribution of values is the same within each segment. A challenge of using popular change point detection algorithms such as Shewart control charts and CUSUM [4] are that they require at least one of either of (i) the parameters governing the distribution of values before the change, (ii) after the change, or (iii) the time of change—however, it is not clear what the correct parameters of the process generating the distribution of values of resource metrics are, and the

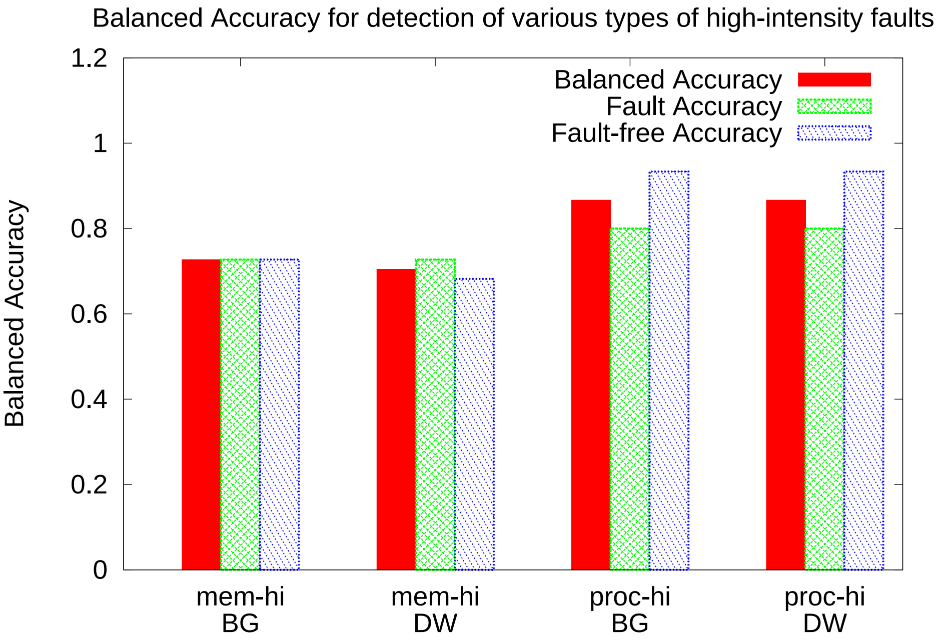

Figure 5: Balanced Accuracy (BA) of failure diagnosis algorithm for different failure types; BG indicates the Breusch-Godfrey and DW indicates the Durbin-Watson tests for serial correlation; hi indicates BA for high-intensity variants of problems injected

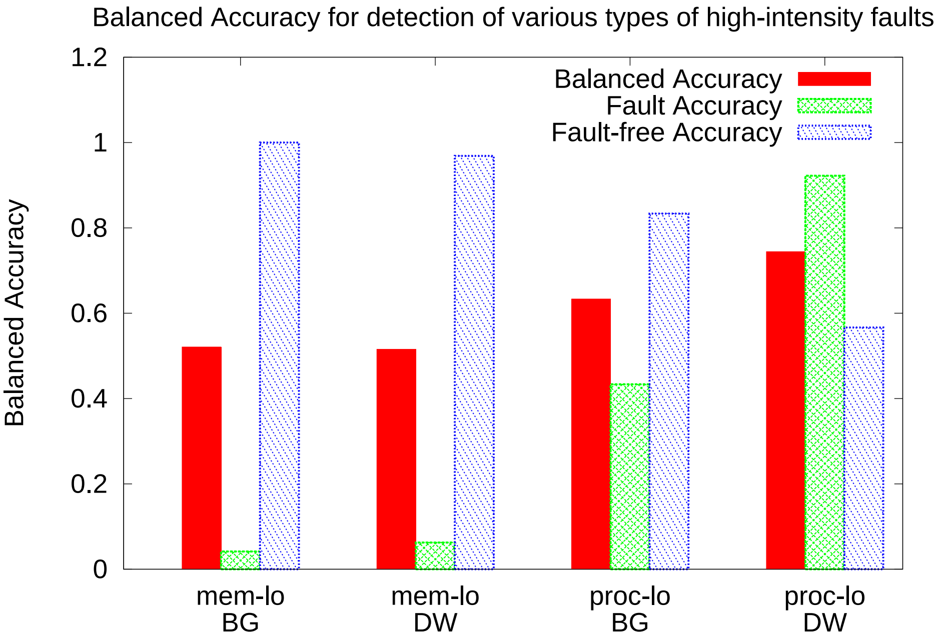

Figure 6: Balanced Accuracy (BA) of failure diagnosis algorithm for different failure types; BG indicates the Breusch-Godfrey and DW indicates the Durbin-Watson tests for serial correlation; lo indicates BA for low-intensity variants of problems injected

objective is precisely to determine the time of change, so that popular change point algorithms are not amenable.

Instead, we use a difference of means algorithm that is a variant of an image edge detection technique [1]. This algorithm had been previously successfully applied to problem determination in enterprise middleware systems, but on individual resource metrics in a group of metrics. Our use of change points analysis differs from [1] in that we are continuously comparing change points across two completely orthogonal measurements (application state counts and resource metrics) to corroborate change points.

6.1.2 Change Point Detection Algorithm

Our change point detection algorithm is described as follows:

Algorithm 1 Decision function for deciding if a given observation in a time series is a change point

```
 1: procedure CHANGEPOINT($obs[], obsnum, window, thresh, prevobs[], prevmax$)    ▷ $prevmaxobs$ is a
    fixed-size persistent queue that stores the last $window$ differences of means
 2:     $$\mu_L \leftarrow \frac{\sum_{i=obsnum-window}^{obsnum-1} obs[i]}{window}$$
 3:     $$\mu_R \leftarrow \frac{\sum_{i=obsnum+1}^{obsnum+window} obs[i]}{window}$$
 4:     $\Delta\mu = \mu_L - (-\mu_R)$
 5:     $prevobs.queue(\Delta\mu)$
 6:     if $(max(prevobs) == \Delta\mu)$ && $(\Delta\mu > prevmax)$ then
 7:         if $\Delta\mu > thresh * \mu_L$ then
 8:             $prevmax = \Delta\mu$
 9:             return true
10:         end if
11:     end if
12:     return false
13: end procedure
```

Algorithm 1 takes as its input a time series of metric values (which can be resource metrics or application state counts), and an observation number ($obsnum$) in the time series, and returns $true$ if the given observation number is a change point (i.e. statistical properties of the value in the time series changed at the time of the given observation), and $false$ otherwise. For a given window size, $window$, the left and right means (μ_L, μ_R) respectively are computed over $window$ samples before and after the given observation. The criteria for determining when the time of the observation is a change point is when the difference between the left and right means ($\Delta\mu$) for the given observation is a local maximum, and exceeds the value of the left mean, μ_L, by a given threshold factor, $thresh$. The first conditional in Line 6 ensures that $\Delta\mu$ is indeed a local maximum over the observation window. An additional heuristic is included in the second conditional in Line 6 to ensure that the same local maxima is reported only once.

The size of the observation window, $window$, and threshold factor $thresh$, are tunable parameters of the detection algorithm, and adjust the sensitivity of change points being reported. We then apply the change point detection algorithm to each time series for each resource metric and each application state count of interest, and compare the change points in the two, to determine if the execution of the application is free of anomalies. This comparison is detailed next.

The change point detection algorithm can be implemented in a lazy, dynamic fashion for online use, by keeping only state that has size that is constant ($O(1)$) in the order of the length of the time series examined, so that the algorithm can be run indefinitely with a constant amount of memory. The only persistent state required by the algorithm is $window$ number of $\Delta\mu$ values, and the value of the previous reported local maxima, $prevmax$.

However, a disadvantage of a lazy, greedy implementation is that continuously rising metric values will result in successive change points being reported, rather than a single local maxima—fortunately, our analysis involves comparing change points across resource metrics and application state counts, which are hypothesized to behave in similar ways in the event of normal operation. Thus, to the extent that our hypothesis will be borne out, this artifact of lazy evaluation does not skew our analysis. Analytically, in this case, a sequence of successive change points implies that the observed value of the metric of interest is changing with a magnitude outside of the threshold of a change detection, and that the change is taking place at an increasing rate; this can be interpreted as a continuous change taking place.

6.1.3 Corroborating system activity change points with application log events: Tests for anomalous system behavior

The next stage of the approach is to corroborate change points in resource metrics with application log events. The algorithm by which we corroborate the two orthogonal system views is as follows:

Algorithm 2 Decision function for deciding if two orthogonal views of the system agree with each other; $changepoint$'s are boolean flags indicating if a change point occurred in the time series of the resource metric or application state count respectively.

```
procedure StateMetricNormalDecision($changepoint_{metric}, changepoint_{state}$)
    if ($changepoint_{metric} == true$) && ($changepoint_{state} == true$) then
        return $true$
    else if ($changepoint_{metric} == false$) && ($changepoint_{state} == false$) then
        return $true$
    else
        return $false$
    end if
end procedure
```

Algorithm 2 is the (simple) decision function for determining whether the application is behaving in a normal fashion. We declare the behavior of the application to be problematic if the absence or presence of a change point in the given system resource metric does not correspond with an absence or presence of a change point in the given application state count respectively. It follows that when the application is not diagnosed as being problematic, then the application is exhibiting normal, problem-free operation.

More formally, our approach to characterizing normal application behavior, based on our hypothesis of change points in application state counts and resource metrics agreeing with each other if and only if the system is problem-free, is that of classifying a point in the time series of application states as being or not being a change point given knowledge of whether the corresponding point in the time series of resource metrics is a change point.

Then, Algorithm 2 is applied to the change point pair for every metric of interest with every application state count of interest, for each time instance for which a diagnosis is desired. The requirement of the computation of left- and right-means for each decision point implies that the algorithm has an intrinsic lag equal to the duration required to collect sufficient samples for the window. Nonetheless, the algorithm can be run online, albeit with a lag, as the space requirements for its state is constant in the order of the duration of the diagnosis run.

6.1.4 Building profiles of application behavior

Finally, training is carried out to identify application state count change points which co-occur with resource metric change points under normal, problem-free operation. Then, diagnosis is carried out using the change point-corroboration framework as described above, on pairs of application state counts and resource metrics that have been found to consistently exhibit change points together during problem-free behavior.

6.2 Experimental Setup and Methodology

We conducted a series of experiments to quantify the behavior of Hadoop in terms of the change points of various operating system-reported resource metrics and Hadoop application state counts. The aim of these experiments was to identify metrics and application states of interest and significance, so as to characetrize the (problem-free) behavior of Hadoop under various workloads. This will facilitate the devising of strategies for maximizing the efficacy of the *BlackSheep* problem-diagnosis approach as applied to Hadoop. A total of approximately 30 to 40 problem-free experiment runs of each workload type examined were run, and all traces were carefully visually inspected to ensure that for each workload, only traces that were similar to other traces for the same workload were considered. Then, traces from representative runs were used for our analysis.

6.2.1 Setup and data collection

We deployed a 6-node (5 slave, 1 master) Hadoop 0.12.3 cluster on six 850 MHz Pentium III nodes on the Emulab remote testbed [20], each with 512 MB of main memory, and running Linux 2.6.20. A seventh 850 MHz Pentium III node was used to generate workloads for the Hadoop cluster.

Operating system-reported resource metrics were collected from the `proc` filesystem using the `sysstat-8.0.4` system monitoring package [30]. Metrics from the following categories were collected using the `sysstat` package: aggregate CPU utilization and process activity, aggregate disk I/O activity, paging and virtual memory subsystem activity, per-disk I/O activity, per-network device activity, and per-process CPU and memory utilization. These metrics were collected at one second intervals, and the additional overhead imposed on the system was not found to be significant. However, further work is required to quantify the overheads of this instrumentation.

Hadoop application state counts were collected by parsing Hadoop activity logs from the Hadoop DataNodes and TaskTrackers using the Hadoop log parser as described in Section 4; this parsing was performed using the offline mode of the log parser to generate time series traces of application event and state counts. Parsing was generally not time-consuming, and parsing an average log file from one node generated from an active workload lasting one hour took less than 2 seconds. Despite this, further work is again needed to quantify the time costs of using the log parser.

6.2.2 Candidate workloads

Candidate workloads for Hadoop were picked from the example Hadoop Map/Reduce applications as provided with the Hadoop distribution, as well as the Nutch distributed web crawler [25] [28]. The `randomwriter` and `sort` example applications were picked as candidate workloads as they are commonly suggested as benchmarks for Hadoop clusters [26], while the Nutch web crawler was picked as it represents a significant real-world application commonly used with Hadoop.

The objective of the choice of candidate workloads is to empirically observe the behavioral characteristics of a gamut of possible Hadoop behavioral profiles when running various types of application workloads, so that the normal behavior of Hadoop as exhibited with these workloads will generalize to arbitrary Hadoop workloads. Application workloads can be classified as being combinations of compute-intensive, disk-read and disk-write intensive, and network-intensive, while real application workloads will generally be composed of some combination of these characteristics.

The `randomwriter` example application writes a given configurable number of structured records comprised of random bytes to disk on each Hadoop node, and represents a disk-write intensive workload with minimal disk-reads and minimal computation. The `sort` example application sorts a given file of structured records by key, and represents a balanced mixed workload with disk-reads, disk-writes, and network transactions to merge sorted records. The Nutch web crawler represents a real-world workload, and also represents a network-intensive workload (relative to disk and compute activity). In our experiments, the `randomwriter` was typically configured to write 2 GB of data to each node, the `sort` was typically set up to sort 2 GB of data per node (or a 10 GB dataset), and Nutch was used to crawl a locally mirrored static website with approximately 2000 pages, served from an independent node local to the experiment cluster but not running any Hadoop instance. Hence, we believe that our choice of workloads feasibly provides adequate coverage of the possible Hadoop workloads.

6.2.3 Change points applied: Characterizing normal application behavior

Next, the time series' of each of the resource metrics and application state counts were assembled into a single trace for each run of an experiment. Visualization tools were then applied to each trace to generate plots of the time series' for resource metrics and application state counts, and the change point algorithm was applied to the time series' for resource metrics and application state counts. Finally, the behaviors of pairs of the change points of resource metrics and application state counts was manually examined to identify consistent patterns during normal application behavior that can be used as behavioral indicators of normal application behavior. These signatures of normal application behavior can then be applied to problem diagnosis by identifying behaviors that deviate from this prior-knowledge of problem-free behavior.

The generation of change points for application state counts and resource metrics is as follows. First, values for the tunable parameters of Algorithm 1 were chosen: these were $window$, the size of the observation window measured in the number of samples to the left and right of the point under consideration in the time series, and $thresh$, the proportion of left mean μ_L, that the difference of means, $\Delta\mu$, must exceed for a change point to be flagged. The tuning parameters were chosen to meet two objectives: (i) to generate application state count change points that corresponded to high-level expectations based on our *a priori* understanding of the behavior of Hadoop (for instance, the number of map tasks changing in the TaskTracker should generate a change point in our algorithm), and (ii) to generate change points in as many resource metrics as possible that lined up with the application state change points. This uses the implicit assumption that the hypothesis of the *BlackSheep* approach, that resource metrics and application state counts should behave similarly, is correct—doing so is theoretically sound from a machine learning point of view, as the assumption is akin to a Bayesian assumption of priors, with the exception that the parameters are being tuned by hand for (at least this initial pass of) this work.

Next, change points were generated for each application state count and each resource metric, based on tuning parameter values that were chosen separately for application state counts, and for resource metrics. The intermediate results and eventual chosen parameter values of the tuning process are reported in Section 6.3.1.

Finally, the corroboration between application state counts and resource metrics was verified using the algorithm presented next, with the addition that the logarithms ($x' = \log(x + 1)$) of all measured metrics was used to perform analysis on the change points rather than the absolute values of observed values.

6.2.4 Evaluation of corroboration between application state counts and resource metrics

Next, we describe the methodology and algorithm for evaluating the paired behavior of change points in application state counts and resource metrics. Algorithm 2 is the (simple) decision function for determining if the presence (or absence) of a change point in the resource metric correctly predicted a presence (or absence) of a change point in the application state, in which case a true positive or true negative was recorded respectively. Evaluation scores of the accuracy of predictions were assigned as follows: non-negative values were assigned to points in time for which true positives or true negatives of application state change points were predicted by resource metric change points, and negative values were assigned to points in time for which false positives or negatives were observed. The rationale for this choice of scores is that the evaluation score is heavily biased against misdiagnoses, so that misdiagnosing a change point (false positives/negatives) will impact the score negatively much more than correctly diagnosing a change point will positively impact it.

Algorithm 3 Decision function for deciding if two orthogonal views of the system agree with each other

```
procedure STATEMETRICCORROBORATEEVAL($changepoint_{metric}, changepoint_{state}$)  ▷ $changepoint$'s are boolean flags indicating if a change point occurred in the time series of the resource metric or application state count respectively
    if ($changepoint_{metric} == true$) && ($changepoint_{state} == true$) then    ▷ True positive
        return $1$
    else if ($changepoint_{metric} == false$) && ($changepoint_{state} == false$) then    ▷ True negative
        return $0$
    else if ($changepoint_{metric} == true$) && ($changepoint_{state} == false$) then    ▷ False positive
        return $-2$
    else if ($changepoint_{metric} == false$) && ($changepoint_{state} == true$) then    ▷ False negative
        return $-1$
    end if
end procedure
```

The notion of true/false positives/negatives is defined relative to resource metric change points being used to predict application state change points. Change points in the application state are arbitrarily chosen, without loss of generality, to be the unobserved ground truth of the application's behavior, and the change points in resource metrics are then used to classify or predict if there is a change point in the application state. The observed application state change points are then used to test if the prediction made by the (absence or presence of a) change point in the resource metric of the application state change point is correct. It should be noted that due to our definition of normal behavior as having both application state and resource metric change points agree with each other, this choice of ground truth is without loss of generality and can be reversed.

6.3 Results and Analysis

Our analysis provides initial evidence supporting our hypothesis that application state counts and resource metrics will agree with each other under problem-free, normal execution. Our results suggest further directions for developing this hypothesis into a full-fledged approach for problem diagnosis.

In general, some application state count-resource metric pairs have exhibited visually corroborating behavior, in which change points in the count of the particular application state occurred only together with change points in the particular resource metric, after allowing for minor edge effects due to possible lags in either of the variables of interest.

In addition, there is evidence that workload types, as defined in Section 6.2.2, can be identified using change point corroboration. Particular application state count-resource metric pairs exhibited corroborating behavior only under particular workloads, and not others, suggesting that the absence or presence of corroboration between particular application state count-resource metric pairs can serve as identifying signatures for workload types.

6.3.1 Parameter tuning

We review the effects of different values for the two tunable parameters for Algorithm 1, the window size measured in number of samples, $window$, and the threshold $thresh$ as a proportion of the left mean, μ_L, on the change points generated for each of application state counts and resource metrics. We considered the change points in the aggregate on state counts for each of the DataNode and TaskTracker (i.e. a change point is said to be observed in the DataNode (or TaskTracker) at time t if a change point is observed in any of the application states for the DataNode (or TaskTracker) at time t, or more formally, $\forall x \in \{states_{datanode/tasktracker}\}, changepoint_{datanode/tasktracker} = max\{x\}$) for tuning purposes. In order to maximize the amenability of the change points generated for problem diagnosis, we aimed to generate change points such that the separation between groups of consecutive change points was maximized. This was to maximize the degree to which we could visually identify corroborating change points between application state counts and resource metrics.

First, we held the value of $thresh$ constant, at $thresh_{state} = 20.0$ and $thresh_{resource} = 1.0$, and varied $window$ for aggregate application state counts for both aggregated DataNode state counts and TaskTracker state counts.

We found that the optimal values of $window$ that resulted in the greatest separation between groups of change points differed for DataNode state counts and resource metrics. Consider Figure 7 and Figure 8: the change points for the resource metric are relatively well-spaced in Figure 7, where the $window = 5$, while the change points for the DataNode state counts are relatively poorly spaced and do not appear to mark out distinguishable logical phases of execution; on the other hand, in Figure 8, with $window = 45$, the change points for the resource metric are less well-spaced and less coherent than in Figure 7, but the change points for the application state counts in Figure 8 have significantly greater separation between groups of consecutive change points. Hence, we propose as a heuristic that the value of $window$ for change point generation for resource metrics be approximately one order of magnitude smaller than that for DataNode state counts, with $window_{resourcemetric} \approx 5$, and $window_{applicationstatecounts} \approx 45$.

However, we found that the optimal value of $window$ for TaskTracker state counts and resource metrics that resulted in the greatest separation between groups of change points was similar—for both TaskTracker state counts and resource metrics, smaller values of $window$ gave rise to greater separation between groups of change points. We propose, as a heuristic, that a value of $window = 5$ be used for computing change points in TaskTracker state counts (while the heuristic for resource metrics from previously holds). This is as shown by Figures 9 and 10, where the change points for $window = 5$ are markedly more separated than for the change points for $window = 45$.

The intuition behind the difference in the optimal window sizes for DataNode state counts and for TaskTracker state counts and resource metrics is that DataNode state counts are experiencing changes in a different time-scale than TaskTracker state counts and resource metrics. In the steady state during periods of workload, the DataNode serviced many requests relative to the TaskTracker as each map or reduce task handled by the TaskTracker involved multiple data blocks. Thus, DataNode states tended to exhibit some intrinsic steady-state fluctuation that was normal and expected of its problem-free behavior, while TaskTracker states were relatively longer lived as compared to DataNode states, and resource metrics experienced less fluctuation/fewer change points than DataNode state counts for the same tuning parameters. Thus, different tuning parameters can be used for the DataNode state counts, as the changes in the DataNode state counts can be argued to be part of the steady state of its behavior.

Next, we studied the effect of varying $thresh$, holding $window_{state} = window_{resource} = 20.0$ constant.

We found that higher threshold values resulted in excessively many change points being omitted from the time series of resource metrics, resulting in a sparse series of change points generated that fails to correspond with the series of change points generated from application state counts. For a value of $thresh = 4.0$, as shown in the top panel of Figure 11, the series of change points generated from the time series of the resource metric is sparse relative to the series of change points from the application state counts in the bottom panel, while for a value of $thresh = 1.0$, as shown in the top panel of Figure 12, the series of change points is less sparse than in Figure 11, but the change points remain well-spaced with significant and clear separation between groups of successive change points. We believe that the value of $thresh$ for resource metrics can be further lowered, but we have nonetheless demonstrated that lower values of $thresh$ are more effective for use with generating change points for resource metrics.

In addition, we found that lower threshold values resulted in excessively many change points being generated for DataNode state counts, reducing the number of consecutive change points, resulting in less smooth state count change point plots, while lower threshold values resulted in excessively few change points being generated for TaskTracker state counts, increasing the number of consecutive change points, resulting in smoother state count change point plots. This is as demonstrated from the bottom panel plots of Figures 11, 12, 13 and 14 respectively. The change

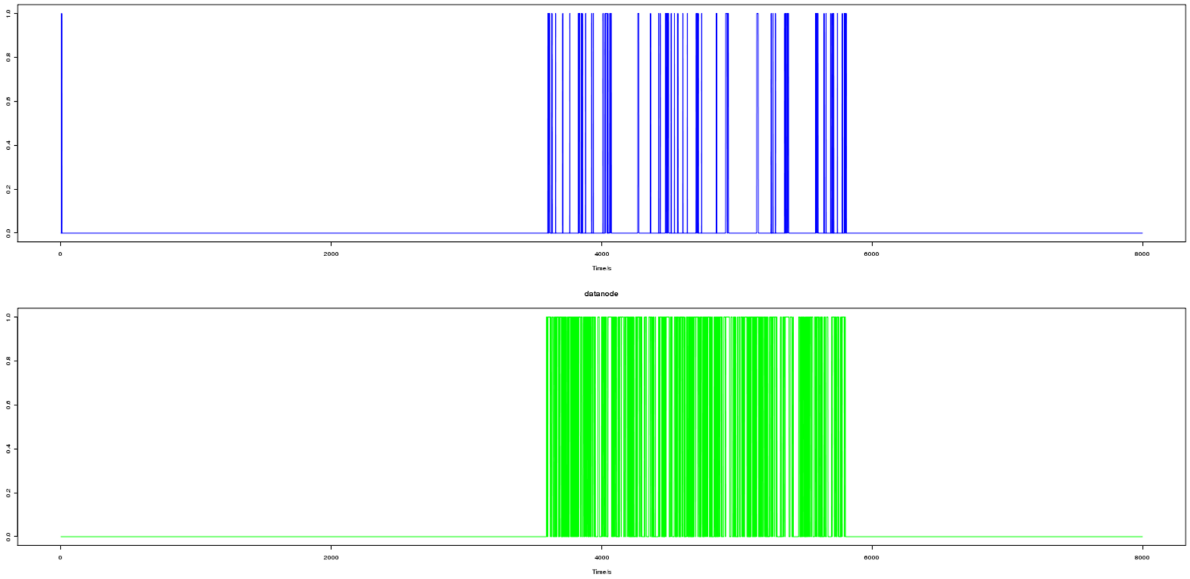

Figure 7: Plot of change points (binary indicators) of resource metric (CPU utilization, user%) in top panel, and of change points of application state counts for DataNode in bottom panel, with x-axis measured in seconds for both plots; $window = 5$.

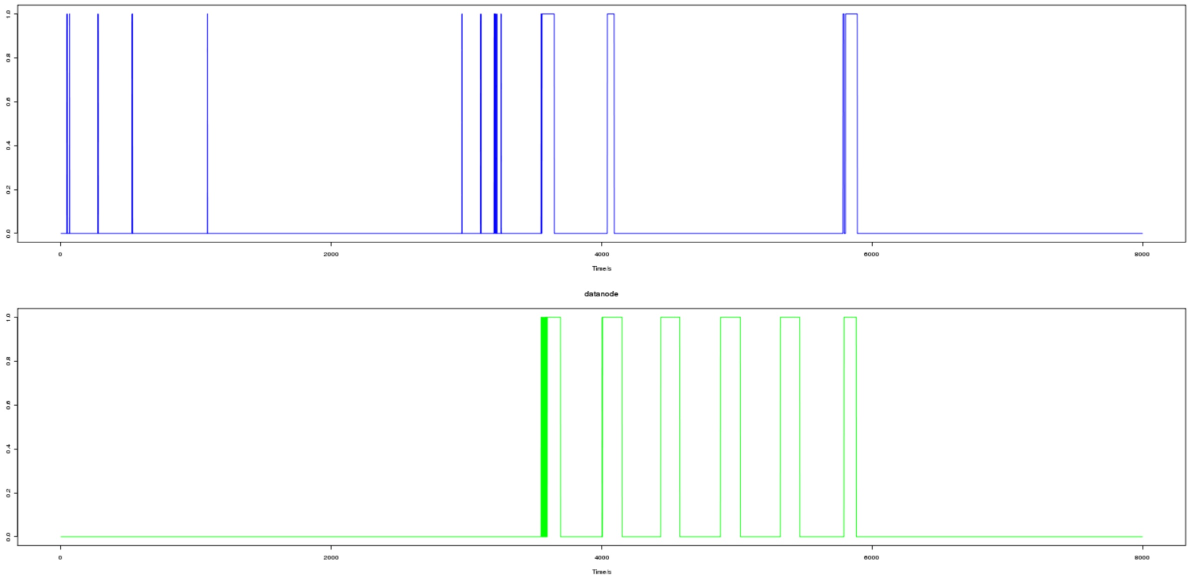

Figure 8: Plot of change points (binary indicators) of resource metric (CPU utilization, user%) in top panel, and of change points of application state counts for DataNode in bottom panel, with x-axis measured in seconds for both plots; $window = 45$.

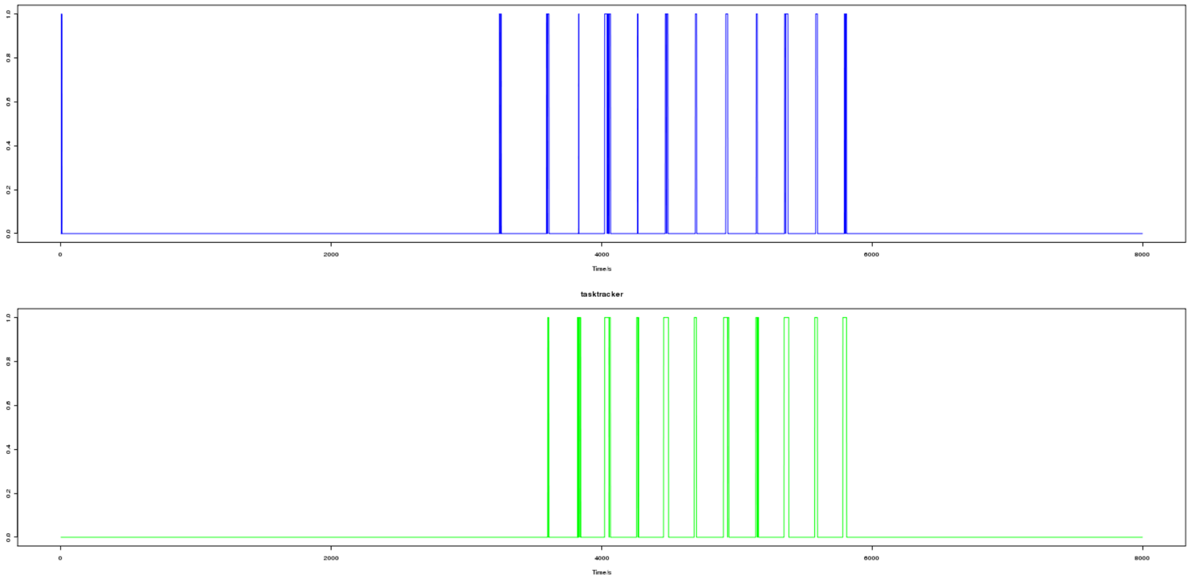

Figure 9: Plot of change points (binary indicators) of resource metric (CPU utilization, user%) in top panel, and of change points of application state counts for TaskTracker in bottom panel, with x-axis measured in seconds for both plots; $window = 5$.

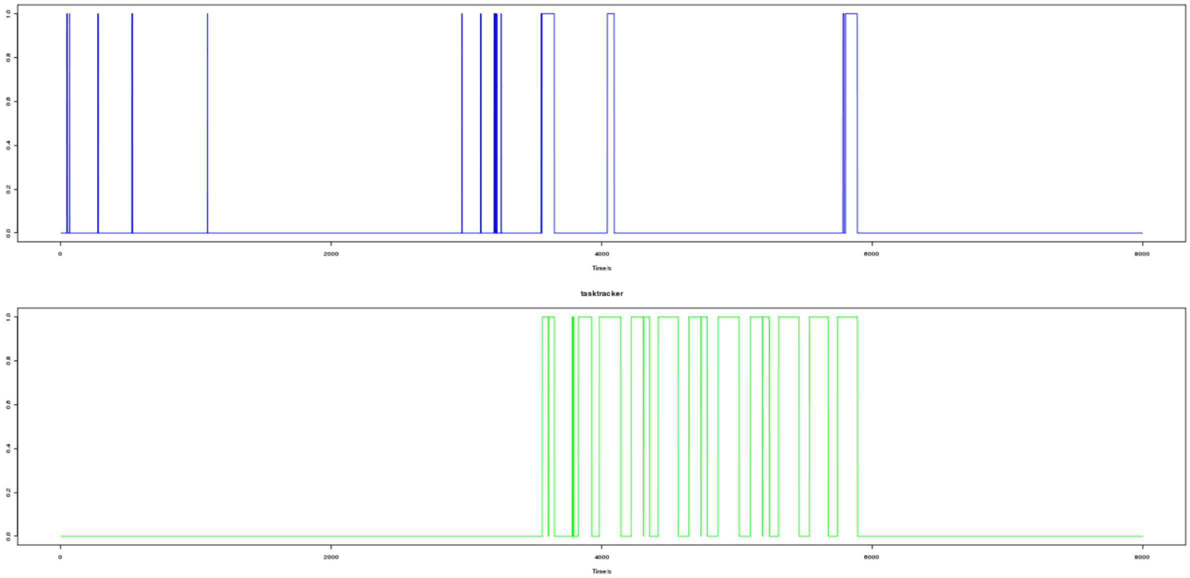

Figure 10: Plot of change points (binary indicators) of resource metric (CPU utilization, user%) in top panel, and of change points of application state counts for TaskTracker in bottom panel, with x-axis measured in seconds for both plots; $window = 45$.

point plots are smoother and the successive change points have greater inter-change point separation for DataNode state counts in Figure 12, with the larger $thresh_{state} = 4.0$ than in Figure 11, with the smaller $thresh_{state} = 1.0$, and so are more amenable to statistical analysis in general. The change point plots for TaskTracker state counts, on the other hand, are smoother in Figure 14, with $thresh_{state} = 1.0$, than in Figure 13, with $thresh_{state} = 4.0$.

The general intuition for the difference between the optimal value of $thresh$ in these cases is that DataNode states, such as ReadBlock and WriteBlock, are short-lived (for the configured block sizes) relative to TaskTracker states, such as Maps and Reduces. Hence, with shorter-lived states, the DataNode has greater steady-state fluctuations than the TaskTracker, so that for high-level meaning to be extracted from the DataNode state counts, greater threshold values must be used to filter out fluctuations that are intrinsic to its steady-state behavior to characterize bulk behavior, which is more useful for problem diagnosis in general. Also, the intuition for the optimal $thresh$ value for resource metrics is that the heuristic that is guiding our choice in this case is our prior assumption of how change points in resource metrics should corroborate with the change points in application state counts.

Thus, we have demonstrated a few optimal tuning parameter values for Algorithm 1, and have in so doing demonstrated that our hypothesis has thus far been supported by the successful tuning of the parameters. The general intuition obtained from the tuning process suggests that the parameters used by the algorithm are sensitive to the configuration of the application, and has illuminated some aspects of the behavior of the application.

This is a double-edged sword, as it demonstrates that analyzing the change points of application states and resource metrics together can aid in understanding the application, but also that tuning the algorithm can be a challenge for users without significant prerequisite knowledge about the application to be profiled/to have problem diagnosis carried out. However, this is also an opportunity, as Bayesian hyper-parameter learning can be applied to the problem of learning the optimal values of the tuning parameters, and the learning process itself can provide positive feedback to the diagnostic process, as will be described in Section 8.3.2.

6.3.2 Distinguishing workloads

From the trace data collected, we found that particular pairs of the counts of particular application states and the metrics of particular resources displayed consistent behavior within each workload, but varied across workloads. One such case in point is the change point series of the counts of the ReduceTask state for the TaskTracker, and the change point series of the user-space percentage CPU utilization (`user%`) resource metric.

From traces shown in Figures 15 and 16 for the `randomwriter` and `sort` workloads respectively, it can be seen that there is a strong co-occurrence of the change points of the counts of the ReduceTask state and the change points of the `user%` metric, such that the presence (and absence) of a change point in the state count or metric serves as a good predictor of the presence (and absence) of the metric or state count respectively.

On the other hand, from the trace shown in Figure 17, for the Nutch web crawler workload, it can be seen that the co-occurrence of the change points of the counts of the ReduceTask state and the change points of the `user%` metric is much weaker than in the previous two workloads, although all three traces were drawn from problem-free runs. This suggests that the patterns of the strength of co-occurrence of change points in application state counts and metrics can be used as a signature for workloads to infer the type of workload being executed on a given node.

An explanation for the difference in change point behavior between the `randomwriter` and `sort` workloads and the Nutch web crawler workload is that the former two workloads contain periods of disk-I/O-bound activity, when large amounts of blocks are being written to disk, while the web crawler workload does not have such a phase of execution. This suggests that a strong co-occurrence of change points for the ReduceTask state and the `user%` metric can be an indicator for disk-I/O-bound activity.

Again, the higher level implication of this observation is that patterns can be found in co-occurrences of behaviors in application state counts and resource metrics to learn signatures of workload types for anomaly detection.

6.3.3 Change point corroboration with resource metrics

Next, we present, in Figure 18 a visualization of the operation of our change point corroboration algorithm, Algorithm 3, using a single resource metric and a single application state count, when applied to every point in the time series of change points generated from a resource metric and an application state count. The values of the evaluation score defined in Section 6.2.4 are plotted against time alongside the time series of change points for the resource metric and counts of the chosen application state. This illustrates how we evaluate the efficacy of the corroboration of check points for a single experimental run.

This example trace highlights one source of false positives and negatives in the corroboration of change points-—edge effects, due to minor lags in the response of application state counts to resource metrics, or vice versa. Even in the case that application state count change points appear to visually line up with resource metric change points, false positives and negatives still occur in the immediate vicinity of the change points in the time series, resulting in

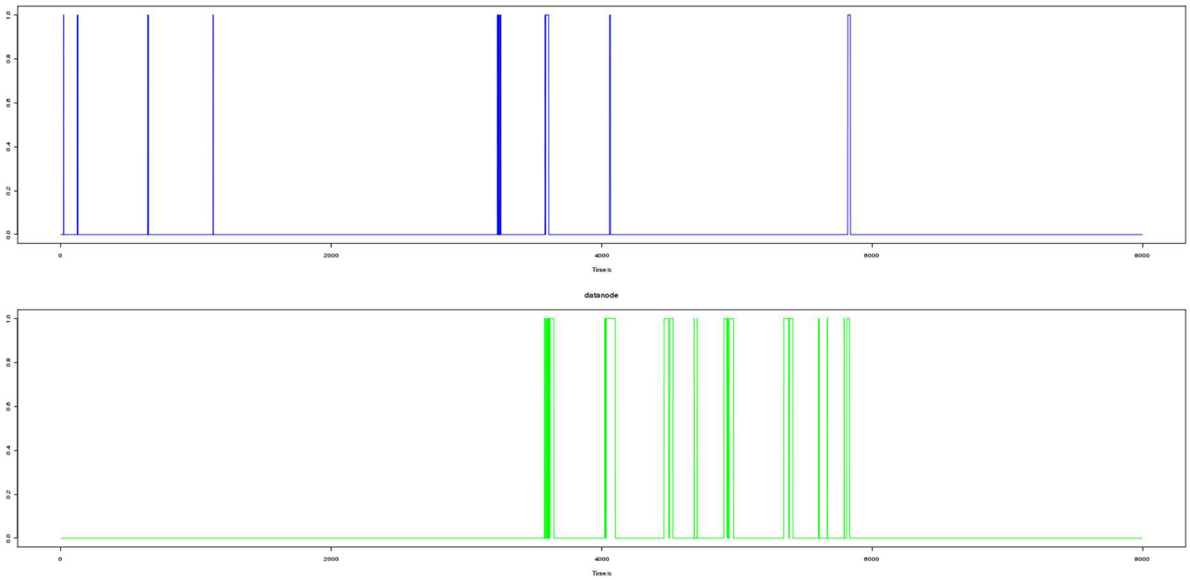

Figure 11: Plot of change points (binary indicators) of resource metric (CPU utilization, user%) in top panel, and of change points of application state counts for DataNode in bottom panel, with x-axis measured in seconds for both plots; $thresh_{metric} = 4.0, thresh_{state} = 4.0$.

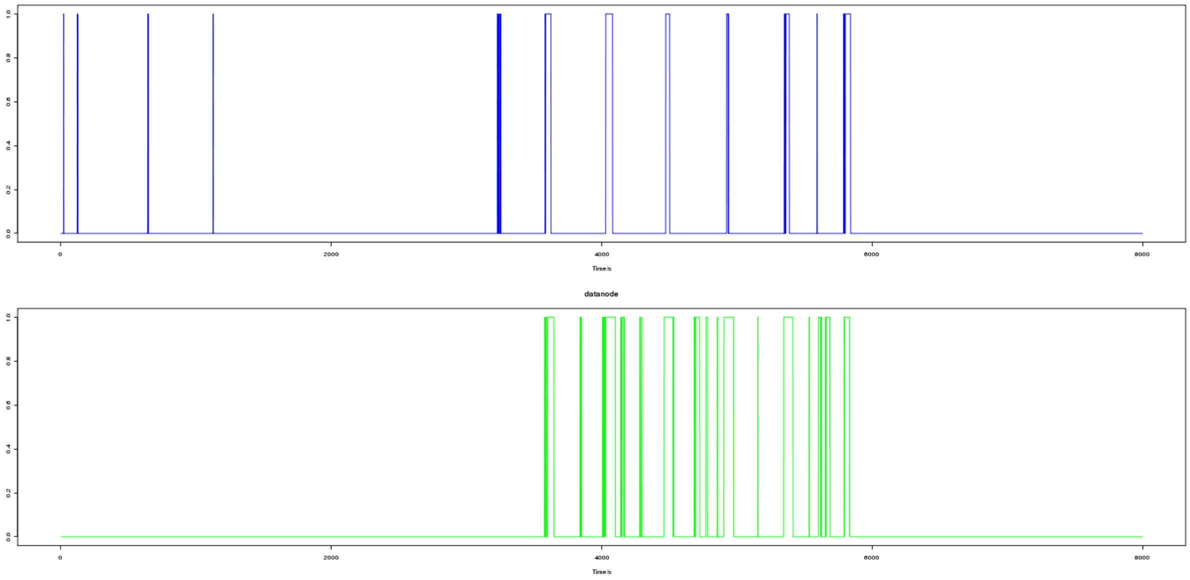

Figure 12: Plot of change points (binary indicators) of resource metric (CPU utilization, user%) in top panel, and of change points of application state counts for DataNode in bottom panel, with x-axis measured in seconds for both plots; $thresh_{metric} = 1.0, thresh_{state} = 1.0$.

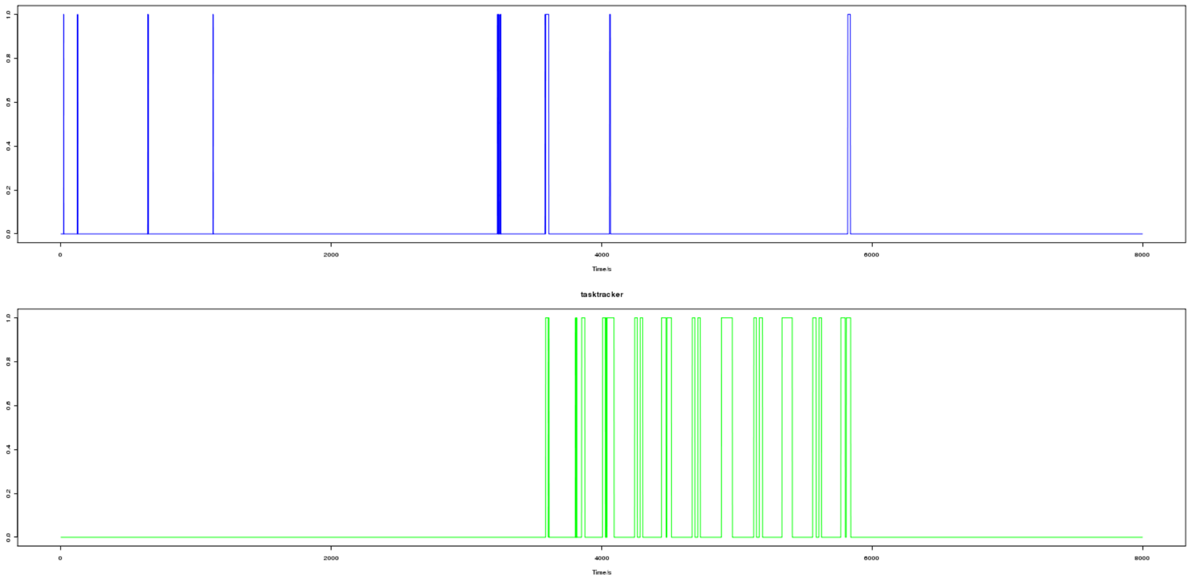

Figure 13: Plot of change points (binary indicators) of resource metric (CPU utilization, user%) in top panel, and of change points of application state counts for TaskTracker in bottom panel, with x-axis measured in seconds for both plots; $thresh_{metric} = 4.0, thresh_{state} = 4.0$.

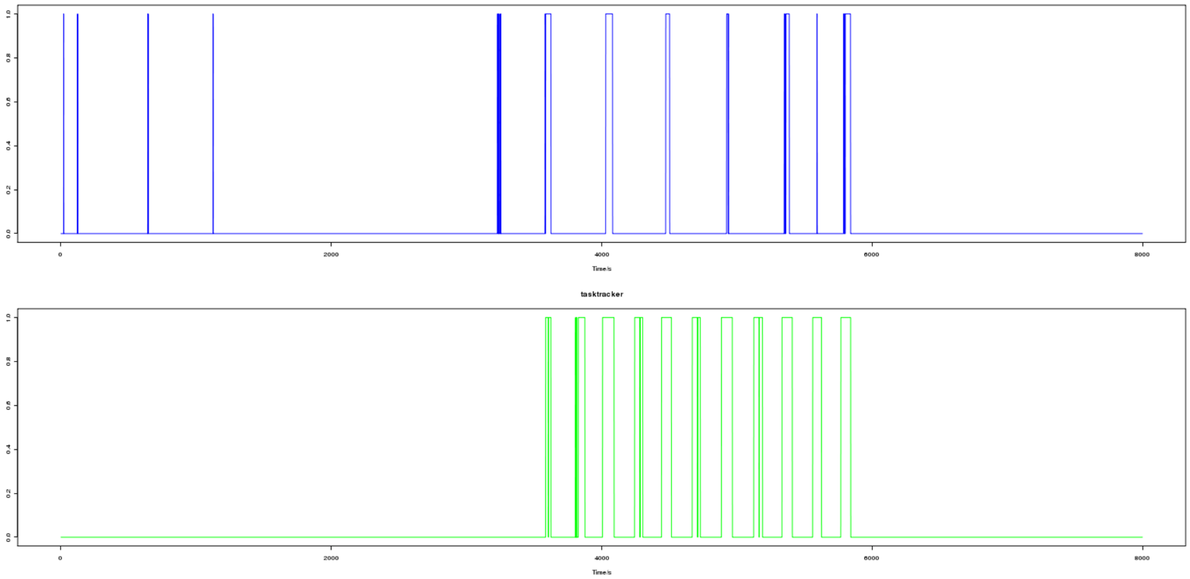

Figure 14: Plot of change points (binary indicators) of resource metric (CPU utilization, user%) in top panel, and of change points of application state counts for TaskTracker in bottom panel, with x-axis measured in seconds for both plots; $thresh_{metric} = 1.0, thresh_{state} = 1.0$.

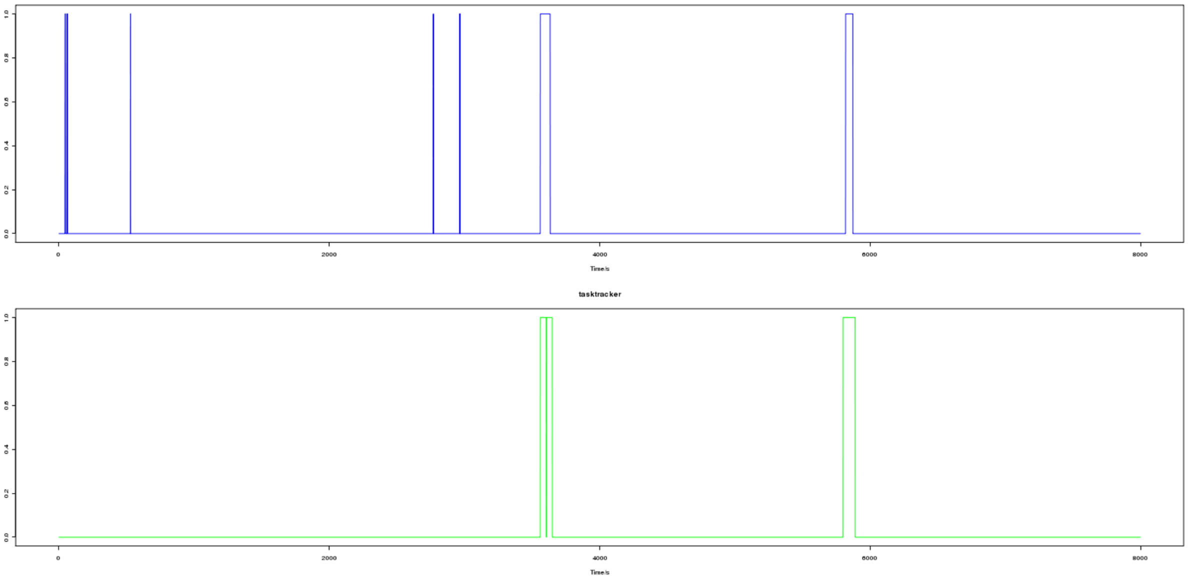

Figure 15: Plot of change points (binary indicators) of resource metric (CPU utilization, user%) in top panel, and of change points of application state counts for TaskTracker in bottom panel, with x-axis measured in seconds for both plots; Trace of a single run of a `randomwriter` workload.

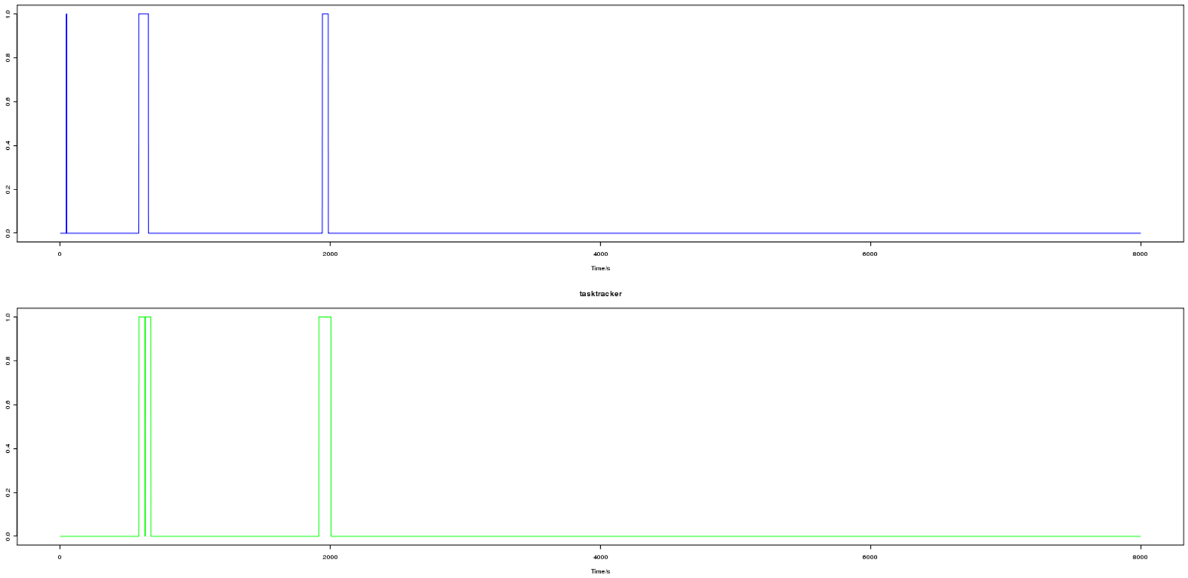

Figure 16: Plot of change points (binary indicators) of resource metric (CPU utilization, user%) in top panel, and of change points of application state counts for TaskTracker in bottom panel, with x-axis measured in seconds for both plots; Trace of a single run of a `sort` workload.

false positives and negatives that are spurious and not truly indicative of prediction error. This issue is addressed, and a solution is proposed, in Section 8.3.1. Apart from edge effects, it appears that false positives and negatives also result from tuning parameter values that result in different high-level sensitivities of the change points produced, with change points being generated for changes that have different orders, that parameter tuning will be highly critical to any success of this approach.

6.3.4 Change point corroboration: Evaluation

Finally, we present aggregate statistics of the evaluation scores for the corroboration between the change points of every pair of application state and resource metrics for a single representative node in a single representative experimental run for each workload type, as an illustration of the general performance of our corroboration technique in its current form. The histograms presents the frequency of mean evaluation scores for application state-resource metric pairs.

From Figures 19, 20, and 21, it can be seen that the evaluation scores for all application state-resource metric pairs are negative, indicating that the false positive/negative rate for the change point corroboration is currently not sufficiently effective for the corroboration of change points to be used as a problem diagnosis algorithm.

Nonetheless, the modal mean evaluation score is approximately < -0.1, with a strong distribution of mean evaluation scores around this range. This implies that there are slightly more than two false negatives or one false positive on average for every true positive (see Section 6.2.4 for a detailed definition of true/false positives/negatives in this context). Given that the mean evaluation score is almost 0 and only slightly negative, this suggests that there is potential for the algorithm to be refined to produce better classification results than a random classifier.

Finally, the histograms of mean evaluation scores for all three workloads are similar in shape, suggesting that our approach is possibly agnostic to different workload types.

7 Related Work

7.1 Problem-Diagnosis Techniques

There are many existing techniques to perform problem diagnosis in distributed systems. The *RAMS* approach proposed here differs from those of Cohen et al [11] and Pinpoint [5] in a few ways. First, we do not employ any learning or training prior to problem diagnosis. Our approach is based on a hypothesized *a priori* model of problem-free system behavior, and we use statistical methods to establish whether this hypothesized behavior is being followed. Our technique has no learning overhead but is constrained by the degree to which our hypothesized model of system behavior is applicable to other types of systems.

Second, both Pinpoint and Cohen's ”ensembles” utilize a system-wide, global approach that examines metrics on every node in the distributed system. This may cause scalability issues in terms of computation and communication overhead in large systems (although [11] presents a scalable approach). Both the *RAMS* and *BlackSheep* approaches addresses scalability by making the rather strong assumption that information local to a node alone is sufficient for problem diagnosis, thereby saving the network bandwidth needed to transmit metrics to a central location for analysis, and limiting the analysis to the size of the performance-data set of one node.

Our approach is also a black-box technique (although we do make use of white-box information, obtained via black-box techniques, in the form of application logs), and *RAMS* aims for problem diagnosis without using any application-level knowledge, in constrast with Pip's white-box approach [16]. Finally, the ”odd-man-out” peer-comparison heuristic [15] proposed by Pertet et al might apply to the target distributed system used in our paper, as the slave nodes in a Map/Reduce cluster could conceivably be running very similar workloads and therefore, might lend themselves to the peer comparison of performance data for problem diagnosis.

7.2 Vertical Profiling

The idea of correlating system behavior across multiple layers of a system is not new. Hauswirth et al's ”vertical profiling” [9] aims to understand the behavior of object-oriented applications by correlating metrics collected at various abstraction levels in the system. Vertical profiling was used to diagnose performance problems in applications in a debugging context at development time, requiring access to source code while our approach diagnoses performance problems in production systems without using application knowledge.

8 Future Work

8.1 Sliding windows for *RAMS*

Currently, the use of an ordinary least squares linear regression to compute a test statistic as a criteria for diagnosis requires the use of large windows of samples (at least 30 to 50 samples) for the linear regression to produce statistically sound (unbiased estimators with reasonably good fits) estimates of the various parameters. Hence, the next area of improvement for *RAMS* is to use more direct measures of correlation other than autocorrelation between lagged residuals in an ordinary least squares linear regression. This will reduce the computational cost of computing the test statistic needed for determining if a node is a problem node. Also, the use of a sliding window, in conjunction with more direct measures of correlation, will hopefully reduce the number of samples needed for a statistically sound test statistic to be computed.

8.2 Experimental Setup for *BlackSheep*

The two main areas of improvement for the experimental procedure for *BlackSheep* are in (i) the controlled measuring of the overheads of instrumentation, as measured in system resource usage and impact on system performance, and (ii) the varying of workloads for Hadoop to increase the generality of the experiments ran, to create workloads with a variable mix of modes of operation (disk-, compute-, memory-, or network-intensive, for instance), and to identify any characteristics of Hadoop that exhibit a stable relationship with workloads that vary along the dimensions we have defined.

8.3 Change Point Corroboration

Various improvements and enhancements can be made to the overall change point corroboration algorithm to improve its accuracy in correctly predicting application state behavior (in terms of change points) using the behavior of resource metrics, to create a viable problem diagnosis approach.

8.3.1 Accounting for edge-effects in change point corroboration

To account for edge-effects and possible lags in application behavior, we intend to implement a low-pass Gaussian filter over a tunable window size of change points observed before and after the given instance in time of observation-–for a given change point detected in the system resource metric at time t, if a change point is observed in the application state count in a time within the given window $t' \in [t-w, t+w]$, then, the algorithm diagnoses the application as being problem-free with a probability that has a Gaussian fall-off, so that the further from t that the application state count change point is observed, the lower the probability that the application is truly problem free. Conversely, if a change point is not observed in the system resource metric at time t, but a change point is observed in the application state count within the given window at $t' \in [t-w, t+w]$, then the algorithm diagnoses the application as having a problem with probability that has a Gaussian fall-off, so that the greater the difference $|t-t'|$, the lower the probability that the application has a problem.

8.3.2 Dealing with magic numbers: Bayesian hyper-parameter learning

The *BlackSheep* approach currently uses two tunable parameters: a window size and a threshold. However, initial results have proved that optimal values for these parameters can be highly sensitive to the particular variables in question that they are applied to, specifically, resource metrics and application state counts. Hence, an approach to these magic numbers, or optimal values for tunable parameters, would be to introduce an additional layer of Bayesian hyper-parameter learning to learn values for these tunable parameters that will optimize the classification problem of change point identification in application states.

8.3.3 Learning workload identities

Finally, an extension of the *BlackSheep* change point corroboration technique would be to use the same change point corroboration ideas to attempt to learn identities of workloads, and to find out if the parameters that identify these workloads can be composed in an intelligible manner to create signatures of arbitrary workloads as defined using change point corroborations between application state counts and resource metrics.

8.4 Application logs

Finally, yet another extension to the work presented here with the Hadoop log parser would be to identify characteristics of applications and their logs in general that would render them amenable to similar treatment of extracting events, and more importantly, inferring states of executions of the applications.

9 Conclusion

In conclusion, we have presented: (i) what we believe to be a novel use of application logs to extract application events, and to use these events to infer high-level, semantically-rich states of execution of the application; (ii) *RAMS*, a new, scalable black-box approach to problem diagnosis using extremely low-level metrics, hardware performance counters, in conjunction with an *a priori* statistical model of the behavior of nodes in a distributed system, to perform node-local problem determination in a distributed system, and (iii) *BlackSheep*, a black-box technique for characterizing application software behavior by synthesizing application behavior, as reported through application logs using our newly presented log parsing technique and library, together with collections of operating system-reported resource metrics, with the eventual objective of performing problem diagnosis by detecting anomalies from normal application behavior. Not only have we described the principles behind the algorithm and the architecture of our log parser for inferring state, we have also presented rudimentary results demonstrating the efficacy of *RAMS* at problem determination. Lastly, we have shown an approach to synthesizing information from application logs with operating system metrics.

Acknowledgements

I would like to thank Priya Narasimhan for her advice, constant encouragement and inspiring thoughts, and Xinghao Pan for enduring many animated descriptions of this work, without which this project would not have been possible. This work is also dedicated to my parents and sister, for their unwavering, whole-hearted support for me to pursue my dreams.

References

[1] M.K. Agarwal, M. Gupta, V. Mann, N. Sachindran, N. Anerousis, L. Mummert. *Problem Determination in Enterprise Middleware Systems using Change Point Correlation of Time Series Data* Proc. 10th IEEE/IFIP Network Operations and Management Symposium, Vancouver, BC, 2006.

[2] P. Barham, A. Donnelly, R. Isaacs, R. Mortier. *Using Magpie for request extraction and workload modelling.* Proc. 6th Symposium on Operating Systems Design & Implementation, San Francisco, CA, 2004.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield. *Xen and the Art of Virtualization,* Proc. 2003 Symposium on Operating Systems Principles, Albany, NY.

[4] M. Basseville, I. V. Nikiforov. ”Change Detection Algorithms” in *Detection of Abrupt Changes: Theory and Application,* 1st ed. New Jersey, USA: Prentice Hall, 1983, pp. 23-62.

[5] M. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer. *Pinpoint: Problem Determination in Large, Dynamic Internet Services.* Proc. International Conference on Dependable Systems and Networks, Bethesda, MD, 2002.

[6] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, A. Fox. *Capturing, Indexing, Clustering, and Retrieving System History.* Proc. 2003 Symposium on Operating Systems Principles, New York, NY.

[7] J. Dean, S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters.* Proc. 6th Symposium on Operating Systems Design & Implementation, San Francisco, CA, 2004.

[8] S. Ghemawat, H. Gobioff, and S. Leung. *The Google File System.* 19th Proc. 19th Symposium on Operating Systems Principles, Lake George, NY, October, 2003.

[9] M. Hauswirth, A. Diwan, P. Sweeney, M. Hind. *Vertical Profiling: Understanding the Behavior of Object-Oriented Applications.* presented at the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, BC, Canada, 2004.

[10] J. Hansen. *Trend Analysis and Modeling of Uni/Multi-Processor Event Logs.* Electrical and Computer Engineering Department, Carnegie Mellon University. Pittsburgh, PA. Rep. CMU-ECE-1988-025, 1988.

[11] C. Huang, I. Cohen, J. Symons, T. Abdelzaher. ”Achieving Scalable Automated Diagnosis of Distributed Systems Performance Problems,” Enterprise Systems and Software Laboratory, HP Laboratories Palo Alto. Palo Alto, CA. Rep. HPL-2006-160(R.1), 2007.

[12] T. T Y. Lin, D P Siewiorek. *Error Log Analysis: Statistical Modeling and Heuristic Trend Analysis.* IEEE Transactions on Reliability, Volume 39, Issue 4, Oct 1990, pp. 419-432.

[13] G. S. Maddala, ”Autocorrelation” in *Introduction to Econometrics,* 3rd ed. West Sussex, UK: John Wiley & Sons, 2001, pp. 228-249.

[14] A. Menon, J. Santos, Y. Turner, G. Janakiraman, W. Zwaenepoel. *Diagnosing Performance Overheads in the Xen Virtual Machine Environment.* Proc. First ACM/USENIX International Conference on Virtual Execution Environments, Chicago, IL, 2005.

[15] S. Pertet, R. Gandhi, P. Narasimhan. *Fingerpointing Correlated Failures in Replicated Systems.* presented at the USENIX Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML), Cambridge, MA, April 2007.

[16] P. Reynolds, C. Killian, J. Wiener, J. Mogul, M. Shah, A. Vahdat. *Pip: Detecting the Unexpected in Distributed Systems.* Proc. 3rd Symposium on Networked Systems Design & Implementation, San Jose, CA, 2006.

[17] D. Tang, R. Iyer. *Analysis of the VAX/VMS Error Logs in Multicomputer Environments A Case Study.* Proc., 3rd International Symposium onSoftware Reliability Engineering, Research Triangle Park, NC, 1992.

[18] J. Tucek, S. Lu, C. Huang, S. Xanthos, Y. Zhou. *Triage: diagnosing production run failures at the user's site.* Proc. 21st Symposium on Operating Systems Principles, Stevenson, WA, 2007.

[19] I. Witten, E. Frank, *Data mining : practical machine learning tools and techniques*, 2nd ed. Boston, MA: Morgan Kaufman, 2005, pp. 161-176.

[20] White, B. Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar. *An integrated experimental environment for distributed systems and networks.* Proc. 5th Symposium on Operating Systems Design & Implementation, Boston, MA, 2002.

[21] O. R. Zaane, H. M. Taxin, J. Han . *Discovering Web Access Patterns and Trends by Applying OLAP and Data Mining Technology on Web Logs.* Proc. Advances in Digital Libraries, Pittsburgh, PA, 1998.

[22] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2* , Intel Corporation, Santa Clara, CA, 2008.

[23] Apache Software Foundation issue database. https://issues.apache.org/jira, 2006.

[24] Apache Logging Services Project. http://logging.apache.org/log4j/, 2007.

[25] Hadoop. http://lucene.apache.org/hadoop, 2007.

[26] core-user@hadoop.apache.org Archives. http://mail-archives.apache.org/mod_mbox/hadoop-core-user/, 2006.

[27] Java Virtual Machine Tool Interface. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/index.html, 2004.

[28] Nutch. http://lucene.apache.org/nutch, 2004.

[29] oprofile. http://oprofile.sourceforge.net, 2003.

[30] SYSSTAT. http://pagesperso-orange.fr/sebastien.godard/, 2002.

APPENDIX

A Hadoop Application States

The list of DataNode and TaskTracker events and states that our Hadoop log parser extracts from the DataNode and TaskTracker logs respectively are as listed. The $Idle$ state is a special state which is never reported, but is included for completeness' sake. The TaskTracker and DataNode are each implied to be in the $Idle$ state by an absence of counts of all other states.

The $Error$ state can either be an instant or persistent state–instant $Error$ states are ones reported on encountering error messages in the log, while persistent $Error$ states are reported when any of the other persistent states are reported to have been terminated due to an error.

A.1 TaskTracker Events and States

States / Events	StartStartEvent?	StateStopEvent?	InstantStateEvent?
Idle	N	N	N
Error	Y	Y	Y
ReduceTask	Y	Y	N
ReduceCopyTask	Y	Y	N
ReduceCopyTask_Local	Y	Y	N
ReduceCopyTask_Remote	Y	Y	N
ReduceSortTask	N	N	Y
ReduceMergeCopy	Y	Y	N
ReduceReduceTask	N	N	Y
MapTask	Y	Y	N
CleanUp	N	N	Y

A.2 DataNode Events and States

States / Events	StartStartEvent?	StateStopEvent?	InstantStateEvent?
Idle	N	N	N
DeleteBlock	N	N	Y
ReadBlockRemote	Y	Y	N
WriteBlockLocal	Y	Y	N
WriteBlockRemote	Y	Y	N
WriteBlockLocal_Replicated	Y	Y	N
WriteBlockRemote_Replicated	Y	Y	N
Error	Y	Y	Y

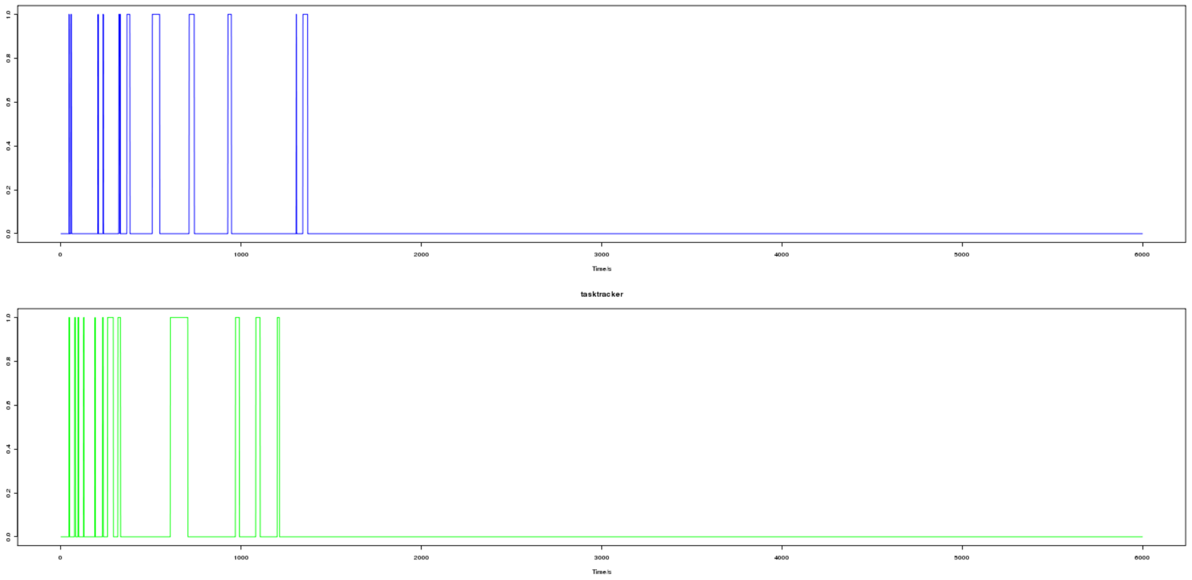

Figure 17: Plot of change points (binary indicators) of resource metric (CPU utilization, user%) in top panel, and of change points of application state counts for TaskTracker in bottom panel, with x-axis measured in seconds for both plots; Trace of a single run of a Nutch workload.

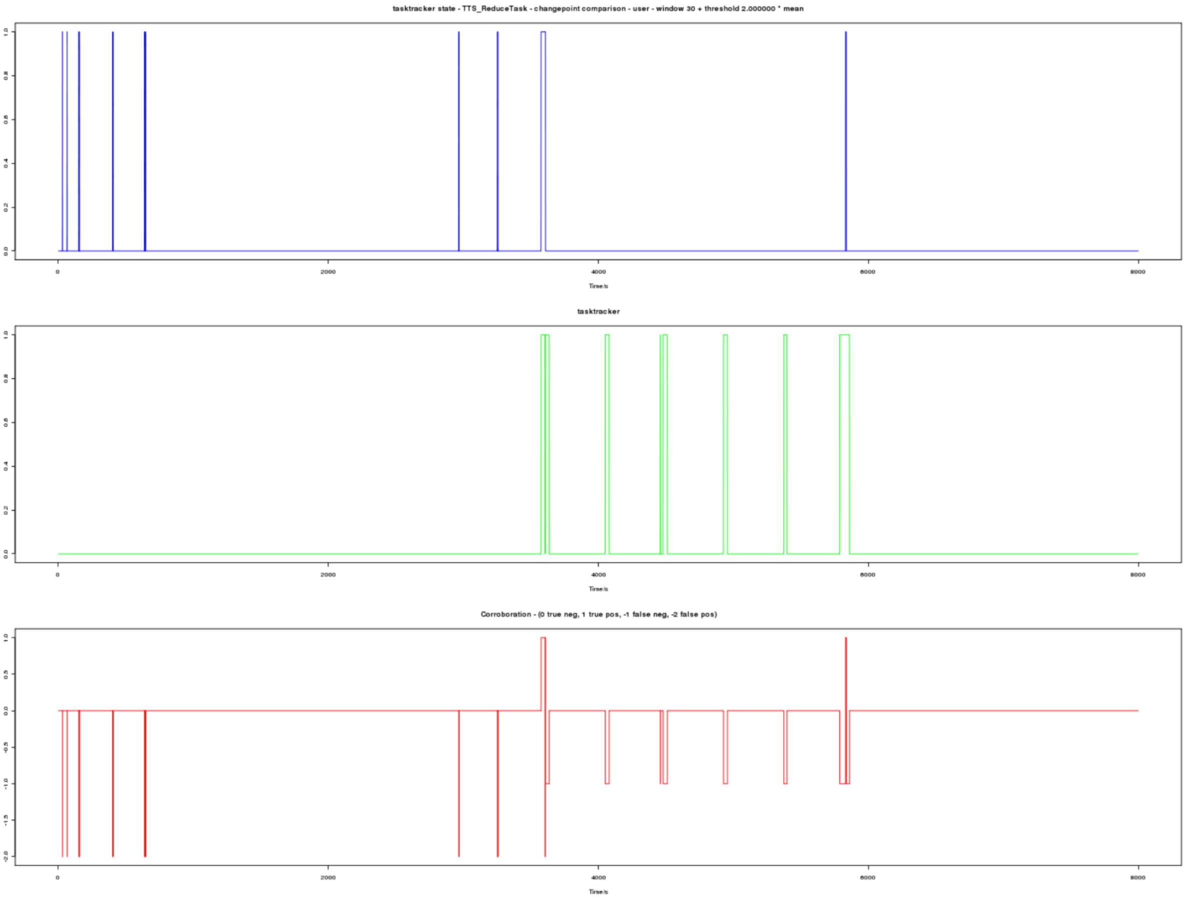

Figure 18: Plot of change points (binary indicators) of resource metric (CPU utilization, user%) in top panel, and of change points of application state counts for TaskTracker in the middle panel, and the evaluation score of the change point in the resource metric for predicting a change point in the application state count, with x-axis measured in seconds for all three plots; Trace of a single run of a `randomwriter` workload.

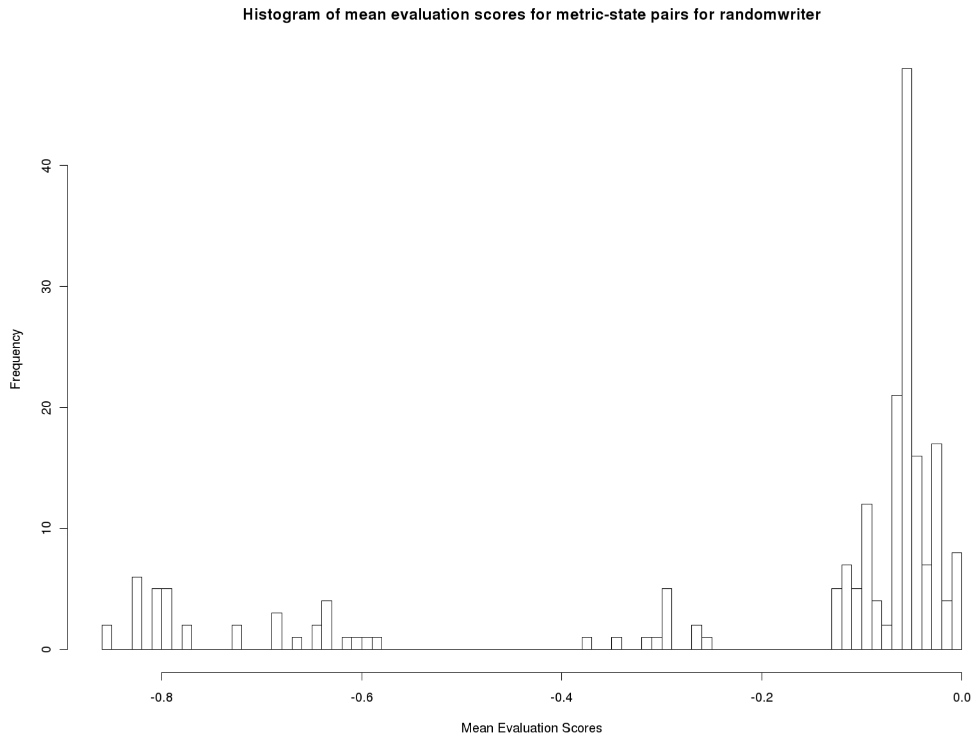

Figure 19: Plot of histogram of evaluation score values for each possible (application state)-(resource metric) change point series pair. Trace of a single run of a `randomwriter` workload on a single representative node.

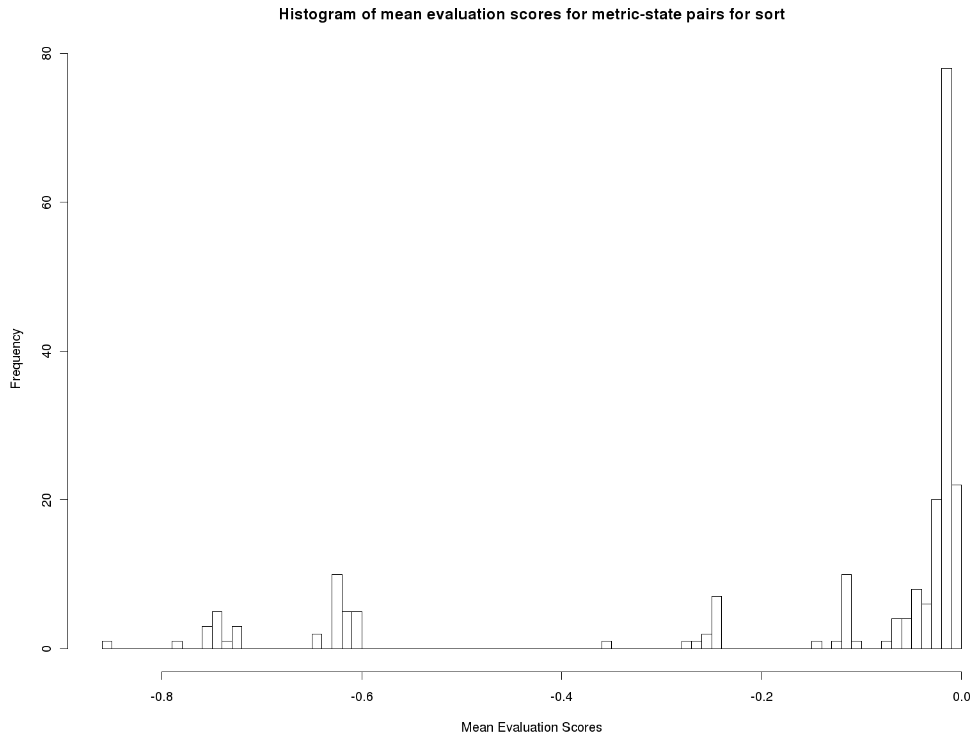

Figure 20: Plot of histogram of evaluation score values for each possible (application state)-(resource metric) change point series pair. Trace of a single run of a `sort` workload.

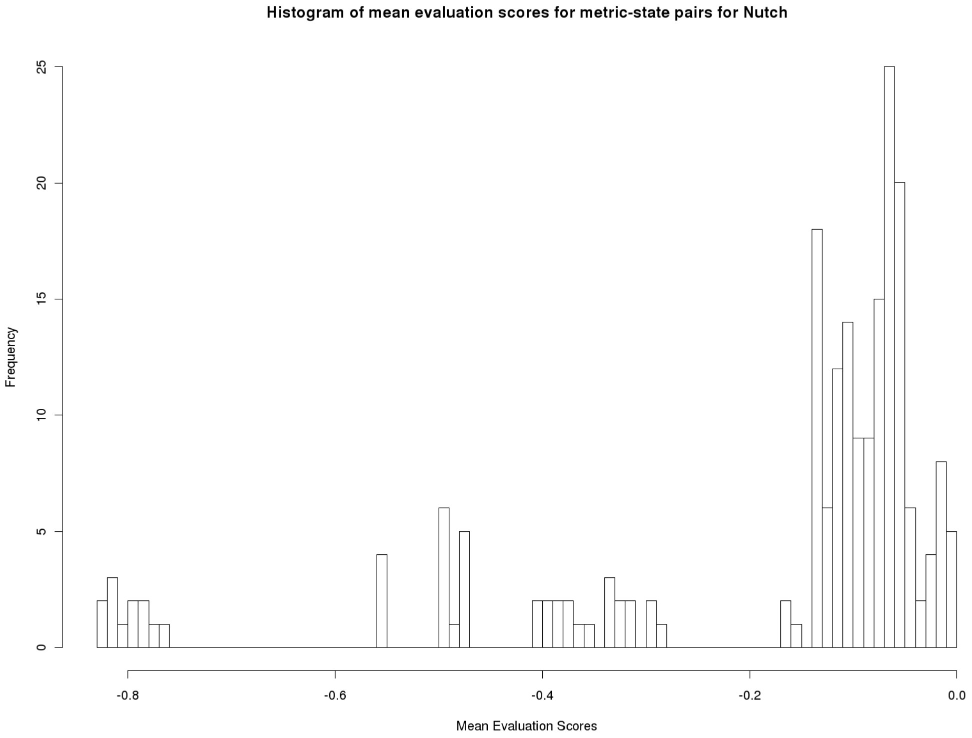

Figure 21: Plot of histogram of evaluation score values for each possible (application state)-(resource metric) change point series pair. Trace of a single run of a `Nutch` workload.

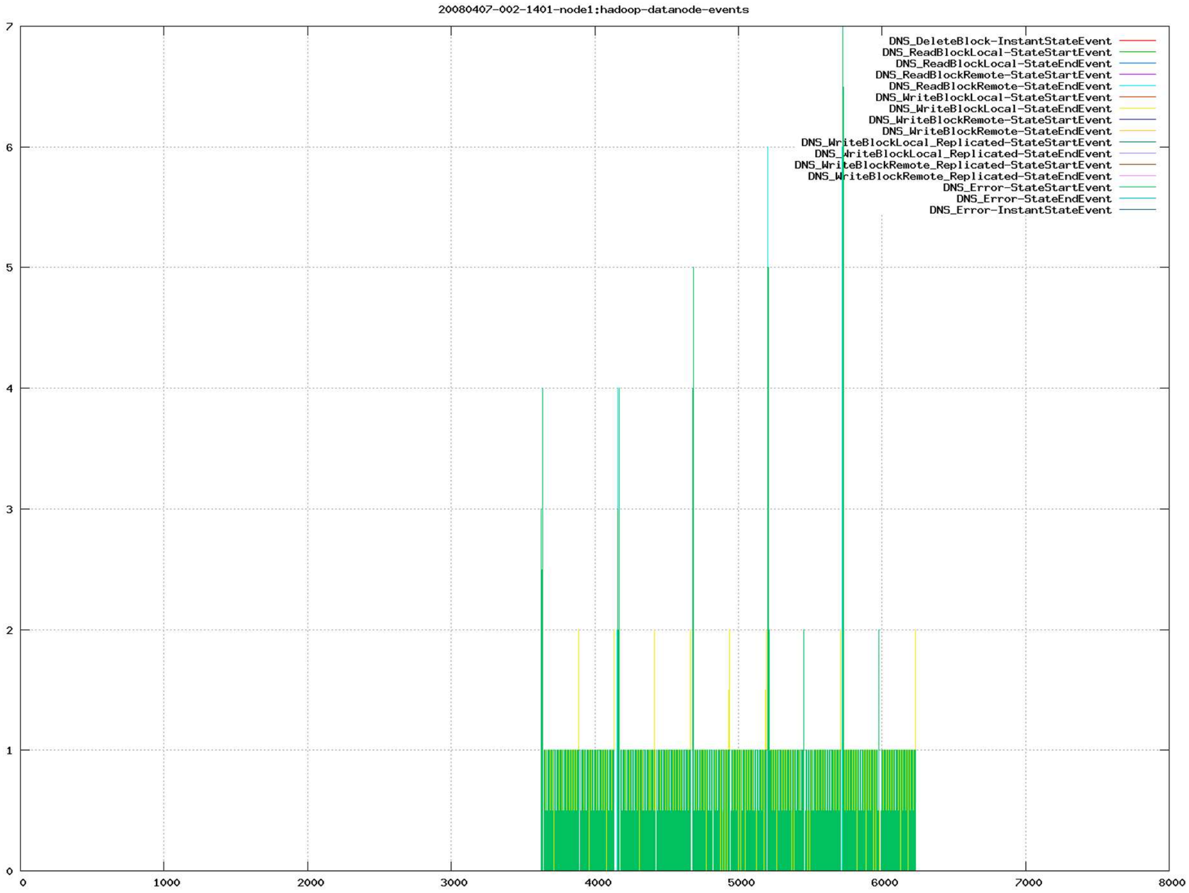

Figure 22: Plot of time series of counts of application events as reported by the Hadoop DataNode, as parsed by our Hadoop log parser.

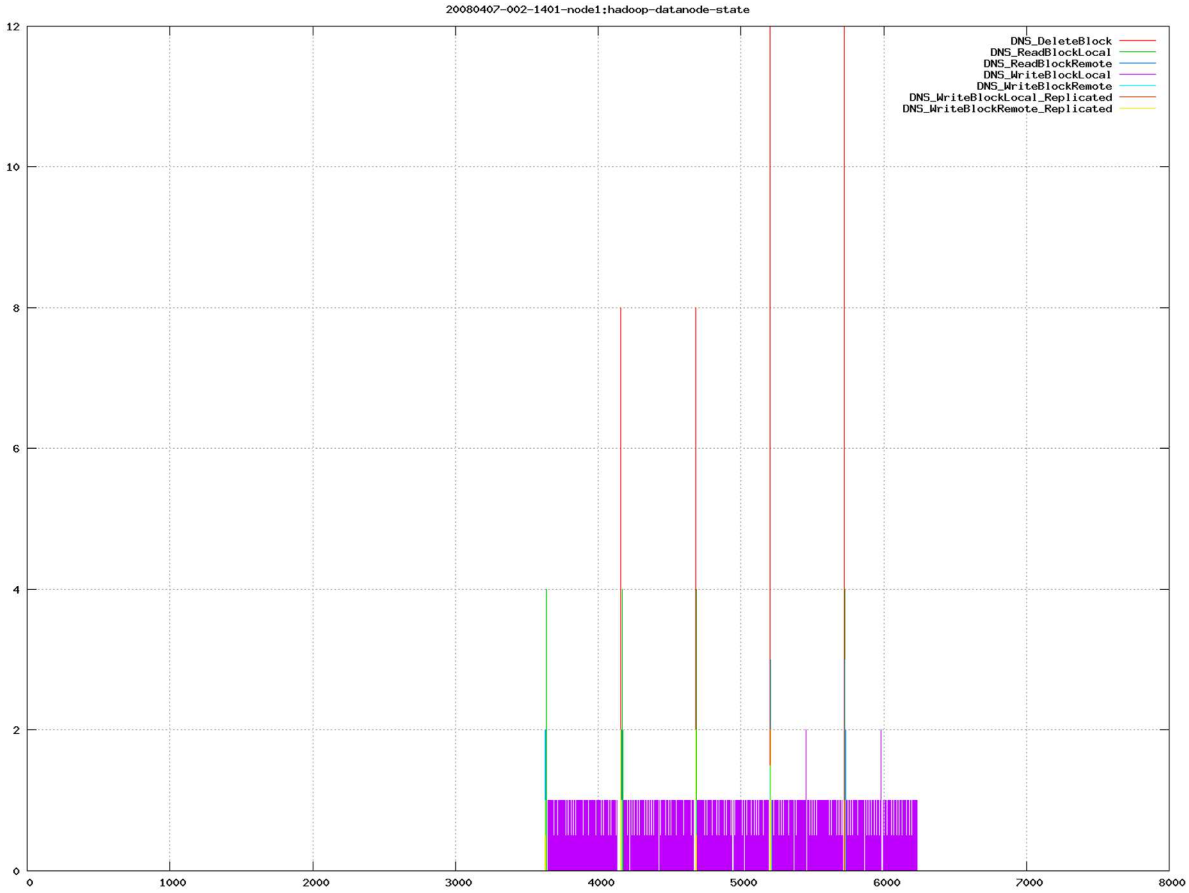

Figure 23: Plot of time series of counts of application states as reported by the Hadoop DataNode, as parsed by our Hadoop log parser.

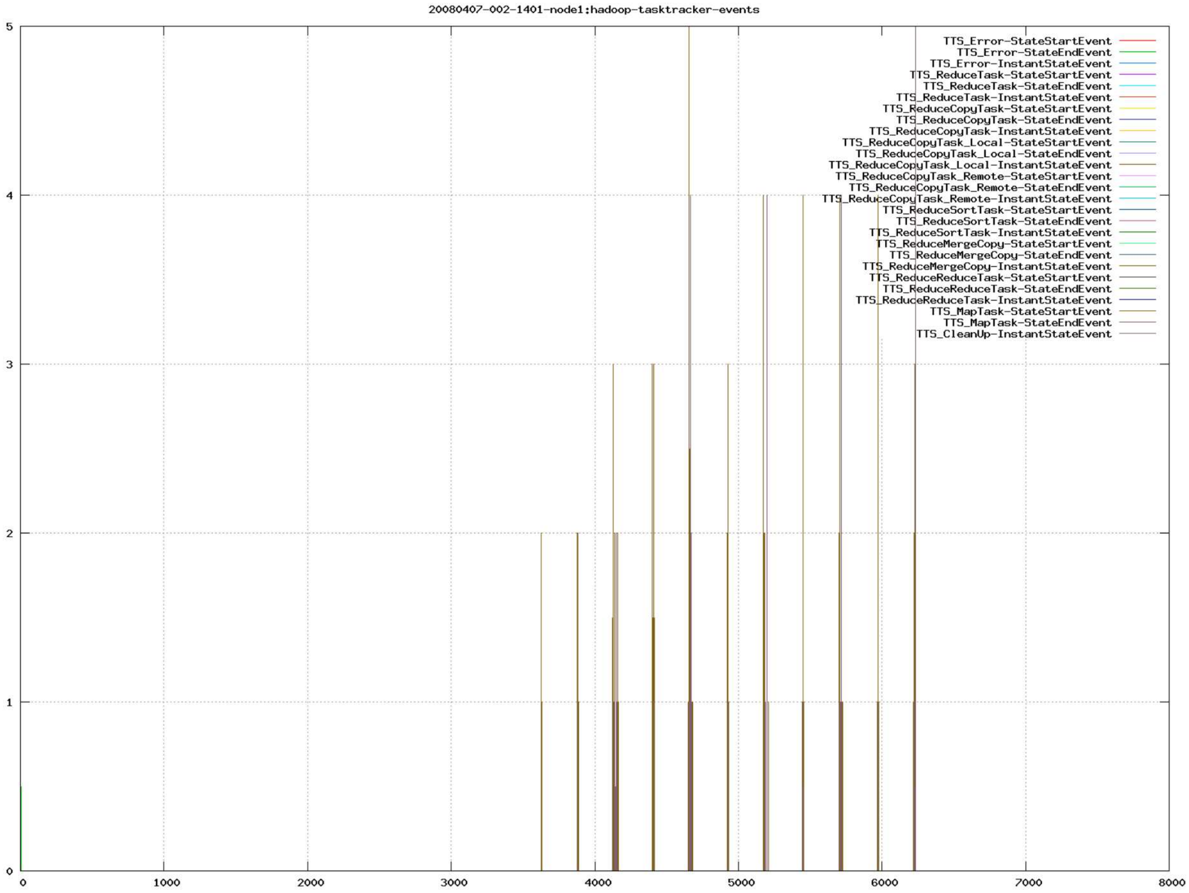

Figure 24: Plot of time series of counts of application events as reported by the Hadoop TaskTracker, as parsed by our Hadoop log parser.

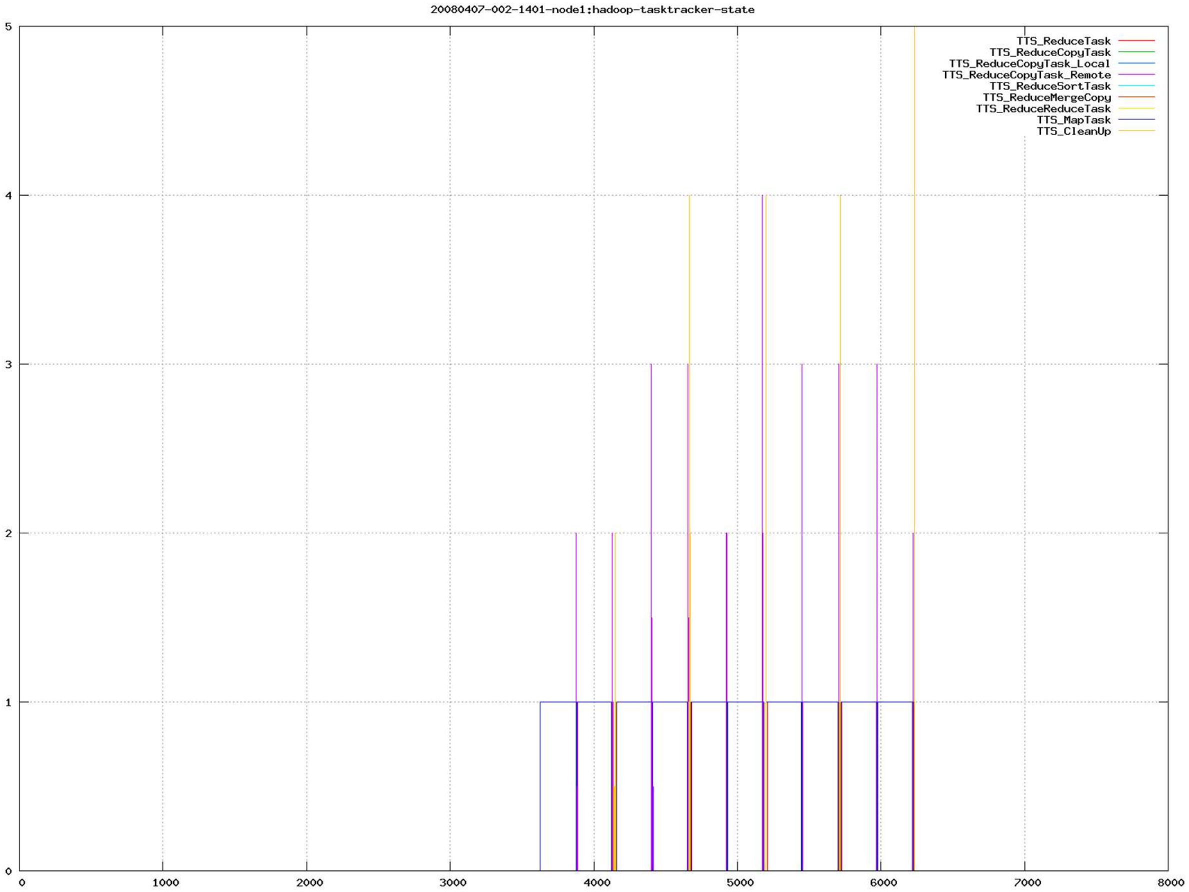

Figure 25: Plot of time series of counts of application states as reported by the Hadoop TaskTracker, as parsed by our Hadoop log parser.