CMU-PT/RNQ/0015/2009

Trustworthy and Resilient Operations in a Network Environment

Deliverable D4

First Evaluation of the Failure Diagnosis Algorithm

Information and Communication Technologies Institute
Carnegie Mellon | PORTUGAL
AN INTERNATIONAL PARTNERSHIP

Executed by: **Electrical and Computer Engineering**

Direction/Department: **CMU**

Date: **20-04-2012**

Version Number	Owner	Change Control	Date
0.1	CMU	Document creation with the first draft specification of the diagnosis algorithm	04-20-2012

Additional information:

Author/s:	...
Beneficiaries contributing on the deliverable:	CMU
WP contributing to the deliverable:	WPx
Estimation of pm spent on the deliverable:	X
Nature of the deliverable:	Report
Total number of pages:	34

— Document generated on 20 April 2012 at 06:43 —

Contents

1 Introduction **7**

2 Related Work **9**

3 Problem Statement **11**

3.1 Goals and Non-goals 11

3.2 Fault Model 11

3.3 Assumptions 12

4 Target Systems **13**

5 Approach and Implementation **15**

5.1 The *RAMS* Hypothesis 15

5.2 Illustration 15

5.3 *RAMS-FD*: Failure Detection Algorithm 16

5.4 *RAMS-FD* Detection Algorithm Parameters 18

5.5 *RAMS-DT*: Problem Classification using Decision Trees 19

5.6 Complexity and Scalability 19

5.7 Implementation and Parameters 20

6 Experimental Evaluation **21**

6.1 Evaluation Approach and Criteria 21

6.2 Experimental Setup: Data-Intensive Processing 23

6.3 Experimental Setup: Multi-Tier Web Request Processing 24

7 *RAMS-FD* Results: Data-Intensive Processing **25**

7.1 Summary of Results 25

7.2 Virtualized vs. Non-virtualized 26

7.3 Metric Choice 27

8 *RAMS-FD* Results: Multi-Tier Web Systems **27**

8.1 Summary of Results 27

8.2 Metric Choice 28

9 Results: *RAMS-DT* for Data-Intensive Processing **29**

10 Summary and Future Work **31**

List of Figures

1 Architecture of Hadoop. 13

2 Time-series of User-space CPU utilization (red), and the resource metric chosen, Kernel-space CPU utilization (blue) (above), and the correlation between the two metrics (below), for the duration of one experiment. Horizontal axes show time elapsed in seconds. A CPU hog was injected on 1 node in a 5-node Hadoop cluster. Correlation falls when the fault is injected 500 seconds into the experiment. 16

3 F-scores for *RAMS-FD* for all experiments for Hadoop MapReduce using correlations between user-space CPU utilization and all resource metrics. 25

4 F-scores for *RAMS-FD* using the best correlation metric for all experiments for Hadoop MapReduce. The best correlation pair is listed for each fault. . . 25

5 F-scores for all experiments for RuBiS using correlations between user-space CPU utilization and all resource metrics, and for top metrics. (See Fig. 3 for legend.) 28

6 Decision-tree learned for the Nutch (on FP/EC2 test-beds) Hadoop workloads. 29

Executive Summary

This project aims to develop a fault intrusion tolerant framework for monitoring cloud infrastructures in a trustworthy way. It targets to fullfil typical datacenter scenarios as those of Portugal Telecom. The framework will improve resiliency and guarantee the needed trustworthiness in a specific environment identified by PT.

We present the First Specification of the Failure Diagnosis algorithm henceforth referred to as *RAMS* for the deliverable D3 proposed in the TRONE project.

Detecting failures in large-scale distributed systems is challenging, as modern datacenters run a variety of applications and systems. Current techniques for detecting failures often require training to detect failures, have limited scalability, or are not intuitive to sysadmins. We present *RAMS*, a lightweight and scalable algorithm for distributed systems which detects failures using only correlations of operating system metrics collected transparently. The detection algorithm is based on our hypothesis of server application behavior, and hence does not require training, and can perform detection with complexity linear in the number of nodes, with results that are intuitively interpretable by sysadmins. Further, with some training, *RAMS* can identify the category of a problem that has previously been seen and determine the root cause of the problem. As part of our work for year 2 of the project, we plan to show that *RAMS* is versatile, and can diagnose several faults in i) multi-tier web request systems that often run in datacenters, ii) in Hadoop MapReduce systems that are often used to parallelize search, data mining and machine learning algorithms, and further show how *RAMS* is intuitive to sysadmins.

The rest of this deliverable is structured as follows. We briefly introduce the need for failure diagnosis techniques in Section 1. Then, we present a brief survey of related work on failure diagnosis techniques. The goals of our failure diagnosis algorithm are covered in Section 2 including the problems that we plan to target and also the assumptions that the diagnosis technique relies upon to work correctly. A detailed description of the diagnosis algorithm is presented in Section 4 along with hypothesis on which the failure diagnosis algorithm is based upon and some empirical evidence in support of our hypothesis. Finally, we discuss some of our future steps in Section 10.

1 Introduction

Businesses often rely on large-scale cloud computing systems to support Internet and telecommunication services such as e-Commerce, VoIP and business analytics. To satisfy the high availability requirements of these systems, there are real-time operations teams that diagnose problems by monitoring both low-level alarms derived from the equipment, *e.g*, server and network errors, as well as end-to-end indicators such as customer complaints. These operations teams work to ensure that major outages or blackouts are rare, and that they are resolved quickly when they occur. On the otherhand, performance degradations or request failures affecting a subset of end users—occur more frequently and are much more elusive to diagnose. For example, Thereska et al [1] describe a TCP buffer overflow at a switch which resulted in performance degradation when striping data over more than 5 servers that was difficult to diagnose manually. Sambasivan et al [2] describe a bug in their distributed storage system in which the hash table responsible for storing mappings from filehandles to object names on disk stored mappings only in the lower bucket resulting in increased query times. Thus there is a significant need and interest in developing techniques that can detect failures and performance degradation problems rapidly.

Detecting failures and performance problems (hereby collectively rederred to as "failures") in large scale distributed systems is highly challenging. As distributed systems grow, the data to be analyzed grows as well. In addition, interactions between components become more complex. Distributed systems have been growing to meet larger processing demands, e.g. due to high volumes of web requests in multi-tier Internet services, and large datasets in data-intensive systems e.g. MapReduce [3]. This growth has been facilitated by pay-as-you-use compute facilities e.g. Portugal Telecom's cloud computing service and Amazon's Elastic Compute Cloud (EC2). This growth has created new challenges in quickly detecting failures in large systems. Algorithms for failure detection need to efficiently handle larger and more complex datasets from larger systems, and from different types of systems.

Failure detection and diagnosis in distributed systems has been the subject of much recent research. Some techniques extract/infer application-specific information e.g. requests paths, to detect anomalies [4, 5, 6, 7, 8, 9], but this may be computationally expensive, and the instrumentation is often invasive and infeasible on production systems. Others infer failures using application-agnostic system metrics [10, 11, 12, 13], but they reason about metrics from all nodes in the system at once, which may not scale to large systems.

While many of the above-mentioned techniques may be effective at isolating failures, many are too expensive to be "always-on" due to invasive instrumentation and high overheads.

Instead, we target lightweight techniques to quickly detect if there is a failure in the system, before heavyweight techniques are used. We propose a new method, *RAMS*, for failure detection on distributed software systems, that is lightweight, scalable and versatile to be applicable to multiple types of systems. *RAMS* consists of two algorithms: (1) *RAMS-FD* is a lightweight failure detection algorithm which provides first-pass alarms to notify sysadmins of failures, allowing them to use more sophisticated techniques which are too expensive to be used in an "always-on" fashion. *RAMS-FD* is based on our *RAMS* **hypothesized model (§ 5.1) of normal, fault-free system behavior of server applications**. *RAMS-FD* is lightweight: it has low instrumentation overhead ($< 1\%$), using only few (≈ 10) widely-available OS performance counters. It is also scalable: (i) it uses simple computations which are linear in the size of the input data, and (ii) it can detect failures in a distributed system

We have evaluated the *RAMS* algorithms on two major classes of distributed server applications: (i) Apache/J2EE/MySQL-based RuBiS auction benchmark, which is a multi-tier web request processing application, and (ii) on the Hadoop [14] MapReduce [3] parallel distributed data-intensive processing system. Our contributions are: (i) a hypothesized model of normal, fault-free behavior in distributed server applications, (ii) a lightweight, scalable failure detection algorithm, *RAMS-FD*, based on this hypothesis, (iii) a failure diagnosis algorithm, *RAMS-DT*, using decision-trees on *RAMS-FD* decision alarms, and (iv) a detailed experimental evaluation of our algorithms on these two classes of systems.

2 Related Work

Machine learning has been commonly used for failure diagnosis in distributed systems [4, 5, 7, 10, 11, 12, 13]. Given knowledge of failed requests (e.g. those with Service Level Objectives, or SLOs, violated), supervised learning techniques localize failures to their originating node or metric [4, 5, 10] based on models learned from training data. *RAMS-FD*'s detection does not require any training as it is based on the *RAMS* hypothesis, although the *RAMS-DT* diagnosis requires training. Another class of techniques extracts request paths in the system for diagnosis [6, 7, 8, 9]. *RAMS* uses OS-level metrics directly for diagnosis.

Another class of techniques solves the more fundamental problem of generating alarms to notify users of a failure, when knowledge of request failures is unavailable [11, 15, 16]. This is useful for systems with long running jobs or noevl user-programmable workloads, e.g. MapReduce, or in the case of partial failures, when the problem is a nascent one which has not escalated into an SLO violation. *RAMS* falls in this class of techniques, and solves both instances of the problem–for both Hadoop and for multi-tier web request systems. Like *RAMS*, [11] uses OS-level metrics. They focus on macro datacenter state, whereas *RAMS* diagnoses problems on individual nodes. *RAMS-FD* is also more scalable than [11] as a failure detector as it requires only local state for each node.

Similarly, [15] uses metrics from all nodes in a system, while *RAMS-FD* uses only node-local metrics for diagnosis. Similarly to *RAMS*, [16] uses correlations, but they correlate behavior across nodes, incurring $O(n^2)$ complexity, while *RAMS* correlates behavior between metrics on the same node, which is more scalable. [17] focused on a method to cheaply collect system metrics to enable reconstruction of past incidents for investigation. Their correlation-like "rules" for inferring system problems, were not generalized, unlike the *RAMS* hypothesis.

Some diagnostic approaches have used regression to automatically discover correlations between metric pairs [18, 19]. However, these approaches do not scale well to large numbers of nodes and metrics as they search for metric correlations both locally, and remotely between nodes. *RAMS-FD* exploits semantic knowledge to analyze a small number of metrics, providing scalable diagnosis for large-scale systems. Cherkasova et al [20] and Stewart et al [21] exploited queuing theory and regression to model the relationship between resource-usage and transaction response times in Internet applications. This allowed them to distinguish between workload changes and anomalies for transaction types in their training sets. *RAMS-FD* relies solely on OS metrics to detect problems allowing it to be easily ported across systems. While [22], like *RAMS-DT* uses decision-trees to diagnose failures, *RAMS-DT* is a second-order algorithm applied to detection alarms, while [22] was applied directly to

application logs. *RAMS-DT* applies our earlier work, BliMeE, on using decision-trees on detection alarms [23] to *RAMS-FD* detection alarms. Some techniques have diagnosed failures on Hadoop using white-box information from Hadoop's natively generated logs [24, 25, 26], rather than the black-box OS-level metrics that *RAMS* uses. [12, 13] used OS metrics to diagnose failures in Hadoop, albeit using a peer-comparison approach with $O(n^2)$ complexity. *RAMS* is more scalable, and has wider applicability to multi-tier web request processing systems as well.

3 Problem Statement

3.1 Goals and Non-goals

The *RAMS* algorithms aim to perform failure diagnosis on server applications in distributed systems that is:

Lightweight, Transparent: We aim to use metrics that can be collected with minimum overhead and transparently, i.e. without modifying target applications. This would allow the metrics to be continuously collected without impacting system performance, and enable our algorithms to be used in production settings, where sysadmins are unlikely to allow invasive or high-overhead instrumentation.

Scalable: Our algorithms must have low computational complexity to enable them to scale up to diagnose large distributed systems. We aim for *RAMS-FD* to require only information from a single node to diagnose that node, so that diagnosis can be distributed across the system. We also aim for *RAMS-DT* to efficiently classify failures, with low computational complexity.

Versatile: We aim to detect failures in multiple types of systems. This would enable our algorithm to be used in large, complex modern datacenters running multiple types of applications.

Effective Diagnosis: Our algorithm aims to be effective in detecting failures in our target systems–we aim to minimize false-positives while detecting as many failures as possible. We measure the performance of *RAMS-FD* using the F_1 score (§6.1), and we aim to maximize F_1. We also aim for *RAMS-DT* to accurately classify whether a node is free from fault, or if it is faulty, to classify the closest previously observed fault it is suffering from.

Non-goals: We do not present an online implementation of the *RAMS* algorithms. Instead, we focus on presenting and evaluating the diagnosis of our algorithms. As there is only one master node in a Hadoop cluster, we currently focus on failures on the potentially many of slave nodes, and defer master node failures to future work.

3.2 Fault Model

RAMS targets performance problems: faults that result in a slowdown, causing the processing of tasks in the system to take a longer time than without the fault, causing increased runtime and reduced system throughput. Performance problems can also be considered as *partial failures*, since they do not result in a system crash. We do not consider crashes as

these can be detected using simpler methods e.g. heartbeats. Nonetheless, we envision that *RAMS* would be able to detect crashes before the underlying fault has resulted in an outright crash. We do not target value faults from executions terminating with incorrect results.

While *RAMS* targets performance problems it can also be used to diagnose some of the security related problems as well. Several security problems, such as Denial of Service (DoS) attacks, can manifest as performance problems and can, therefore, be diagnosed using *RAMS*. One of the important issues that this project will try address in Years 2 and 3 will be the effectiveness of *RAMS* in correctly attributing the root cause of the performance degradation to a security attack and the kind (DoS etc.) of the security attack.

3.3 Assumptions

We assume that the target application under diagnosis is the dominant source of activity in the operating system on each node, as we use system-level OS metrics *(This assumption can be discharged by tracking per-process performance counters, but this incurs slightly higher overheads to collect)*. This is likely to be the case in virtualized environments where each VM hosts a single service.

4 Target Systems

Data-Intensive Processing MapReduce [3] is a framework that enables distributed, data-intensive, parallel applications by enabling a job described as a Map and a Reduce function to be decomposed into multiple copies of Map and Reduce tasks and a massive data-set into smaller partitions, so that each task processes a different partition in parallel. Hadoop has a master-slave architecture, with a single master and multiple slave hosts, as shown in Figure 1. Hadoop consists of an execution layer which executes Maps and Reduces, and the Hadoop Distributed Filesystem (HDFS), an implementation of the Google FileSystem [27]. The master host runs the JobTracker daemon, which schedules task execution on slaves and implements fault-tolerance using heartbeats sent to slaves, and the NameNode daemon, which provides the namespace for HDFS. Each slave host runs the TaskTracker daemon, which executes Maps and Reduces locally, and the DataNode daemon, which stores and serves data blocks for HDFS. Each Hadoop daemon is a Java process, and natively generates logs which records error messages, as do typical logs, as well as system execution events, such as the starts and ends of Maps and Reduces. We currently only target detection of faults on the slave nodes.

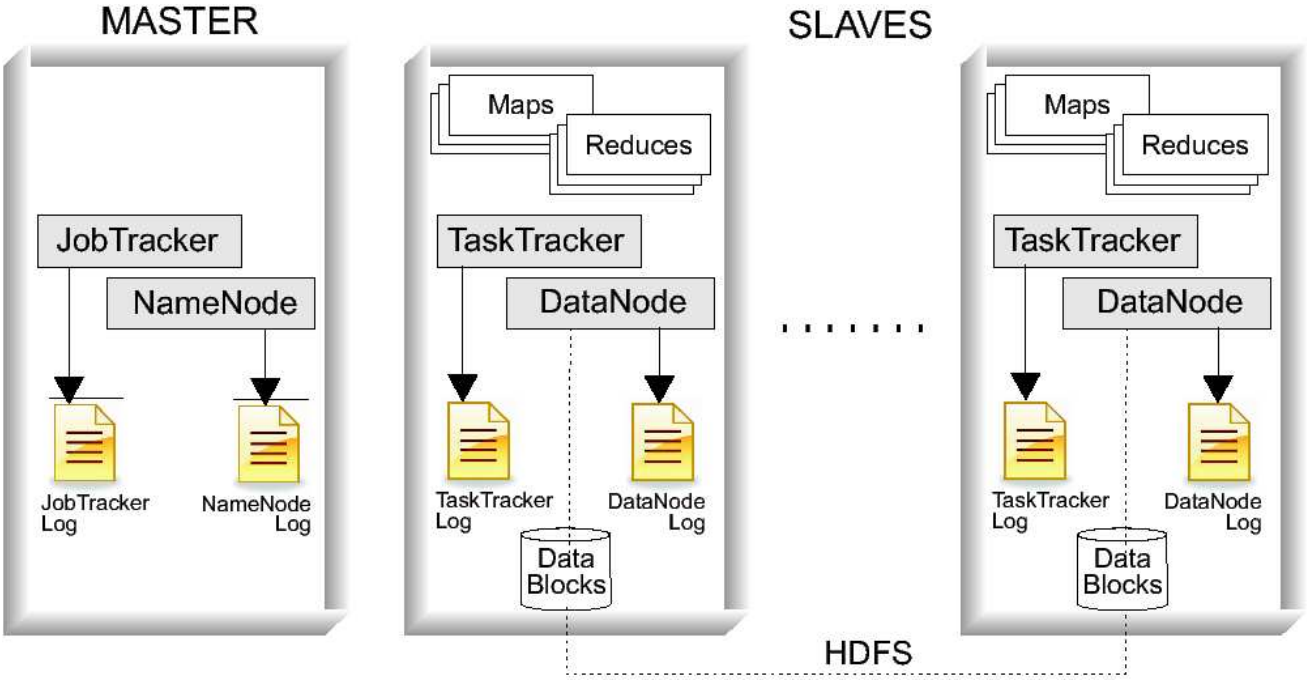


Figure 1: Architecture of Hadoop.

Multi-Tier Web Request Processing RuBiS [28] is an auction website benchmark modeled after the popular eBay auction website. RuBiS has a three-tier architecture, consisting of a web-server, an application server (using Java Servlets and Enterprise JavaBeans, or EJB) and a database server. In our setup, we used the Apache HTTPD web server, the JBoss J2EE (Java 2 Enterprise Edition) application server for handling the business logic of the application, and the MySQL relational database. Our setup consisted of 1 web server, 3 JBoss servers in round-robin load-balanced mode, and a single MySQL database server.

5 Approach and Implementation

5.1 The *RAMS* Hypothesis

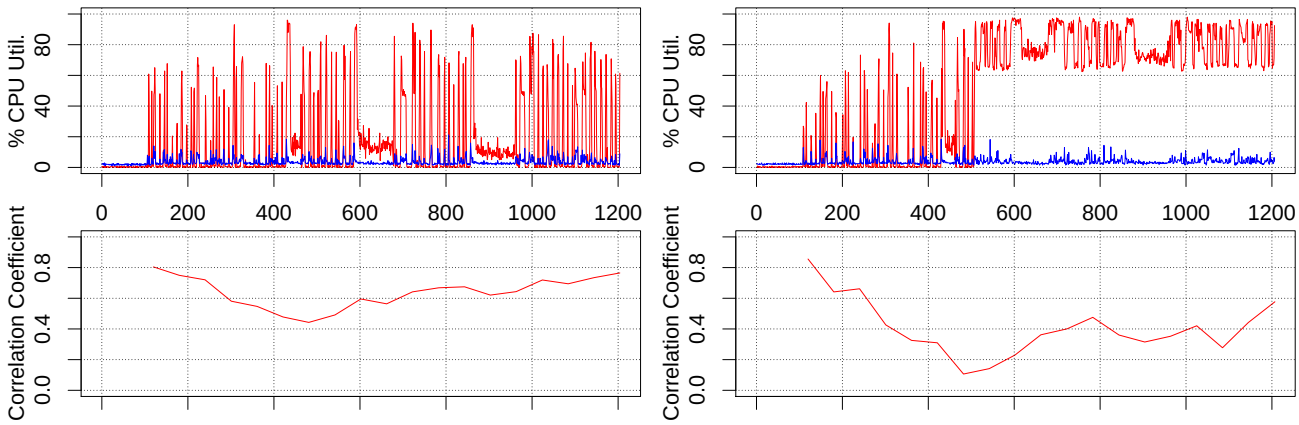
The *RAMS* hypothesis proposes that at a conceptual level, normal, fault-free processing to service a user request in a server application comprises, and alternates between, two phases: (i) a **communications** phase, when the application receives instructions from the user via the network (potentially indirectly, e.g. database receiving instructions from the web server in multi-tier web request processing, or slave nodes receiving instructions from a MapReduce master node), reads data from disk, or passes results back to the user via the network, or to disk, and (ii) a **compute** phase, when the application performs some operation based on the received inputs and instructions. At an operational level, the **compute** phase is marked by increased user-space CPU activity, while the **communications** phase is marked by increased activity in one or more of the system resources: disk, network, or kernel-space CPU activity to service the disk and network operations. Hence, application activity alternates between these two phases at a micro-level (e.g. in time-scales of micro- to milli-seconds, at the granularity of a single thread of execution). Then, we hypothesize that, taken at a macro-scale, this alternating activity of multiple interleaved threads induces correlated behavior at the macro-level between user-space CPU activity and system resource consumption.

Further, we hypothesize that in the presence of failures in the system, at the micro-level, user-space **compute** activity would show a marked deviation in its relationship with the indicators of the resources used during the **communications** phase (i.e. disk, network, kernel-space CPU activity). In the window of observation, either processing activity is impeded, and the **compute** phase dominates the **communications** phase, e.g. when there is a hang in the processing of the user job, or the **communications** phase dominates the **compute** phase, e.g. when there are problems with the disk, network, and other system resources, or when the **compute** task terminates prematurely, leading to a quick return to the **communications** phase. These micro-level disruptions then lead to macro-level disruptions in the correlated behavior between user-space CPU activity, and the system resources, i.e. disk, network, or kernel-space CPU activity.

5.2 Illustration

We illustrate the intuition behind the *RAMS* hypothesis of server application behavior with an example. Figure 2 shows a trace of the user-space CPU utilization, U, and the resource measure, the kernel-space CPU utilization, K, on two slave nodes in the same Hadoop

MapReduce cluster processing the same MapReduce job. We inject an external CPU load which consumes 70% of CPU utilization to simulate a fault on one node. Figure 2(b) shows a trace of the faulty node, where U increases significantly and remains high after the fault is injected 500 seconds into the experiment, while Figure 2(a) shows the trace of a fault-free node, where U and K exhibit similar behavior. Comparing the movement of the correlation coefficient $\rho_{U,K}$ for user- and kernel-space CPU utilization between the faulty and fault-free nodes, we can also see that $\rho_{U,K}$ remains significantly higher for the fault-free node as compared to the faulty node, and $\rho_{U,K}$ falls significantly after the fault is injected. This difference in the behavior of $\rho_{U,K}$ on faulty versus fault-free operation, provides support for the *RAMS* hypothesis.



(a) Normal node (b) Faulty node

Figure 2: Time-series of User-space CPU utilization (red), and the resource metric chosen, Kernel-space CPU utilization (blue) (above), and the correlation between the two metrics (below), for the duration of one experiment. Horizontal axes show time elapsed in seconds. A CPU hog was injected on 1 node in a 5-node Hadoop cluster. Correlation falls when the fault is injected 500 seconds into the experiment.

5.3 *RAMS-FD*: Failure Detection Algorithm

Based on our hypothesis, the *RAMS-FD* algorithm computes the Pearson's correlation coefficient (ρ) between user-space CPU utilization and a resource metric for one of the resources (disk, network, kernel-space CPU utilization) used in the **communications** phase, and raises an alarm indicating a failure when the coefficient falls below a threshold, T. This computation is done independently for each of node in the system being diagnosed, for each time-series of sampled metrics collected on that node. $\rho \in [-1.0, 1.0]$ measures the degree of correlation, or linear dependence, between two variables. Its value is the ratio of the

covariance between two variables to the product of their standard deviations. For instance, the Pearson's correlation coefficient between user-space and kernel-space CPU utilization is as follows:

$$\begin{aligned} \rho_{U,K} &= \frac{covariance(U,K)}{\sigma_U \sigma_K} \\ \text{where } U &= \text{samples of user-space CPU utilization} = u_1, u_2, \ldots, u_n \\ K &= \text{samples of kernel-space CPU utilization} = k_1, k_2, \ldots, k_n \end{aligned}$$

In total, we compute **correlation coefficients** between user-space CPU utilization, U, and 7 resource-related metrics. Together with U, these raw metrics are sampled from `/proc` using the `sar` tool once per second:

1. $K = system\%$: Kernel-space CPU utilization
2. $DR = rtps$: Disk read transactions per second
3. $DW = wtps$: Disk write transactions per second
4. $NR = rxpck$: Network packets received
5. $NW = txpck$: Network packets transmitted
6. $MR = pgpgin$: Memory pages paged in
7. $MW = pgpgout$: Memory pages paged out

We refer to each of these 7 correlations as *RAMS-FD* detection metrics. In addition, we create compound measures of the conformance of the system's behavior to the *RAMS* hypothesis. This allows a combination of activity in multiple resource categories (disk, network, kernel-space activity) across time (e.g. user activity U may be strongly correlated with disk metrics at one point and network metrics at another), whereas a correlation between U and any of the above metrics measures only correlation with a single resource category. We create two compound metrics which take the maximum of the correlation between U and multiple metrics. They are:

$$\begin{aligned} \rho_{U,DiskNet} &= max \quad \{\rho_{U,DR}, \rho_{U,DW}, \rho_{U,NR}, \rho_{U,NW}\} \\ \rho_{U,DiskNetSys} &= max \quad \{\rho_{U,K}, \rho_{U,DR}, \rho_{U,DW}, \rho_{U,NR}, \rho_{U,NW}\} \end{aligned}$$

To raise alarms, we consider detection windows of D consecutive correlation coefficients, and raise an alarm if any of the D correlation coefficients falls below the threshold value T. The detection window also smooths the raw metrics to remove spurious correlations not indicative of true processing e.g. those caused by perfect sawtooth patterns during idle activity. To reduce the noise from the sampled parameters, we use a low-pass mean filter: we take the means of metric in small windows (in the order of less than 10s, discussed in §5.4) before computing the correlation coefficient.

Hence, for each node (host) in a system under detection by *RAMS-FD*, the algorithm takes as input the time-series's of user-space CPU utilization and of 1 of the 7 resource-related metrics (and the 2 additional compound metrics), and outputs a list of times when alarms are raised, when the correlation coefficients generated in the detection window D result in $\geq F$ threshold violations. For a distributed system with N nodes, N separate lists of violation times are generated. *RAMS-FD* does not need any training. It takes a user-specified threshold, T, and reports correlations which fall short of the threshold.

5.4 *RAMS-FD* Detection Algorithm Parameters

In total, there are 5 tunable parameters in the *RAMS-FD* algorithm. They are:

1. *LW, Low-pass Window Width*: Raw metric samples to take mean of
2. *LS, Low-pass Window Slide*: Raw metric samples to slide low-pass filter by
3. *W, Correlation Window Width*: Samples (after low-pass filter) to correlate
4. *S, Correlation Window Slide*: Samples to slide detection window by
5. *D, Diagnosis Window*: Correlation coefficients to consider

Then, the latency of the algorithm (i.e. minimum time between when alarms can be raised) in the steady state is the product $LS \times S$, i.e. the time interval between when *RAMS-FD* can raise each alarm is $LS \times S$ seconds, while the amount of information considered in each flag raised is $LW \times W$. Finally, we consider the threshold, T, to be a run-time user parameter that is user-tunable according to whether the user would prefer more alarms to be returned while suffering higher false-positives. For instance, in times when the sysadmin is inundated by other tasks, he can increase the threshold to reduce the amount of attention he needs to pay to the system, while he can reduce the threshold to investigate more alarms when he has the luxury of time to do so. We discuss parameter choices in §5.7.

5.5 *RAMS-DT*: Problem Classification using Decision Trees

In the *RAMS-DT* algorithm, we learn decision trees from the detection alarms from *RAMS-FD* to classify the type of failure encountered according to past labeled failures. *RAMS-FD* returns 9 binary detection alarms for each node in a system under diagnosis, one for each category of system resources. These independent alarms can be reasoned about collectively to form a more complete diagnosis. We use decision trees to classify nodes by their collective behavior, as observed from the 9 *RAMS-FD* alarms, to determine if the node's behavior is similar to those in a previously diagnosed and labeled failure. One decision-tree is learned for each workload under diagnosis, as unlike *RAMS-FD* which is based on the general *RAMS* hypothesis, *RAMS-DT* performs best when decision-trees are tailored to the workload.

The *RAMS-DT* decision-tree is a binary-tree, where each interior node consists of one of the *RAMS-FD* detection metrics, and the left or right branch is taken depending on whether an alarm was raised for the metric. The decision-tree is learned from length-9 vectors of *RAMS-FD* alarms from a node in a distributed system, and a corresponding label of whether that node is fault-free, or the name of the fault it is suffering, for the duration of the *RAMS-FD* alarms. Then, the leaves of the decision-tree are used to classify subsequent nodes. The vector of detection alarms for the node is used to traverse the decision tree to a leaf node. Then, the *RAMS-DT* decision-tree returns the most frequently occurring training label at the leaf node in the tree. An approximate confidence in this diagnosis can also be obtained from the "purity" of the training nodes: the proportion of training nodes with the returned label. When this proportion is low, users can be prompted to carry out further manual investigation. We defer this aspect of *RAMS-DT* usage to future work.

In a deployment setting, we expect sysadmins would first respond to the basic *RAMS-FD* detection alarms, and perform in-depth investigation, after which they would close the case by ascribing a label to the episode. This label can then be associated with the *RAMS-FD* alarms collected from the episode. Over time, a decision-tree can be learned from this built-up collection of labeled *RAMS-FD* detection alarms, and used to classify future episodes.

5.6 Complexity and Scalability

As *RAMS-FD* uses only information from each node to compute alarms for that node, the detection can be carried out independently of all other nodes. The computation for detection on each node can be carried out on that node itself. This saves the bandwidth for transmitting metrics to a central location, and reduces the complexity of the global failure detection computation to a constant in the size of the distributed system.

In addition, *RAMS-FD* relies on computing the Pearson's correlation coefficient. The multiplications and divisions required are linear in the number of samples of the two variables involved. The number of samples involved in is determined by the correlation window size, W, which is in turn independent of the number of nodes in the system. Hence, the computation of each coefficient is constant in the number of nodes in the system. On the whole, diagnosing a distributed system with k hosts/nodes will involve computing k coefficients; hence, *RAMS-FD* scales linearly with the size of the system, and is highly scalable. In addition, *RAMS-FD* has no added training overhead.

After decision-trees have been learned for *RAMS-DT*, classification will involve only a traversal of the learned trees, which are at worst linear in the size of the training data, but in reality its size will be limited by the 9 *RAMS-FD* detection alarms used to split its nodes, resulting in a small tree for traversal. As the focus of *RAMS* is on the scalable *RAMS-FD* detection, the learning of *RAMS-DT* decision-trees is critical to the lightweight nature of *RAMS-FD*.

5.7 Implementation and Parameters

RAMS is implemented as a set of Python scripts, and is currently built for offline bulk processing of collected instrumentation data. An early version of *RAMS* had previously been implemented for our ASDF (Automated System for Diagnosing Failures) [**?**] pluggable framework for automated data collection and online failure diagnosis. We used the following parameters in our evaluation: $LW = 5, LS = 5, W = 60, S = 12, D = 10$. This resulted in a diagnosis latency of $LS \times S = 60$ seconds, and each diagnosis result considered the last $LW \times W = 300$ seconds of system behaviour. We selected parameters to minimize the diagnostic latency, $LS \times S$. We found that $LS < 5$ gave poor diagnostic performance, and likewise with $S < 12$. We chose LW, W to be sufficiently large to consider relevant system behavior, while not picking excessively large values, which would include behavior from too far in the past that is no longer relevant. We found that smaller values of LW, W resulted in poor performance as well. We envision that an exponentially decaying window can also be used in place of LW, W, but would increase the computational complexity of the algorithm. **We used the same set of parameters for the algorithm for diagnosing faults in both our target platforms: on the RuBiS multi-tier web request processing system and the Hadoop MapReduce data-intensive processing system. This demonstrates the versatility of the *RAMS* algorithm.** Finally, our decision tree learning and classification algorithm for previously seen problems was implemented in GNU R using the *rpart* package for decision-tree construction.

Category	[Source] Reported Failure	[Problem Name] Problem-Injection Methodology
Resource Contention	[Hadoop users' mailing list, Sep 13 2007] CPU bottleneck as master and slave daemons ran on same host	[CPUHog] Emulate CPU-intensive task to consume 70% CPU util.
Resource Contention	[Hadoop users' mailing list, Sep 26 2007] Excessive messages logged to file during startup	[DiskHog] Sequential disk workload wrote 20GB of data to filesystem
Network Problem	[HADOOP-2956] Degraded network connectivity between DataNodes results in long block transfer times	[PacketLoss5/50] Drop packets with 0.05, 0.50 chance
Framework Bug	[HADOOP-1036] Hang at TaskTracker due to unhandled exception. Offending TaskTracker sends heartbeats although task has terminated.	[HANG-1036] Revert to older version and trigger bug by throwing NullPointerException
Framework Bug	[HADOOP-1152] Reducers at TaskTrackers hang due to race condition when file is deleted between rename and attempt to call getLength()	[HANG-1152] Simulate race by flagging renamed file as being flushed to disk and throw exceptions in filesystem code
Framework Bug	[HADOOP-2080] Reduces at TaskTrackers hang due to a miscalculated checksum.	[HANG-2080] Deliberately miscomputed checksum to trigger a hang at reducer

Table 1: Faults injected in our Hadoop experiments, and the reported real-world failures they simulate/reproduce. HADOOP-xxxx denotes a Hadoop bug-database ID.

6 Experimental Evaluation

6.1 Evaluation Approach and Criteria

Next, we evaluate the efficacy of using the *RAMS-FD* algorithm to detect failures in two classes of server systems: (i) the RuBiS online auction system, representing multi-tier web request processing systems, and (ii) Hadoop MapReduce, representing parallel distributed data-intensive programming systems. Our evaluation is based on faults we inject (§6.2.2, 6.3.2) that include both common problems and actual system bugs reproduced from their respective bug databases. We measured the efficacy of *RAMS-FD* at correctly indicting the node (host) with the injected fault across multiple experiment iterations. We used the following metrics to evaluate the efficacy of detection:

Precision and Recall are defined as:

$$Prec = \frac{TP}{TP + FP}; \qquad Recall = \frac{TP}{TP + FN}$$

In our setting, true-positives are nodes with injected faults that *RAMS-FD* correctly indicts[1], while false-negatives are nodes with injected faults that *RAMS-FD* does not indict, and

[1] i.e. Alarm raised when the faulty node is undergoing fault injection.

Category	[Source] Reported Failure	[Problem Name] Problem-Injection Methodology
HTTPD Bug	[HTTPD-41142] Endless loop in function that removes pool nodes from memory.	[spinlockinf_alldst_httpd] Intercept HTTPD and trigger infinite loop in function that removes pool nodes.
HTTPD Bug	[HTTPD-41644] Proxied requests pause for 5 seconds because sockets being created do not have the TCP_NODELAY option set.	[tcpnodelay_httpd] Intercept the `setsockopt` system call and disable the `TCP_NODELAY` option.
JBoss Bug	[JBOSS-994] Race-condition in connection pool manager causes JBoss to run out of connections.	[JBOSS-994] Revert to older version that omits a mutex when re-assigning released connections.
JBoss Bug	[JBOSS-1560] Background removal of expired passivated Session Beans causes deadlock.	[JBOSS-1560] Randomly delay session removal requests by 2-3 mins.
JBoss Bug	[JBOSS-2428] Lock contention when obtaining locks in the BeanLockManager degrades performance.	[JBOSS-2428] Randomly delay lock acquisition by 2-3minutes.
JBoss Bug	[JBoss users' mailing list, Jul 11, 2008] Infinite loop when passivating stateless session bean	[spinlockinf_putbid_jboss1] Intercept EJB that places bids in Rubis and trigger infinite loop.
MySQL Bug	[MYSQL-56405] Deadlock in metadata locking subsystem.	[pthreadhang_mysql] Emulate deadlock: Intercept pthread lock functions and return EBUSY signal.
Bad configuration	[HTTPD users' mailing list, Jul 23, 2010] Low number of connection pool threads causes web server to reject connections.	[lowpool_httpd] Set maximum size of connection pool to 10 for Apache.

Table 2: Faults injected in our Rubis experiments, and the real-world reported failures that they simulate or reproduce. JBOSS-xxx, MYSQL-xxx and HTTPD-xxx represent a JBoss, MySQL, or Apache HTTPD bug-database entry ID respectively.

false-positives are nodes without injected faults that *RAMS-FD* wrongly indicts[2]. Precision measures the probability that an alarm that is raised by the algorithm is a true fault, while recall measures the probability that all faults are identified. Precision and recall vary from 0.0 to 1.0, and a perfect algorithm achieves $Prec = Recall = 1.0$. There is an inherent trade-off between precision and recall: a perfect recall can be achieved by simply indicting all nodes, at the expense of precision. This trade-off for *RAMS-FD* can be adjusted by users by varying the diagnosis threshold T.

F_1 Score The F_1 score is the harmonic mean between precision and recall:

$$F_1 = \frac{2 \times Prec \times Recall}{Prec + Recall}$$

We use the F_1 score as an evaluation metric to summarize the overall trade-off between precision and recall across all thresholds, and we report the highest F_1 score achieved across all values of the threshold, T.

[2]This includes faulty nodes indicted when no fault is being injected.

To evaluate the diagnosis accuracy of *RAMS-DT*, we simply consider the accuracy of the diagnosis: the proportion of nodes being classified for which a label is correctly returned. This is because *RAMS-DT* diagnosis involves picking one of many labels for a node. For our evaluation, we used 100-fold cross-validation to evaluate the learned decision-trees.

6.2 Experimental Setup: Data-Intensive Processing

We conducted experiments on two Hadoop 0.18 clusters: a 6-node (5-slave, 1-master) cluster on our internal test-bed, which we will refer to as *FP*, and 11-, 26-, 51-, and 101-node (1-master, remaining slaves) clusters on Amazon's Elastic Compute Cloud (EC2) hosted virtualized service. We collected per-second sampled values of OS-collected performance counters from the `proc` filesystem for later analysis. These are the configurations of the two test-beds:

FP: Nodes had AMD Opteron 1220 dual-core CPU, 4GB of memory, and a dedicated 320 GB harddisk for Hadoop storage, Gigabit Ethernet, and ran amd64 Debian/GNU Linux 4.0 without virtualization.

EC2: Nodes were "extra-large" instances with the equivalent of 4 dual-core Intel Xeon-class CPUs, 15GB RAM, with 1.6 TB of Elastic Block Store (EBS) storage. Each node is a virtual machine running Debian/GNU Linux 4.0, hosted by EC2 using the Xen hypervisor (we had no access to the hypervisor).

6.2.1 Workloads

We tested the efficacy of *RAMS* diagnosis on two workloads: (1) the Nutch distributed web-crawler, which is a series of Hadoop MapReduce jobs, represents a workload deployed in production settings, and (2) GridMix, a realistic benchmark developed by Yahoo! to represent a comprehensive mix of data-intensive workloads as seen at Yahoo!'s Hadoop clusters. GridMix consists of randomly interleaved jobs of the following kinds: sorting of numerical and textual data, sampling from a large reference data-set, and a pipelined three-stage MapReduce job modeling complex multi-stage workloads. We ran Nutch on the *FP* and *EC2* test-beds, and ran GridMix only on the *EC2* test-bed due to size limitations of our internal *FP* test-bed.

6.2.2 Fault Injection

We injected faults to induce partial failures, and evaluate if *RAMS* can indict the node with the injected fault. Our injected faults simulate or reintroduce reported faults and bugs reported in the Hadoop users' mailing list and the Hadoop bug database [**?**]. These faults, the methods used to inject them, and the actual bug or user problem they are based on, are listed in Table 1. They can be broadly classified as resource contention problems (CPUHog, DiskHog), framework bugs (HADOOP-1036,1152,2080–hangs in the Map, Reduce, and Reduce phases respectively), and network problems (Packet Loss).

6.2.3 Experiments

We injected all faults listed in Table 1 in the experiments with the GridMix workload on EC2. We then investigated the difference in the diagnosis efficacy of *RAMS-FD* on non-virtualized as compared to virtualized infrastructures by injecting a subset of the faults: the CPUHog, DiskHog, and 50% packet-loss faults, on the Nutch workload on the FP and EC2 test-beds respectively. We focused on the resource-related faults to focus on possible differences in resource behavior in virtualized versus non-virtualized environments. Each experiment consists of one of the two workloads (Nutch, GridMix), with one of the faults injected persistently on one of the slave nodes $500s$ into a $1200s$ experiment. We currently do not consider faults on the Hadoop/MapReduce master node, but we intend to study if the *RAMS-FD* hypothesis applies to the master node in future work.

6.3 Experimental Setup: Multi-Tier Web Request Processing

We conducted experiments on RuBiS 1.4.3 with the Apache HTTPD 2.2.6 web-server, three instances of the JBoss 3.2.8 J2EE application servers running in a round-robin load-balanced configuration, and the MySQL 5.1.3 database server. These servers ran on our internal `FP` cluster with the same configuration as above, on amd64 Debian/GNU Linux 4.0 without virtualization. The main goal of our RuBiS experiments was to evaluate the detection efficacy of *RAMS-FD* on a different class of server applications from MapReduce systems.

6.3.1 Workloads

We ran the RuBiS application with user sessions performing browse and bid/buy actions. Our workload consisted of a 15% bid, 85% browse combination, out of 1000 simultaneous client requests which we spawned using the client emulator included in RuBiS.

6.3.2 Fault Injection

We injected faults in each of the three tiers in RuBiS. We describe the faults, and list their names in parentheses. We injected 4 faults in the JBoss application server, 3 in the Apache HTTPD web-server, and 1 in the MySQL database server. Table 2 summarizes the fault injection methodology and the actual bugs or user problems that each injected fault simulates. Each experiment consisted of the same workload, with one of the faults listed above injected in each experiment.

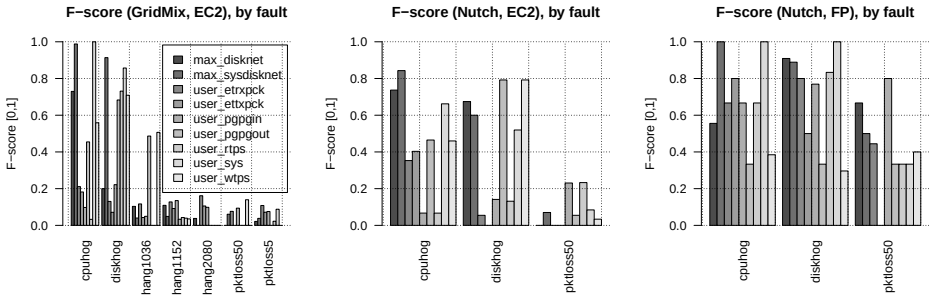


Figure 3: F-scores for *RAMS-FD* for all experiments for Hadoop MapReduce using correlations between user-space CPU utilization and all resource metrics.

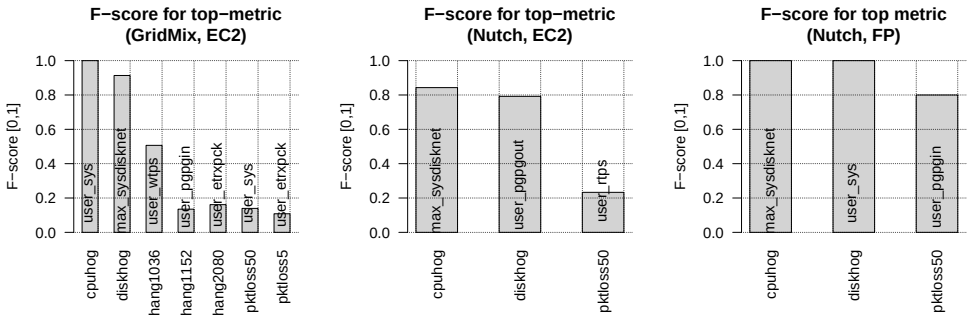


Figure 4: F-scores for *RAMS-FD* using the best correlation metric for all experiments for Hadoop MapReduce. The best correlation pair is listed for each fault.

7 *RAMS-FD* Results: Data-Intensive Processing

7.1 Summary of Results

First, we present overall results for *RAMS-FD* detection on three sets of experiments on Hadoop MapReduce: (i) on all injected faults on the GridMix workload (EC2 virtualized test-bed), (ii) on resource-related faults on the Nutch workload (EC2 test-bed), and (iii) on

resource-related faults on Nutch (FP non-virtualized test-bed). We present the F-scores, F_1, for *RAMS-FD* on each resource metric, for each of the Workload/Test-bed pairs in Figure 3.

RAMS-FD was able to detect the two resource hogs, CPUHog and DiskHog, strongly across both workloads in both clusters, with $F_1 \geq 0.8$ across all 3 sets of workload/test-bed combinations. This strong detection was achieved using the metrics $\rho_{U,DiskNetSys}$ for CPUHog on all 3 experiment setups, and for DiskHog on GridMix/EC2 and Nutch/FP, and using the metrics $\rho_{U,DW}, \rho_{U,MW}$ for DiskHog on Nutch/EC2. *RAMS-FD* was also able to detect the 50% packet-loss on Nutch/FP strongly, with $F_1 = 0.8$ for the $\rho_{U,MR}$ metric. *RAMS-FD* detected the HANG-1036 hangs in the Map phase moderately for the GridMix/EC2 workload/test-bed, with $F_1 \approx 0.5$ for the $\rho_{U,DW}$ metric. However, *RAMS-FD* was not effective at detecting the hangs in the Reduce phase, HANG-1152 and HANG-2080, nor the packet-losses, with $F_1 \approx 0.10$, on GridMix/EC2. *RAMS-FD* also detected the 50% packet-loss on Nutch/EC2 marginally better than on GridMix/EC2 with $F_1 > 0.2$, as compared to. This is because Nutch, being a web-crawler, is more network-oriented than GridMix.

The strong detection results for *RAMS-FD* on resource faults shows that *RAMS-FD* performs well at detecting resource issues and contentions, even when these contentions do not cause outright system crashes. However, *RAMS-FD* is relatively weaker at diagnosing the more subtle application-level bugs such as hangs in Maps and Reduces in Hadoop, and this was because these faults did not manifest strongly on system behavior. Finally, *RAMS-FD* is able to diagnose network problems more effectively as compared to our previous work [12, 13], which suffered from high false-positives in diagnosing packet-losses in Hadoop.

7.2 Virtualized vs. Non-virtualized

We compare the results of the Nutch/EC2 and Nutch/FP experiments to evaluate differences in performance of *RAMS-FD* on virtualized and non-virtualized environments. *RAMS-FD* strongly detects the CPUHog and DiskHog faults on both the EC2 and FP test-beds with $F_1 \geq 0.75$, although *RAMS-FD* does better on the non-virtualized FP test-bed. This is likely due to virtualization interferences e.g. guest-OS virtualization services. *RAMS-FD* was poor at diagnosing the 50% packet-loss on the virtualized EC2 test-bed, but was strong at diagnosing it on the FP test-bed with $F_1 = 0.8$. Diagnosing packet-losses on the virtualized test-bed is harder due to interferences from shared network resources. We plan to explore using additional virtualization-specific metrics to our correlations in future to better detect failures in virtualized settings.

7.3 Metric Choice

Figure 4 shows the F-scores achieved using the best-performing correlation metric for each fault in each set of Workload/Test-bed experiments by *RAMS-FD*. The correlation metric able to best diagnose each fault is useful for two reasons.

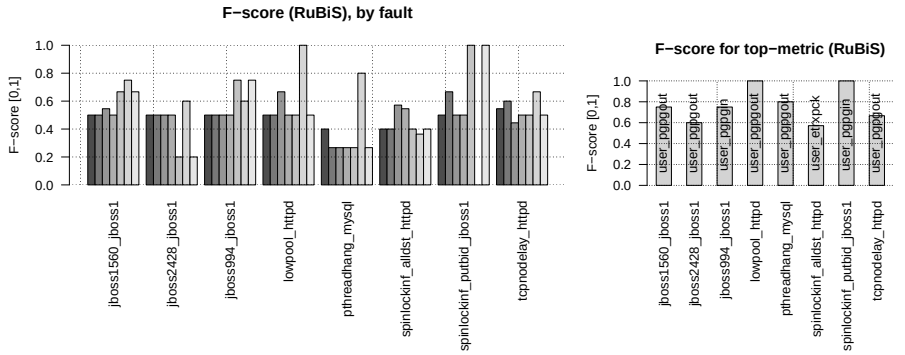
First, users can use the metric best at diagnosing most of the faults. From Figure 4, $\rho_{U,DiskNetSys}$ performs best: for faults with $F_1 \geq 0.5$, the correlation metric best detecting these faults is $\rho_{U,DiskNetSys}$. Hence, $\rho_{U,DiskNetSys}$ is be best suited for use in an *RAMS-FD* deployment, as it covers all resource categories. Users can first detect problems on the $\rho_{U,DiskNetSys}$ metric, then further reason about behavior using the remaining resource-specific metrics. Second, the correlation metric which causes *RAMS* to generate an alarm of a fault can be used to understand system behavior. For instance, compare DiskHog in Nutch/EC2 to the same fault in Nutch/FP. In the former, $\rho_{U,MW}$ best detected the fault, i.e. user-space CPU activity and memory pages written to disk were uncorrelated during the fault, as compared to fault-free operation. In the latter, $\rho_{U,K}$ best detected the fault, i.e. user-space CPU activity and kernel activity were uncorrelated during the fault. Hence, we can infer that the same fault manifests differently in virtualized versus non-virtualized infrastructures: in the former, memory contention is higher as compared to increased kernel activity as a result of the fault, whereas memory writes are much less affected in the non-virtualized case, with $\rho_{U,MW}$ yielding $F_1 < 0.4$ in the non-virtualized case. This demonstrates that **with the *RAMS* hypothesis**, it is easy and intuitive to reason about system behavior using our correlations-based detection metrics.

8 *RAMS-FD* Results: Multi-Tier Web Systems

8.1 Summary of Results

We present overall results using *RAMS-FD* to diagnose faults injected in RuBiS. Figure 5(a) shows the F-scores, F_1, for *RAMS-FD* using each of the resource metrics, for each fault we injected in RuBiS. *RAMS-FD* was able to diagnose all faults at least fairly, with $F_1 \geq 0.5$.

RAMS-FD detected 3 faults strongly with $F_1 \geq 0.8$: the spinlock infinite loop in the JBoss application server, the thread exhaustion fault in the web server (`lowpool_httpd`), and the thread hang in the MySQL server (`pthreadhang_mysql`). This was using the $\rho_{U,MR}$ metric in the first case, and using $\rho_{U,MW}$ in the latter two. Next, *RAMS-FD* detected 4 faults moderately well with $F_1 \geq 0.6$: the three JBoss application bugs, and the web server



(a) F-scores for RuBiS experiments

(b) F-scores using the best correlation metric.

Figure 5: F-scores for all experiments for RuBiS using correlations between user-space CPU utilization and all resource metrics, and for top metrics. (See Fig. 3 for legend.)

network misconfiguration (`tcpnodelay_httpd`). This was using $\rho_{U,MR}$ for the `JBOSS-994` JBoss hang, and $\rho_{U,MW}$ for the latter three faults. Finally, *RAMS-FD* detected the remaining faults with $F_1 \geq 0.5$, using the network-read metric $\rho_{U,NR}$ for the web server spinlock bug (`spinlockinf_alldst_httpd`).

The three faults with the strongest detection were infinite loops, which *RAMS-FD* can detect very strongly, while the application bugs, such as in JBoss, were moderately detected because these had less apparent effects than the infinite loop bugs. Finally, the last fault which was detected with only moderate success, was due to weak manifestations of the faults on system resources. The Apache web server was able to overcome a spinlock in one of its worker threads (`spinlockinf_alldst_httpd`) as it uses a pool of threads to service requests.

8.2 Metric Choice

Figure 5(b) shows the F-scores achieved using the best-performing correlation metric for each fault, together with the correlation metric. The best correlation metric was memory-related ($\rho_{U,MR}, \rho_{U,MW}$) in 7 of the 8 faults, and network-related in 1 of the faults ($\rho_{U,NR}$). The best detection metric for the RuBiS system is memory-related, in contrast to *RAMS-FD* detection on Hadoop, where $\rho_{U,DiskNetSys}$, which is balanced amongst the resource types, was the best detection metric. This is because the subsystems in the multi-tier web request processing system are predominantly event-driven processing systems which are less compute- and disk-intensive than Hadoop, which is data-intensive.

In addition, studying the faults in comparison to the correlation metric that best detected each fault, we see a relationship between the nature of the fault and the best diagnosis metric

for that fault. For instance, the network metrics $\rho_{U,NR}, \rho_{U,NW}$ best diagnosed the spinlock fault on the HTTPD web server–this suggests that the fault most severely affected the network behavior of the HTTPD server. Again, as with Hadoop, we can use the correlation metric that best detects each fault to infer the nature of the fault, showing how *RAMS-FD* can be useful in understanding the nature of the faulty system behavior.

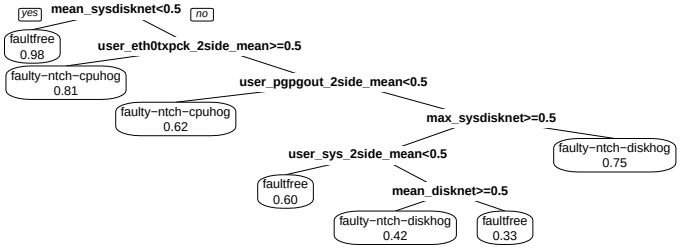


Figure 6: Decision-tree learned for the Nutch (on FP/EC2 test-beds) Hadoop workloads.

9 Results: *RAMS-DT* for Data-Intensive Processing

Next, we present results using *RAMS-DT* to obtain a diagnosis of the classification of the category of a given node, given the 9 alarm metrics generated by *RAMS-FD* for a given node in a system. We illustrate one learned decision-tree for the Nutch Hadoop workload across both the *EC2* and *FP* test-beds in Figure 8.2. The inner nodes of the tree show the *RAMS-FD* alarm to split on, and the leaf nodes show the highest occuring label among training instances in that node, and the proportion of that leaf's instance with that label. This proportion can serve as an approximate indicator of the confidence in the diagnosis. We observe that for faults such as PACKETLOSS-50 with low F_1, *RAMS-DT* is unable to diagnose nodes with such a fault (does not appear in any leaf node). Hence, *RAMS-DT* summarizes what *RAMS-FD* is able to detect; nonetheless, it provides a means to provide a diagnosis among the different kinds of failures *RAMS-FD* has previously detected, which have been (manually) labeled.

Workload	Label returned	Accuracy	Workload	Label returned	Accuracy
Nutch	Fault-free	97.9%	GridMix	Fault-free	97.1%
Nutch	CPUHog	68.5%	GridMix	CPUHog	73.6%
Nutch	DiskHog	22.9%	GridMix	DiskHog	93.9%

Table 3: Accuracy of decision-tree classification; faults that could not be diagnosed were omitted.

Evaluation We present the results of *RAMS-DT* diagnosis on our Hadoop experiments (§7) as a case-study for *RAMS-DT*. To evaluate the performance of *RAMS-DT*, we performed a 100-fold cross-validation of decision-trees learned separately from Nutch and GridMix data, and evaluate the accuracy of the diagnosis of the test-sets. Table 3 summarizes the accuracies of diagnosis for each label on a leaf-node in each tree. Faults for which *RAMS-FD* obtained $F_1 < 0.5$ were not diagnosable by *RAMS-DT*. We found that *RAMS-DT* was strong at diagnosing faults that *RAMS-FD* could detect in GridMix, with accuracies ranging from 73% to 97%, while faults in Nutch were diagnosable, but with considerably less accuracy. This was due to misdiagnosis of CPUHog faults as DiskHog faults, because of the similarity of the manifestation of these two faults on *RAMS-FD* alarms. Hence, *RAMS-DT* is susceptible to confusion when failures manifest similarly on system metrics.

10 Summary and Future Work

We have presented *RAMS*, a hypothesis on system behavior under fault-free conditions, the resulting *RAMS-FD* algorithm for detecting failures in server applications, and the *RAMS-DT* algorithm for considering *RAMS-FD* alarms collectively using decision-trees to diagnose the root cause of a failure.

We plan to evaluate the efficacy of using *RAMS-FD* to detect failures in the Hadoop MapReduce data-intensive processing system, and in multi-tier web request processing systems, on the RuBiS online auction benchmark. We plan to investigate whether *RAMS-FD* is able to detect resource contention faults on Hadoop, and is able to detect application exceptions, server bugs, and hang faults on RuBiS. In addition, we plan to evaluate whether the results we achieve using correlation based metrics are intuitively interpretable to sysadmins, and can overcome stigmas about how automated diagnosis techniques are too complex to use. We also plan to demonstrate that the *RAMS-FD* is versatile by evaluating it across two systems. Finally, we plan to evaluate whether the *RAMS-DT* companion algorithm to *RAMS-FD* can take advantage of the multiple detection metrics to generate a root cause of a fault or a security attack that has been previously observed.

In future, we also plan to validate *RAMS* on more workloads and target systems, and to implement *RAMS* as a true black-box diagnosis tool which can simply analyze OS-level metrics to perform diagnosis transparently to the application. We also plan to explore augmenting the *RAMS* algorithm with multiple diagnosis windows and exponentially weighted samples to detect faults that may be less persistent, such as application bugs.

References

[1] E. Thereska, A. Ailamaki, G. Ganger, and D. Narayanan, "Observer: keeping system models from becoming obsolete," in *Second Workshop on HotTopics in Automatic Computing (HotAC II)*, Jacksonville, FL, June 2007.

[2] R. R. Sambasivan, A. X. Zheng, E. Thereska, and G. Ganger, "Categorizing and differencing system behaviours," in *Second Workshop on HotTopics in Automatic Computing (HotAC II)*, Jacksonville, FL, June 2007.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec 2004, pp. 137–150.

[4] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *IEEE Conference on Dependable Systems and Networks*, Bethesda, MD, Jun 2002.

[5] E. Kiciman and A. Fox, "Detecting application-level failures in component-based internet services," *IEEE Trans. on Neural Networks: Special Issue on Adaptive Learning Systems in Communication Networks*, vol. 16, no. 5, pp. 1027– 1041, Sep 2005.

[6] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed system of black boxes," in *ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct 2003, pp. 74–89.

[7] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for request extraction and workload modelling," in *USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec 2004.

[8] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. Ganger, "Stardust: tracking activity in a distributed storage system," *SIGMETRICS Perform. Eval. Rev.*, vol. 34, no. 1, pp. 3–14, 2006.

[9] G. Khanna, I. Laguna, F. A. Arshad, and S. Bagchi, "Distributed diagnosis of failures in a three tier e-commerce system," in *IEEE Symposium on Reliable Distributed Systems (SRDS)*, Beijing, China, Oct 2007.

[10] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," in *ACM Symposium on Operating Systems Principles*, Brighton, United Kingdom, Oct 2005, pp. 105–118.

[11] P. Bodik, M. Goldszmidt, A. Fox, D. Woodard, and H. Andersen, "Fingerprinting the Datacenter: Automated Classification of Performance Crises," in *EuroSys*, Paris, France, Apr 2010.

[12] J. Tan, X. Pan, S. Kavulya, E. Marinelli, R. Gandhi, and P. Narasimhan, "Kahuna: Problem Diagnosis for MapReduce-based Cloud Computing Environments," in *12th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Osaka, Japan, Apr 2010.

[13] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Ganesha: Black-Box Diagnosis of MapReduce Systems," in *Workshop on Hot Topics in Measurement & Modeling of Computer Systems (HotMetrics)*, Seattle, WA, Jun 2009.

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

[15] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan, "Online detection of utility cloud anomalies using metric distributions," in *Network Operations and Management Symposium (NOMS)*, Osaka, Japan, Apr 2010.

[16] H. Kang, H. Chen, and G. Jiang, "PeerWatch: A fault detection and diagnosis tool for virtualized consolidated systems," in *International Conference on Autonomic Computing (ICAC)*, Washington D.C., USA, Jun 2010.

[17] S. Bhatia, A. Kumar, M. Fiuczynski, and L. Peterson, "Lightweight, High-Resolution Monitoring for Troubleshooting Production Systems," in *Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec 2008.

[18] G. Jiang, H. Chen, K. Yoshihira, and A. Saxena, "Ranking the importance of alerts for problem determination in large computer systems," in *International Conference on Autonomic Computing, ICAC*, Barcelona, Spain, June 2009, pp. 3–12.

[19] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. S. Ward, "System monitoring with metric-correlation models: problems and solutions," in *International Conference on Autonomic Computing, ICAC*, Barcelona, Spain, June 2009, pp. 13–22.

[20] L. Cherkasova, K. M. Ozonat, N. Mi, J. Symons, and E. Smirni, "Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change," in *IEEE Conference on Dependable Systems and Networks*, Anchorage, Alaska, June 2008, pp. 452–461.

[21] C. Stewart, T. Kelly, and A. Zhang, "Exploiting nonstationarity for performance prediction," in *EuroSys*, Lisbon, Portugal, Mar 2007.

[22] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer, "Failure diagnosis using decision trees," in *International Conference on Autonomic Computing*, New York, NY, May 2004, pp. 36–43.

[23] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, "The Blind Men and the Elephant: Piecing Together Hadoop for Diagnosis," in *IEEE International Symposium on Software Reliability Engineering (ISSRE), Industrial Track*, Mysuru, India, Nov 2009.

[24] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Detecting large-scale system problems by mining console logs," in *Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct 2009.

[25] J. Lou, Q. Fu, Y. Wang, and J. Li, "Mining dependency in distributed systems through unstructured log analysis," in *2nd USENIX Workshop on Analysis of System Logs (WASL)*, Big Sky, MT, Oct 2009.

[26] J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Visual, Log-Based Causal Tracing for Performance Debugging of MapReduce Systems," in *International Conference on Distributed Computing Systems (ICDCS)*, Genoa, Italy, Jun 2010.

[27] S. Ghemawat, H. Gobioff, and S. Leung, "The Google file system." in *ACM Symposium on Operating Systems Principles*, Lake George, NY, Oct 2003, pp. 29 – 43.

[28] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel, "Performance comparison of middleware architectures for generating dynamic web content," in *ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, Jun 2003.