Light-weight Black-box Failure Detection for Distributed Systems

Jiaqi Tan Soila Kavulya Rajeev Gandhi Priya Narasimhan
Electrical and Computer Engineering Department, Carnegie Mellon University
tanjiaqi@cmu.edu, spertet@ece.cmu.edu, rgandhi@ece.cmu.edu, priya@cs.cmu.edu

Abstract

Detecting failures in distributed systems is challenging, as modern datacenters run a variety of applications. Current techniques for detecting failures often require training, have limited scalability, or have results that are hard to interpret. We present *LFD*, a light-weight technique to quickly detect performance problems in distributed systems using only correlations of OS metrics. *LFD* is based on our hypothesis of server application behavior, does not require training, and detects failures with complexity linear in the number of nodes, with results that are interpretable by sysadmins. We further show that *LFD* is versatile, and can diagnose faults in Hadoop MapReduce systems and on multi-tier web request systems, and show how *LFD* is intuitive to sysadmins.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Reliability, availability and serviceability

General Terms Failure detection

Keywords MapReduce, Web Applications, Diagnosis

1. Introduction

Detecting failures in distributed systems is challenging: system metrics grow and increase in complexity as systems grow, and as interactions between components becomes more complex. Past techniques in failure detection and diagnosis have used application-specific information e.g. request paths, to detect anomalies [6, 13] although such data is often obtained only with invasive instrumentation and/or high overheads; thus they are infeasible on production systems. Others have used application-agnostic system metrics [4, 8, 15, 19], but they reason about data from all nodes in the system at once, which may not scale to large systems. In addition, these techniques may reason about data in complex ways e.g. learning complex patterns [8, 15], making it hard to extrapolate from the diagnoses of these algorithms to the actual system behavior to aid in investigating failures.

In contrast, we propose LFD (Light-weight Failure Detection), which targets light-weight failure detection which: (i) uses non-invasive, cheap (i.e. low-overhead) instrumentation, (ii) is scalable to large systems with many nodes, and (iii) is intuitively understood, to help sysadmins gain insight into system behaviour from the detection results. LFD provides first-pass alarms to notify sysadmins of failures, and to provide them with quick insight into system behaviour, letting them utilize more sophisticated techniques (with overheads that are too high to be always on) to further isolate the failure. LFD is based on our hypothesized model (§ 2.1) of fault-free system behavior of server applications. LFD is cheap, with instrumentation overhead of $< 1\%$, and non-invasive, using only common OS performance counters. It is scalable, using only simple computations and local information on each node. We have evaluated LFD on a multi-tier web request system, and Hadoop [2] MapReduce [9].

1.1 Goals and Non-goals

Cheap, Transparent: We aim to use metrics that can be collected with minimum overhead and transparently (without modifying target applications). This would enable our algorithms to be used in production settings, where invasive or high-overhead instrumentation are infeasible.
Scalable: Our algorithms must have low computational complexity to enable them to scale up to large systems. We aim for *LFD* to require only information from a single node to diagnose that node, so that detection can be decentralized.
Versatile: We aim to detect failures in multiple types of systems. This would enable our algorithm to be used in complex modern datacenters running multiple types of applications.
Effective Diagnosis: We aim to be effective in detecting failures in our target systems–we aim to minimize false-positives while detecting as many failures as possible.
Non-goals: In this work, we do not present an online implementation of the LFD algorithms. Instead, we focus on presenting and evaluating the detection of our algorithms.

1.2 Fault Model and Assumptions

LFD targets performance problems: faults resulting in a slowdown, causing the processing of tasks in the system to

MBDS '12 September 21, 2012, San Jose, CA, USA

take a longer time than without the fault, causing increased runtime and reduced system throughput. We do not consider crashes as these can be detected using simpler methods e.g. heartbeats. Nonetheless, we envision that LFD can be used to detect crashes before the fault has resulted in a crash. We do not target value faults i.e. execution with incorrect results.

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.

1.3 Target Systems

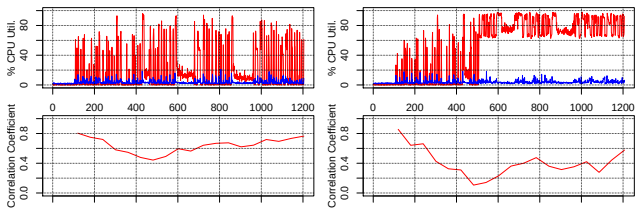
Data-Intensive Processing MapReduce [9] is a popular framework for distributed parallel processing. Jobs described as a Map and a Reduce function are parallelized by running multiple copies of Map and Reduce tasks, each on a different segment of a large data-set. Hadoop is an open-source Java implementation of MapReduce. Hadoop has a master-slave architecture (one master, multiple slave nodes), with the master using heartbeats to detect slave crashes and restart tasks. We currently only target faults on slave nodes.

Multi-Tier Web Request Processing RuBiS [5] is an auction website benchmark. RuBiS has a three-tier architecture: we used the Apache web server, JBoss J2EE application server, and the MySQL database. Our setup comprised 1 web server, 3 JBoss servers in round-robin load-balanced mode, and a single MySQL database server.

2. Approach and Implementation

2.1 The LFD Hypothesis

The LFD hypothesis proposes that at a conceptual level, normal (fault-free) processing to service a user request in a server 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, or slave nodes receiving instructions from a MapReduce master node), reads data from disk, or writes results to network or disk, and (ii) a **compute** phase, when the application performs operations based on received inputs/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 system resources: disk, network, or kernel-space CPU activity to service disk and network operations. Hence, application activity alternates between these two phases at a micro-level (in time-scales of micro- to milli-seconds, at the granularity of one thread of execution). Then, we hypothesize that this alternating activity induces correlated behavior at the macro-level between user-space CPU activity and system resource consumption.



(a) Normal node (b) Faulty node

Figure 1. Time-series of User-space CPU% (red), Kernel-space CPU% (blue) (above), and the correlation between the two metrics (below), in one experiment. CPU hog injected on 1 node in a 5-node Hadoop cluster. Correlation falls when the fault is injected at 500s.

Further, we hypothesize that in the presence of failures, at the micro-level, user-space **compute** activity would show a marked change in its relationship with the indicators of resources used in the **communications** phase (i.e. disk, network, kernel-space CPU activity). Either processing activity slows down, and the **compute** phase dominates the **communications** phase, e.g. 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, or 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.

We illustrate the intuition behind the LFD hypothesis. Fig. 1 shows a trace of the user-space CPU%, U, and the resource metric, kernel-space CPU%, K, on two slave nodes in the same Hadoop cluster. We inject an external CPU load to consume 70% of CPU utilization to simulate a fault on one node. Fig. 1(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 1(a) shows the trace of a fault-free node, where U and K exhibit similar behavior. The Pearson's correlation coefficient $\rho_{U,K}$ for user- and kernel-space CPU% between the faulty and fault-free nodes remains higher for the fault-free node as compared to the faulty node, and $\rho_{U,K}$ falls significantly after the fault is injected. This supports the LFD hypothesis.

2.2 *LFD*: Failure Detection Algorithm

Based on our hypothesis, the *LFD* algorithm computes the Pearson's correlation coefficient[1] (ρ) between user-space CPU% and a resource metric (disk, network, kernel-space CPU%) used in the **communications** phase, and raises an alarm indicating a failure when the coefficient falls below

[1] We chose to use the Pearson's coefficient for the simplicity of its computation, allowing our algorithm to be implemented in a lightweight manner. We intend to explore the use of other measures of correlation in future work.

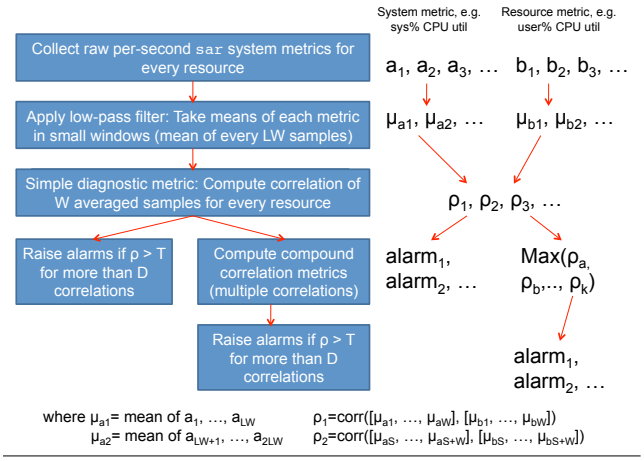


Figure 2. Summary of LFD algorithm

a threshold, T. This computation is done independently for the metrics collected on each node being diagnosed, and for each system resource. $\rho \in [-1.0, 1.0]$, and 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% is:

$$\begin{aligned} \rho_{U,K} &= \frac{covariance(U,K)}{\sigma_U \sigma_K} \\ \text{where } U &= \text{time-series of user-space CPU\%} \\ K &= \text{time-series of kernel-space CPU\%} \end{aligned}$$

In total, we compute ρ between user-space CPU%, U, and 7 resource-related metrics, for each node. These raw metrics are sampled from `/proc` using the `sar` tool once per second:

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

We refer to each of these 7 correlations as *LFD* detection metrics. Further, we create compound measures of the conformance of the system's behavior to the LFD hypothesis, which take the maximum correlation between U and multiple metrics. This allows a combination of activity in multiple resource categories (disk, network, kernel-space activity) in time (e.g. user activity U may be strongly correlated with disk metrics at one point and network metrics at another).

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

To reduce the noise from sampled parameters, we take the means of metrics in small windows ($< 10s$) before computing the correlation coefficient ρ. We raise an alarm when the mean of D consecutive values of ρ falls below threshold T; this is to remove spurious correlations not indicative of true processing. For each node (host) in a system under detection, the algorithm takes the time-series's of user-space CPU% and of 1 of the 7 resource metrics (and 2 compound metrics), and outputs a list of times when ρ has threshold violations. For a distributed system with N nodes, N lists of violation times are generated. *LFD* does not need training.

LW, Low-pass Window Width	Raw samples to take mean of
LS, Low-pass Window Slide	Raw samples to slide low-pass filter
W, Correlation Window Width	Samples correlate
S, Correlation Window Slide	Samples to slide detection window by
D, Diagnosis Window	Correlation coefficients to consider

Table 1. Tunable parameters in LFD algorithm

Algorithm Parameters There are 5 tunable parameters in the *LFD* algorithm. The detection latency (i.e. minimum time between when alarms can be raised) in the steady state is $LS \times S$, while the information considered in each alarm raised is $LW \times W$. Finally, threshold T is a run-time parameter that is user-tuned according to whether he prefers more alarms to be returned while suffering higher false-positives, e.g., when the sysadmin is busy, 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 time to do so. We used the following parameters in our evaluation: $LW = 5, LS = 5, W = 60, S = 12, D = 10$. This resulted in a detection latency of $LS \times S = 60$ seconds, and each result considered the last $LW \times W = 300$ seconds of system behaviour. We chose parameters to minimize detection latency, $LS \times S$. We found that $LS < 5$ gave poor detection performance, and likewise with $S < 12$. We chose LW, W to be sufficiently large to consider relevant system behavior, while not picking 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 used the same set of parameters for detecting faults in both RuBiS and Hadoop, showing the versatility of *LFD*.**

Complexity and Scalability As *LFD* uses only information from each node to compute alarms for that node, the detection computation can be carried out independently on each node. This saves the bandwidth for transmitting metrics to a central location, and the complexity of the system-wide failure detection computation is constant in the size of the distributed system. In addition, *LFD* computes the Pearson's correlation coefficient, which can be computed simply.

Interpreting LFD Alarms The LFD hypothesis proposes that fault-free system behavior is associated with a correlation between *computation* and *communication*, with the communication marked by activity in one or more resource metrics. Hence, an alarm raised by the LFD detection algorithm indicates a lack of correlation between user-level computation, and resource activity. For instance, an alarm in $\rho_{U,DW}$ simply means there is too much processing and too

[Source] Reported Failure	[Problem Name] Problem-Injection Methodology
[Mailing list, 9/13/07] CPU bottleneck as master, slave daemons ran on same host	[CPUHog] Run CPU-intensive task to consume 70% CPU util.
[Mailing list, 9/26/07] Excessive messages logged to file during startup	[DiskHog] Sequential disk workload wrote 20GB to filesystem
[HADOOP-2956] Degraded network leads to long DataNode block transfer times	[PacketLoss5/50] Drop packets with $0.05, 0.50$ chance
[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
[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
[HADOOP-2080] Reduces at TaskTrackers hang due to a miscalculated checksum.	[HANG-2080] Miscomputed checksum to trigger reducer hang

Table 2. Hadoop faults injected, and the real-world failures simulated/reproduced. HADOOP-XX = Hadoop bug-database ID.

[Source] Reported Failure	[Problem Name] Problem-Injection Methodology
[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-41644] Proxied requests pause for 5 seconds as sockets being created do not have TCP_NODELAY option set.	[tcpnodelay_httpd] Intercept the `setsockopt` system call and disable the `TCP_NODELAY` option.
[JBOSS-994] Race-condition in connection pool manager: JBoss runs out of connections.	[JBOSS-994] Revert to older version that omits a mutex when reassigning released connections.
[JBOSS-1560] Background removal of expired passivated Session Beans causes deadlock.	[JBOSS-1560] Randomly delay session removal by 2-3 mins.
[JBOSS-2428] Lock contention in BeanLockManager degrades performance.	[JBOSS-2428] Randomly delay lock acquisition by 2-3minutes.
[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-56405] Deadlock in metadata locking subsystem.	[pthreadhang_mysql] Emulate deadlock: Intercept pthread lock functions and return EBUSY signal.
[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 3. 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.

few disk writes, or too many disk writes and too little processing. In addition, the multiple resource categories covered by the LFD detection metrics enables initial indictment of a "culprit" system resource, helping sysadmins zoom in on the problem. We discuss metric choice and result interpretation further in the extended version of this paper [18].

3. Experimental Evaluation

Evaluation Approach and Criteria Next, we evaluate the efficacy of using the *LFD* algorithm to detect failures in two classes of server systems: (i) the RuBiS online auction system, representing multi-tier web systems, and (ii) Hadoop MapReduce, representing data-intensive processing systems. Our evaluation is based on faults we inject (§3) that comprise common problems and actual bugs reproduced from bug databases. We measured the efficacy of *LFD* at correctly indicting the node (host) with the injected fault across multiple experiment iterations. We evaluate the efficacy of detection using F_1 (harmonic mean of the precision and recall of detection).

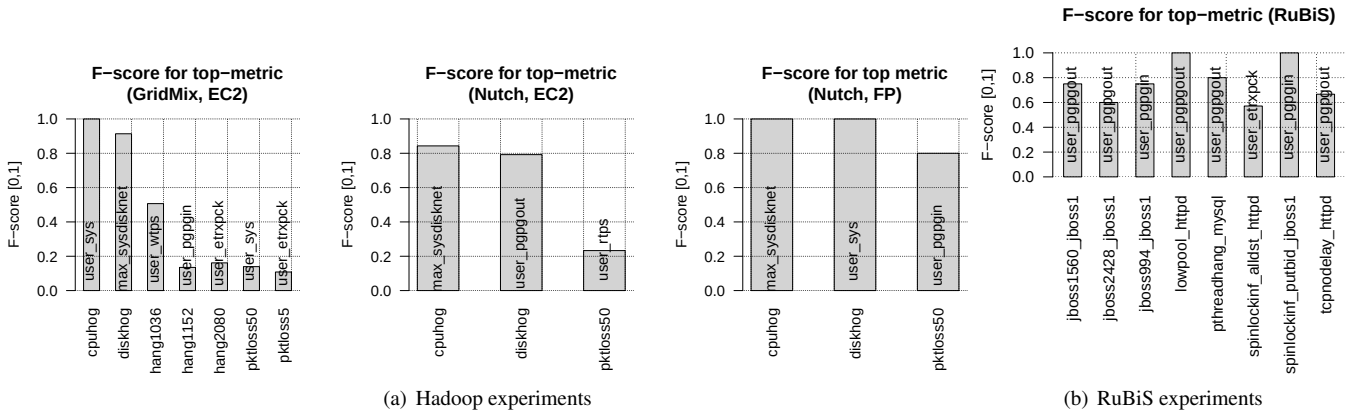
Experimental Setup We conducted experiments on two Hadoop 0.18 clusters: a 6-node (5-slave, 1-master) cluster on our internal test-bed (FP), and 11-, 26-, 51-, and 101-node (1-master, remaining slaves) clusters on Amazon's Elastic Compute Cloud (EC2) hosted virtualized service. We collected per-second sampled OS-collected performance counters from `/proc` for later analysis. Our two test-beds (running Debian Linux 4.0) are:

FP: AMD Opteron 1220 dual-core CPU, 4GB memory, dedicated 320 GB harddisk for Hadoop storage, Gigabit Ethernet, without virtualization.

EC2: "Extra-large" instances: 4 dual-core Intel Xeon-class CPUs, 15GB RAM, 1.6 TB of Elastic Block Store storage. Each node is a virtual machine, hosted by EC2 using Xen.

We conducted experiments on RuBiS 1.4.3 with Apache 2.2.6, three instances of the JBoss 3.2.8 J2EE app-servers running load-balanced round-robin, and the MySQL 5.1.3 database server. These servers ran on our internal FP cluster with the above configuration without virtualization.

Workloads We tested the efficacy of *LFD* detection on two Hadoop 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 benchmark developed by Yahoo! to represent a comprehensive mix of data-intensive workloads as seen at their clusters. 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. We ran the RuBiS application with user sessions performing browse and bid/buy actions. Our RuBiS 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.



(a) Hadoop experiments (b) RuBiS experiments

Figure 3. F-scores for *LFD* for all experiments. The best correlation metric is listed for each fault.

Fault Injection We injected faults in Hadoop to induce partial failures, and evaluate if LFD can indict the node with the injected fault. Our injected faults simulate or reintroduce bugs reported in the Hadoop users' mailing list and bug database [1]. These faults, the methods used to inject them, and the actual bug or user problem they are based on, are listed in Table 2. 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). For RuBiS, we injected faults in each of the three tiers. We injected 4 faults in the JBoss application server, 3 in the Apache web-server, and 1 in the MySQL database server. Table 3 summarizes the injection methodology and actual bugs/user problems that each injected fault simulates.

4. Results and Discussion

Summary: Data-Intensive Processing We present overall results for *LFD* 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 *LFD* on each resource metric and Workload/Test-bed pairs in Figure 3(a).

LFD detected the two resource hogs, CPUHog and DiskHog, strongly on both workloads in both clusters ($F_1 \geq 0.8$). This strong detection was achieved using the metrics $\rho_{U,DiskNetSys}$ for CPUHog on all 3 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. *LFD* also detected the 50% packet-loss on Nutch/FP strongly, ($F_1 = 0.8$ for $\rho_{U,MR}$). *LFD* detected the HANG-1036 hangs in the Map phase moderately for the GridMix/EC2 workload/test-bed, ($F_1 \approx 0.5$ for $\rho_{U,DW}$). However, *LFD* was not effective at detecting the hangs in the Reduce phase, HANG-1152 and HANG-2080, nor packet-losses, with $F_1 \approx 0.10$, on GridMix/EC2. *LFD* detected the 50% packet-loss on Nutch/EC2 marginally better than on GridMix/EC2 ($F_1 > 0.2$) as compared to GridMix. This is because Nutch, a web-crawler, is more network-oriented than GridMix. The strong results for *LFD* on resource faults shows that *LFD* detects resource issues and contentions well, even when these contentions do not cause outright system crashes. However, *LFD* is weak at detecting more subtle application-level bugs e.g. hangs in Maps and Reduces in Hadoop, as these faults did not manifest strongly in system behavior.

Summary: Multi-Tier Web Systems We present results using *LFD* to detect faults injected in RuBiS. Figure 3(b) shows the F-scores, for *LFD* using the best metric, for each fault we injected in RuBiS. *LFD* was able to detect all faults at least fairly with $F_1 \geq 0.5$. *LFD* detected 3 faults strongly ($F_1 \geq 0.8$): the spinlock infinite loop in JBoss (`spinlockinf_putbid_jboss1`), 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, *LFD* detected 4 faults moderately with $F_1 \geq 0.6$: the three JBoss application bugs, and the web server 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. *LFD* detected the remaining fault with $F_1 \geq 0.5$, using the network-read metric $\rho_{U,NR}$ for the web server spinlock bug. The three faults with the strongest detection were infinite loops, which *LFD* detected very strongly, while the application bugs, such as in JBoss, were moderately detected as these had less apparent effects than the infinite loop bugs. Finally, `spinlockinf_alldst_httpd` was detected only moderately well as Apache could overcome a spinlock in one of its worker threads as it uses a thread pool.

5. Related Work

Machine learning has been commonly used for failure diagnosis in distributed systems [4, 6, 8, 13, 15, 19]. Given

knowledge of failed requests (e.g. SLO violations), supervised learning techniques localize failures to their originating node/metric [6, 8, 13] based on learned models. *LFD*'s detection does not require training as it is based on the LFD hypothesis. 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 [4, 12, 20]. This is useful for systems with long running jobs or novel user-programmable workloads, e.g. MapReduce, or in partial failures, when the problem has not escalated into an SLO violation. LFD is in this class of techniques, and solves both instances of the problem–for Hadoop and multi-tier web systems. Like LFD, [4] uses OS metrics, but focuses on macro datacenter state, while LFD detects failures on individual nodes. *LFD* is more scalable than [4] as a failure detector as it uses only local state for each node. Similarly, [20] uses metrics from all nodes, while *LFD* uses only node-local metrics for detection. [12] also uses correlations, but they correlate behavior across nodes, while LFD correlates behavior between metrics on the same node, which is more scalable. [3] focused on cheaply collecting system metrics for reconstruction of past incidents for investigation. Their correlation-like "rules" for inferring system problems are not general, unlike the LFD hypothesis. Some diagnostic approaches have used regression to automatically discover correlations between metric pairs [10, 11]. However, they do not scale well to large numbers of nodes/metrics as they search for metric correlations locally and remotely between nodes. *LFD* exploits semantic knowledge to analyze a small number of metrics, providing scalable detection for large-scale systems. [7] and [16] used 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. *LFD* uses only OS metrics to detect problems allowing it to be easily ported across systems. Some techniques have diagnosed failures using white-box information from Hadoop's natively generated logs [14, 17, 21], rather than black-box OS-level metrics that LFD uses. [15, 19] used OS metrics to diagnose failures in Hadoop, using a peer-comparison approach with $O(n^2)$ complexity. LFD is more scalable, and has wider applicability to multi-tier web systems.

6. Conclusion and Future Work

We have presented LFD, a hypothesis on system behavior under fault-free conditions, and the resulting *LFD* algorithm for detecting failures in server applications. We have shown that *LFD* can detect resource contention faults on Hadoop, and can detect application exceptions, server bugs, and hang faults on RuBiS. We also showed how using correlation based metrics, together with our LFD hypothesis, gives us results which are intuitively interpretable to sysadmins. We also demonstrated the versatility of *LFD* in working across two systems. In future, we plan to validate LFD on more workloads and target systems.

Acknowledgements This research was sponsored in part by the project CMUPT/RNQ/0015/2009 (TRONE), and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC).

References

[1] `https://issues.apache.org/jira`.

[2] `http://hadoop.apache.org/core`.

[3] S. Bhatia, A. Kumar, M. Fiuczynski, and L. Peterson. Lightweight, High-Resolution Monitoring for Troubleshooting Production Systems. In *OSDI*, Dec 2008.

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

[5] 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 Middleware*, Jun 2003.

[6] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN*, Jun 2002.

[7] 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 *DSN*, June 2008.

[8] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP*, Oct 2005.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, Dec 2004.

[10] G. Jiang, H. Chen, K. Yoshihira, and A. Saxena. Ranking the importance of alerts for problem determination in large computer systems. In *ICAC*, June 2009.

[11] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. S. Ward. System monitoring with metric-correlation models: problems and solutions. In *ICAC*, June 2009.

[12] H. Kang, H. Chen, and G. Jiang. PeerWatch: A fault detection and diagnosis tool for virtualized consolidated systems. In *ICAC*, Jun 2010.

[13] E. Kiciman and A. Fox. Detecting application-level failures in component-based internet services. *IEEE Trans. on Neural Networks: Special Issue on Adaptive Learning Systems in Communication Networks*, 16(5):1027– 1041, Sep 2005.

[14] J. Lou, Q. Fu, Y. Wang, and J. Li. Mining dependency in distributed systems through unstructured log analysis. In *USENIX WASL*, Oct 2009.

[15] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Ganesha: Black-Box Diagnosis of MapReduce Systems. In *HotMetrics*, Jun 2009.

[16] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. In *EuroSys*, Mar 2007.

[17] J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Visual, Log-Based Causal Tracing for Performance Debugging of MapReduce Systems. In *ICDCS*, Jun 2010.

[18] J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Lightweight Black-box Failure Detection for Distributed Systems. CMU-PDL-12-106, Jul 2012.

[19] J. Tan, X. Pan, S. Kavulya, E. Marinelli, R. Gandhi, and P. Narasimhan. Kahuna: Problem Diagnosis for MapReduce-based Cloud Computing Environments. In *IEEE/IFIP NOMS*, Apr 2010.

[20] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan. Online detection of utility cloud anomalies using metric distributions. In *IEEE NOMS*, Apr 2010.

[21] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*, Oct 2009.