2013 IEEE 33rd International Conference on Distributed Computing Systems

Diagnosing Data Center Behavior Flow by Flow

Ahsan Arefin†, Vishal K. Singh‡, Guofei Jiang‡, Yueping Zhang‡, Cristian Lumezanu‡
†University of Illinois at Urbana-Champaign, ‡NEC Laboratories America

***Abstract*—Multi-tenant data centers are complex environments, running thousands of applications that compete for the same infrastructure resources and whose behavior is guided by (sometimes) divergent configurations. Small workload changes or simple operator tasks may yield unpredictable results and lead to expensive failures and performance degradation. In this paper, we propose a holistic approach for detecting operational problems in data centers. Our framework, FlowDiff, collects information from *all* entities involved in the operation of a data center—applications, operators, and infrastructure—and continually builds behavioral models for the operation. By comparing current models with pre-computed, known-to-be-stable models, FlowDiff is able to detect many operational problems, ranging from host and network failures to unauthorized access. FlowDiff also identifies common system operations (*e.g.*, VM migration, software upgrades) to validate the behavior changes against planned operator tasks. We show that using passive measurements on control traffic from programmable switches to a centralized controller is sufficient to build strong behavior models; FlowDiff does not require active measurements or expensive server instrumentation. Our experimental results using NEC data center testbed, Amazon EC2, and simulations demonstrate that FlowDiff is effective and robust in detecting anomalous behavior. FlowDiff scales well with the number of applications running in the data center and their traffic volume.**

I. Introduction

Effective diagnosis of performance problems and abnormal behavior is essential for maintaining availability and controlling running costs in large-scale multi-tenant data center networks. When undetected in time, these problems have severe financial or availability implications. A recent study on 41 US-based data centers shows that every minute of outage costs the providers on average $5,600 [11]. Anecdotal evidence suggests that the April 2011 outage of their AWS service cost Amazon around 2 million in revenue loss [10].

The behavior of modern data centers is determined by actions and events at three different layers: the *applications* that run on the data center, the *operators* that manage it, and the network *infrastructure*. Interactions between layers can have unpredictable effects and lead to performance degradation or failures across the data center. For example, the aforementioned Amazon outage was caused by an unpredictable interaction between applications and operators: high-volume application traffic was mistakenly routed into the low-capacity internal network due to a routing misconfiguration [8].

Previous work has shown how to diagnose operational problems caused by unexpected interactions within the same layer (*e.g.*, between different applications [7], [2], [13], [28]). However, as applications become more and more complex, and place different requirements on the infrastructure or operators, it is necessary to consider interactions *between* layers as a potential source of failures. The goal of this paper is to explore a *holistic* approach for detecting operational problems in data centers, where information from applications, operators, and infrastructure comes together to help detect the problem.

We propose *FlowDiff*, an automated framework for scalable and accurate detection of operational problems in large-scale data centers. Two important decisions underline the design of FlowDiff. First, FlowDiff frequently *models the behavior of a data center using expressive signatures* from three perspectives: applications, infrastructure, and operators. To detect problems, it compares the current behavior with a previously computed, stable, and correct behavior. Second, FlowDiff *uses network flow information to model data center behavior* from all three perspectives. Since, capturing all network flows is expensive, FlowDiff uses passive measurements on control traffic from programmable switches to a centralized controller [25]. Our insight is that control traffic is sufficient to build reliable behavior models at little cost.

FlowDiff models the behavior of a data center using *infrastructure*, *application*, and *task* signatures. The infrastructure signature captures the physical topology of the network, the mapping of applications to servers, and baseline performance parameters (such as link utilization and end-to-end delay). Application signatures capture the behavior of each application (*e.g.*, response time distribution, flow statistics) and how applications interact with each other. Task signatures model the valid behavioral changes performed by the operator or by applications (*e.g.*, VM migration, attaching a new server). To automatically detect problems, FlowDiff compares the current behavior of the data center with known past good behavior in terms of application and infrastructure signatures. If it detects changes, it tries to explain them using known task signatures; only if no explanation exists, it reports a problem along with the involved physical components (*e.g.*, servers, switches, links). Note that, FlowDiff does not try to identify the root-cause of the problem, rather it provides debugging information to assist root-cause analyses.

To build correct data center behavior models that capture all interactions between applications, FlowDiff monitors the control traffic between the control and data planes of the network. This requires that the switches in the data center be programmable and offer an open control API (*e.g.*, OpenFlow [25]). Monitoring the control traffic instead of the data traffic is sufficient to detect performance-related behavioral changes (*e.g.*, network failures, performance degradation) because control traffic offers information about changes in data path performance. For example, in OpenFlow, changes in utilization of a link, such as the start or end of a flow,

1063-6927/13 $26.00 © 2013 IEEE
DOI 10.1109/ICDCS.2013.18

are announced through control messages (*e.g.*, `PacketIn` or `FlowRemoved`). In addition, the central controller can also poll flow counters on switches to learn utilization.

Our choice to design FlowDiff for flow-based data centers [14], [20], where switches are programmable and can be controlled from a centralized location, is motivated by the unprecedented scalability and visibility that such an architecture promises. Collecting measurements and building behavior models at a centralized controller eliminates the need for expensive instrumentation and allows the flexibility to choose between different levels of visibility to balance scalability in data collection and expressiveness of measurements. Notwithstanding our choice, the use of the techniques for problem diagnosis that we propose is not limited to flow-based data centers: our behavior modeling works with any scalable monitoring technique (*e.g.*, server logging) that can collect flow information to build expressive signatures.

We bring the following contributions. (1) We provide a holistic framework to compare data center behavior at two different points in time by identifying representative and comprehensive behavioral components that reflect all aspects of data center behavior: application, operator, and infrastructure. By comparing models captured at different points in time, FlowDiff is able to detect a wide range of problems—from application failures and unauthorized traffic to network congestion or failure—with high accuracy (Section V-A) and at reasonable scalability cost (Section V-C). (2) We build stable, robust, and expressive models for operator, infrastructure, or application changes using exclusively control traffic from programmable switches and without active measurements or server instrumentation (Section V-B).

II. Related work

Research work on diagnosing enterprises and distributed systems can be modeling-based or measurement-based. Modeling-based approaches express the operation of a system as a set of states and use model-checking [18], [32], [29] or static analysis [12], [24] to detect sequences of states that lead to bugs. While such proposals are effective in detecting bugs in system configuration and protocols (*e.g.*, reachability problems or loops), they cannot easily find operational problems caused by unexpected traffic workloads or by mistakes of the operators. FlowDiff specifically models known operator tasks and uses the models to filter alarms.

Measurement-based debugging techniques monitor the network traffic and derive models based on the current operation, configuration, and performance of the system. Sherlock [2], Orion [7], Magpie [3] and Macroscope [28] install agents in the network or at the end-hosts, and use measurements to build application dependency graphs (ADG) or canonical models that express how various applications or measurements depend on each other. They use ADGs or profiling to trace back the root cause of a user-reported problem. Another body of work relies on instrumenting applications to gather measurements and detect potential problems [13], [15] or on directly profiling the end-host stack [33]. FlowDiff also uses network flow

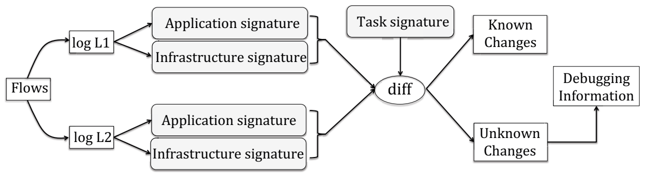


Fig. 1. Components of FlowDiff and their interactions.

measurements to profile the data center behavior but does so in a non-intrusive way by taking advantage of the constant flow of control messages between the switches and the external controller (*e.g.*, using the OpenFlow protocol). Unlike previous approaches, which investigate problems caused by interactions between entities at a single layer, FlowDiff considers interactions between all layers (applications, operators, and infrastructure) that affect the behavior of a data center.

Several approaches use performance measurements to model application behavior and employ statistical machine learning techniques to detect abnormalities. Cohen *et al.* [9] and Goldszmidt *et al.* [16] use a tree-augmented naive (TAN) bayesian network to learn the probabilistic relationship between SLA violations and system resource usage and to identify performance bottlenecks in the system. Bodik *et al.* [5] propose a fingerprinting approach to correlate key performance indicators with system SLA. Jiang *et al.* [22] developed an invariant-based method to profile large systems for management purposes. Unlike FlowDiff, none of these approaches considers application structure and interactions. Closer to our approach is Distalyzer [26], a tool that compares system behaviors extracted from execution logs and infers associations between system components and performance.

III. FlowDiff Design

FlowDiff consists of two major components: *modeling* and *diagnosing*. In the modeling phase, FlowDiff passively captures control traffic in flow-based networks and builds signatures for applications (to model application behavior), infrastructure (to characterize the properties of the underlying network), and operator tasks (to represent known operational tasks such as VM migration or data backup). During the diagnosing phase, FlowDiff identifies differences in application and infrastructure signatures captured at different times and validates them using the learned task signatures.

Figure 1 shows the components of FlowDiff and the interactions among them, where $L1$ and $L2$ are logs captured over two different time intervals. If $L1$ represents the behavior of the data center in a stable state and $L2$ represents the behavior when a problem is reported, FlowDiff uses a `diff` function between $L1$ and $L2$ to infer useful debugging information. In this section, we describe the signature modeling phase as well as how the measurement logs are collected. In Section IV, we describe how FlowDiff finds operational problems using `diff` in application and infrastructure signatures.

A. Measurement collection

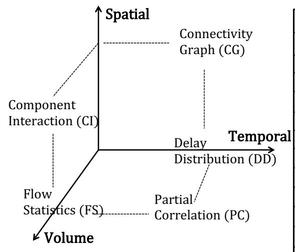
A flow-based network consists of programmable switches (data plane) managed by a logically centralized controller (control plane) using a control protocol, such as OpenFlow [25]. In a typical operation of an OpenFlow network (also called reactive deployment), the controller populates the switch flow tables with forwarding rules as follows. When the switch does not have a matching rule for an incoming flow, it notifies the controller via a `PacketIn` message. The controller uses a `FlowMod` message to install a flow entry on the switch. A flow entry is either a microflow (matches against a single flow) or contains a wildcard (matches against multiple flows). Each flow entry is associated with two timeouts that determine when the entry expires: a soft timeout, computed from the time of the last matched packet against the entry, and a hard timeout, counted from the time of the first matched packet. When the entry expires, the switch notifies the controller using a `FlowRemoved` message that contains, among others, the total number of bytes matched and the duration of the entry.

We capture `PacketIn`, `FlowMod`, and `FlowRemoved` messages at the controller and use them to build data center wide signatures. The granularity of measurements from a switch depends on three parameters: soft timeout, idle timeout, and flow table entry format. By tweaking the timeouts and the flow entry granularity data center operators can balance the scalability of measurement collection with the visibility that the measurements provide. In Section VI, we discuss more about how data center operators can deploy FlowDiff to maximize its effectiveness.

B. Application Signature

Modeling application signatures is challenging since an enormous number of applications with widely varying behaviors [33] may concurrently run inside a data center. To organize them in a manageable and scalable way, FlowDiff groups the applications into application groups: sets of application nodes inside the data center that form a connected communication graph. For example, in a three-tier application, the web, the application, and the database servers constitute an application group. However, multiple application groups may be mistakenly classified as a single one as they connect to common special-purpose nodes (data center services), e.g., network storage or DNS server. To unambiguously identify the application groups, we use domain knowledge to mark the special purpose nodes inside the data center. We consider application nodes connected *only* via special purpose nodes to be in separate application groups. Note that FlowDiff constructs application groups based on the local knowledge, which may not be expressive globally (e.g., two application groups can be connected via a node outside of local data center domain). However, a global knowledge is not required for diagnosing problems inside the data center.

We propose five signatures that capture application behavior in time, space (*i.e.*, across the data center), and volume (*i.e.*, data transferred) dimensions. Although these signatures do not reflect all aspects of an application behavior, they capture a wide range of problem classes, as depicted in Figure 2(b).



(a)

Problem Class	Signature Impact
Host failure	CG PCCI FS DD
Host Performance	DD PCFS
Application failure	CG PCCI FS DD
Application Performance	DD PCFS
Network disconnectivity	CG PC CI FS DD
Network bottleneck	DD PCFS
Switch Misconfiguration	CG PC CI FS DD
Switch Overhead	DD PCFS
Controller overhead	DD PC FS
Switch Failure	CG PC CI FS DD
Controller Failure	CG PC CI FS DD
Unauthorized Access	CG CI FS

(b)

Fig. 2. Application signatures (a) capture application behavior in time, space, and volume dimensions, and (b) capture a wide range of problems classes.

Connectivity graph (CG) A connectivity graph represents the communication relationship between the servers where an application runs. For example, in a three-tier web application, the connectivity graph includes nodes for the web, application, and database servers, as well as links between them. We build CG using the source and destination IP metadata in the OpenFlow `PacketIn` messages.

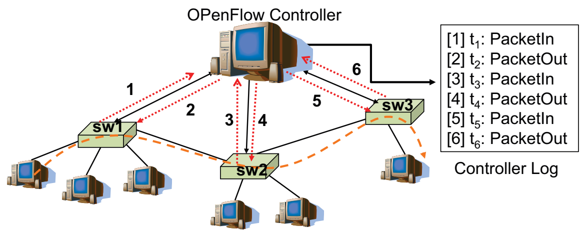
Flow statistics (FS) We use the control traffic measurements to compute the flow duration, the byte count, and the packet count of each flow corresponding to each application group. We also measure max, min, and average flow counts and volumes per unit of time.

Component interaction (CI) The component interaction at a node in CG represents the number of flows on each incoming or outgoing edge of the application node inside each application group. We normalize the CI value to the total number of communications to and from the node.

Delay distribution (DD) Given CG and CI, it is still difficult to identify the causal relationship between flows. As demonstrated by Chen *et al.* [7], the delays between dependent flows are time-invariant and can be used as a reliable indicator of dependencies. For example, suppose any request from flow f_1 arriving at a certain application node always triggers an outgoing flow f_2. If we measure the delays between all f_1's and all subsequent f_2's, the most frequent delay value is the processing time of f_1 at the application node. Inspired by this fact, we use peaks of the delay distribution frequency as one of the application signatures. FlowDiff identifies certain anomalies in the data center (*e.g.*, node overload and link congestion) by detecting changes in these delay peaks.

Partial correlation (PC) Although DD captures the causal relationship between dependent flows, the *strength* of the dependency is unknown. To quantify this, we calculate the partial correlation between adjacent edges for each CG using flow volume statistics. We divide the logging interval into equal spaced epoch intervals and, using the `PacketIn` messages during each epoch, we measure the flow count for each edge in the CG and compute the correlation over these time series data using the Pearson's coefficient [21].

Signatures may sometimes be unstable. For instance, if an application node does not use any linear decision logic across its outgoing flows, the component interaction signature becomes unstable. We do not use unstable signatures in the



[1] t_1: PacketIn
[2] t_2: PacketOut
[3] t_3: PacketIn
[4] t_4: PacketOut
[5] t_5: PacketIn
[6] t_6: PacketOut

Fig. 3. Modeling inter-switch latency and controller response time.

problem detection to avoid false positives in raising debugging flags. To determine whether a signature is stable, FlowDiff partitions the log into several time intervals and computes the application signatures for each interval. If a signature does not change significantly across all interval, we consider it stable and use it during problem detection.

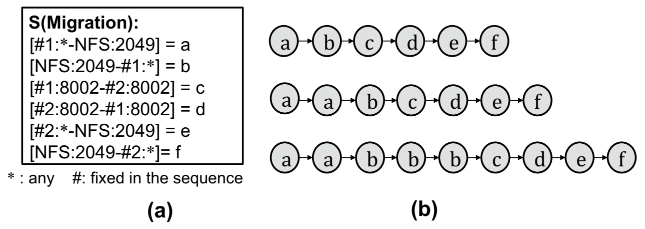
C. Infrastructure Signature

We use the infrastructure signature to characterize the connectivity and performance baseline of the physical data center infrastructure, composed of physical components (*e.g.*, switches, routers, servers) and common service nodes (*e.g.*, DNS, NTP, OpenFlow controller).

Physical topology (PT) FlowDiff computes the physical topology of the network using information from `PacketIn` and `FlowMod` messages. `PacketIn` messages contain information about the ingress port where the corresponding flow entered the switch, while `FlowMod` informs us of the output port. By combining `PacketIn` and `FlowMod` information from all switches that a flow traverses, we can determine the order of traversal and infer physical connectivity between them.

Inter-switch latency (ISL) The Inter-switch latency measures the delay between any two switches in the data center. When a flow arrives at a switch that has no matching entry in the flow table, it sends a `PacketIn` message to the controller. For a new flow, such reporting is performed by all the switches along the path. Using the timestamp of the receiving message at the controller, we infer the inter-switch latency. Figure 3 explains the process. The latency between `sw1` and `sw2` is $(t_3 - t_2)$, and between `sw2` and `sw3` is $(t_5 - t_4)$. t_i represent the times when the switches send `PacketIn` or receive `FlowMod` and can be inferred from controller timestamps of the control messages. Since the individual latency measurements may vary depending on switch processing times [30], we use a statistical mean and the standard deviation of the ISL values as the signature.

Controller response time (CRT) We characterize the OpenFlow controller using its response time, the duration it spends processing a `PacketIn` message. We compute the response time by taking the difference between the timestamps of a `PacketIn` and its corresponding `FlowMod` message. In Figure 3, $(t_2 - t_1)$, $(t_4 - t_3)$, and $(t_6 - t_5)$ are controller response times. Similarly to the ISL signature, we use the mean and standard deviation of the response time.



S(Migration):
[#1:*-NFS:2049] = a
[NFS:2049-#1:*] = b
[#1:8002-#2:8002] = c
[#2:8002-#1:8002] = d
[#2:*-NFS:2049] = e
[NFS:2049-#2:*]= f
* : any #: fixed in the sequence

Fig. 4. (a) Common flows and (b) flow sequences for three different runs of a VM migration task, derived from captured real-time traces. VM images are stored in a network file system (NFS). The physical hosts communicate to NFS via the port 2049. While migrating a VM from host A to host B, A first updates the image stored with the current VM states at the NFS and sends the migration request to host B at port 8002. When B accepts the migration, A starts sending the VM states to B. When the transfer is done, B synchronizes the VM state with the NFS server.

D. Task Signature

The task signature must capture the sequence of operational tasks from the collected flows. It is represented by a *task time series*, which is a time series of operational tasks occurred in different points in time. However, identifying the operational tasks from the application flows (captured via `PacketIn`) is challenging because the sequence of application flows even for a single task may vary due to caching, network retransmissions, packet reordering, or changes in application logic. For example, when a virtual machine starts inside a data center, the sequence of operations in terms of network flows (such as communication to the DHCP server, DNS server, NetBios, etc.) varies depending on the configuration of the virtual machine and its underlying OS (see Figure 4). To capture all possible flow sequences for an operational task in a compact form, we build a *finite-state automaton* using a supervised learning approach in three stages: (1) finding common flows across different logs for the same task, (2) extracting the frequent flow sequences, and (3) building the task automaton. Figure 5 depicts these three stages.

(1) Finding common flows First, for a specific task T, we capture multiple logs in different regions of the data center and find the common flows among them, $S(T)$. For a task T, the logs are represented as T_1,T_2,$\cdots$, T_n and the set of flows inside the logs are $S(T_1)$,$S(T_2)$,$\cdots$, $S(T_n)$, respectively, where n is the number of logs captured. Therefore $S(T) = \cap_{i=1}^{n} S(T_i)$. A flow is defined by the source-destination IPs and ports.

(2) State extraction We now construct T'_i from T_i ($1 \leq i \leq n$), by removing the flows that do not belong to set $S(T)$. For example, Figure 4 shows the common flows $S(T)$ (Figure 4(a)) and the constructed logs T'_i for a VM migration task across three different runs (Figure 4(b)).

Once we generate T'_i, the state extraction algorithm is similar to the classic sequential frequent pattern mining algorithm [1]. The goal is to capture the sub-sequence of flows that are frequent in T'_i for all $1 \leq i \leq n$ and to consider each of them in a single state in the automata to reduce the number of state transition. A sequence is frequent if its support value is higher than minimum support value (min_sup) defined by the operator. The algorithm runs in several iterations and collects

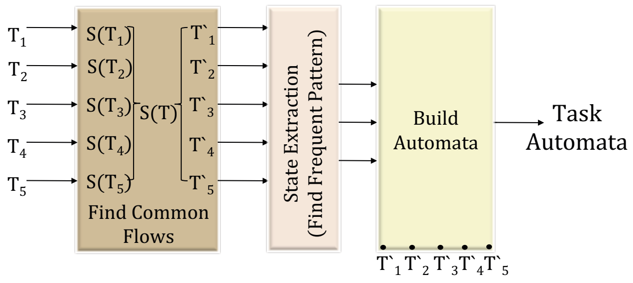


Fig. 5. Steps in building Task Signature.

the frequent patterns in increasing length starting from length 1. Frequent patterns of length $(i-1)$ are used to capture the frequent patterns of length i. Finally, we perform a pruning to find the list of all closed frequent pattern [19] sequences in all T'_i. The closed frequent pattern reduces the state redundancy by ensuring that if p_1 and p_2 are two frequent patterns of any length, where $p_1 \in p_2$ and they have the same support value, then p_1 is pruned from the frequent pattern list.

Figure 6(a) illustrates the extraction process. Assume that we have three extracted logs: $T'_1 = f_1f_2f_3f_4f_5$, $T'_2 = f_3f_4f_5f_1$ and $T'_3 = f_3f_4f_5f_2f_1$, where f_j represent different flows, and the min_sup is set to 0.6. The number in the parentheses (in Figure 6(a)) is the support value associated with the sequence appeared in the traces. The length-2 sequences marked with 'X' do not pass the threshold and will not be put into the frequent pattern list for the next step. The extraction process ends at the length-3 pattern. Thus, the length of the longest state is three. Finally, if we apply pruning on them. f_3, f_4, and f_5 at first iteration, and f_3f_4 and f_4f_5 at second iteration will be pruned, because the length-3 sequence $f_3f_4f_5$ subsumes all of them with the same support value.

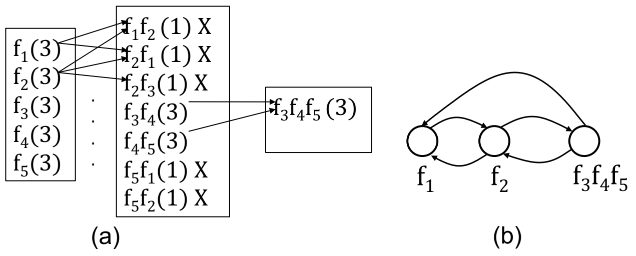


Fig. 6. (a) Example of state extraction, and (b) the final task automata out of the extracted logs.

(3) Building the automaton We sort the extracted frequent pattern list (or states) of same length according to their support value. For each T'_i, we build the automaton using the sorted states with two rules: 1) for different sets of state, choose the one with longer length first, and 2) for states of the same set, choose the more frequent one first (*i.e.*, with higher support value). Based on these rules, all extracted logs can be precisely represented by the constructed automata. Using three extracted logs T'_1, T'_2 and T'_3 for task T (of the previous example), the final automaton is shown in Figure 6(b).

To build the task signature (i.e, the task time series), FlowDiff detects operational tasks from the input logs using the learned task automata. The detection process first checks whether, at any point of time, a flow (reported by `PacketIn`) matches the start state of any task automata previously learned. If a match is found, a child thread is created that checks for the complete match of that automata starting from that time. The main process then moves into the next flow and keeps checking for the further match with any start states. The child process in parallel tries to match the automata by taking the transitions for the incoming flows. Note that, we consider a flexible matching of the automata since due to Internet concurrency the flow sequence of a task signature can be interleaved with other network traffics. However, we bound the interleaving threshold by 1 second. If the threshold is over, the child process checking the existence of a task automata is terminated. The child processes that arrive at the final state of any automata during matching are considered successful and the tasks corresponding to the child processes are added into the task time series with the stating timestamp.

IV. Diagnosis with FlowDiff

To detect performance and operational problems in data center networks, FlowDiff compares application and infrastructure signatures taken at different times and validates the differences against known task signatures. We describe these two steps below. We consider two controller logs $L1$ and $L2$ collected during two different time intervals $[t_a, t_b]$ and $[t_m, t_n]$. $L1$ captures the normal behavior of the data center. Each log contains a list of events (*e.g.*, `PacketIn`, `FlowMod`, `FlowRemoved` messages) and their timestamps. FlowDiff first builds stable application and infrastructure signatures from the reference log $L1$ by dividing the log into multiple time segments. FlowDiff also constructs signatures of $L2$ in the interval $[t_m, t_n]$ to compare against $L1$.

A. Comparing signatures

Application signatures To compare different application signatures, we use following approaches:

- To detect changes in scalar application signatures, such as partial correlation and flow statistics, FlowDiff compares their values in the two logs ($L1$ and $L2$).
- For changes in the connectivity graph of an application, we use a simple graph matching algorithm, which returns the list of missing or new edges in the connectivity graph of $L2$ compared to that of $L1$.
- To compare the component interaction signatures, FlowDiff performs a χ^2 fitness test between the flow count distributions on each edge at each application node. The χ^2 value is: $\chi^2 = \sum_{i=1}^{N} \frac{(O_i - E_i)^2}{E_i}$, where E is the expected flow count value (from $L1$), O is the observed flow count value (from $L2$), and N is the number of incoming and outgoing edges at a node. Operators define the threshold value to determine significant differences.
- We measure the peaks in the delay distribution between the same adjacent edges in logs $L1$ and $L2$. If the peak value shift is higher that a predetermined threshold, the server that connects the two edges may experience performance degradation and FlowDiff raises an alarm.

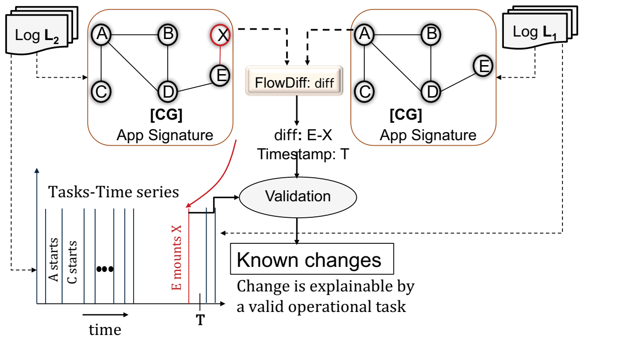


Fig. 7. Identifying known and unknown changes using task-time series generated from Log $L2$.

Infrastructure signatures We compare changes in the controller response time, inter-switch latency, and physical path for each application between $L1$ and $L2$. FlowDiff raises an alarm if the latency increases beyond a certain threshold or if the controller becomes overloaded and potentially causes increased queuing on switches or packet drops.

B. Validating changes

The goal of this step is to identify if a detected change is valid or not. Changes identified in the previous sections are categorized into known and unknown changes in both application and infrastructure layers. Known changes can be explained by valid operational tasks defined by the task signatures (generated from both $L1$ and $L2$). For example, VM migration moves a running VM from one switch to another and causes a known and expected change in the physical topology. Unknown changes occur due to unexpected problems such as the ones listed in Figure 2(b). Figure 7 shows a diagram of how FlowDiff validates a change in the connectivity graph signature (an edge from E to X is present in the current log $L2$ at timestamp T but not in the baseline log $L1$).

C. How FlowDiff can help operators

The goal of FlowDiff is to detect operational problems and notify operators which components (*i.e.*, servers, switches, or links) are related to the problems. We discuss below how operators may use FlowDiff information to gain more details about what causes the operational problems.

To identify the type of problem, we find the dependencies between application and infrastructure signature changes, which are not validated by the task signatures. Let A be the dependency matrix, where the rows are associated with application signatures and the columns with infrastructure signatures, as shown in Figure 8. A_{ij} is 1 if for a change in the i^{th} component of the application signature, there is also a change in the j^{th} component of the infrastructure signature.

Each combination of dependencies between application and infrastructure signatures (*i.e.*, combination of cells with value 1) represents a type of problem. For example, when congestion occurs, the delay distribution (DD), partial correlation (PC), and flow statistics (FS) change along with the inter-switch latency (ISL) and the corresponding cells in the matrix are 1 (see Figure 8(a)). Figure 8(b) shows the dependency matrix for

	PT	ISL	CC
CG	0	0	0
DD	0	1	0
CI	0	0	0
PC	0	1	0
FS	0	1	0

(a) congestion

	PT	ISL	CC
CG	1	0	0
DD	0	0	0
CI	0	0	0
PC	0	0	0
FS	0	0	0

(b) switch failure

Fig. 8. Example of dependency matrix for congestion and switch failure.

switch failure. All possible interpretations of the dependency matrix are skipped due to the brevity.

Moreover, while comparison, FlowDiff returns a set of edges and nodes that are related to each infrastructure and application signature changes. To localize the operational problem that triggered these changes, we rank the components based on the number of changes they are associated with. The components with higher rank is more likely to be related to the problem.

V. Evaluation

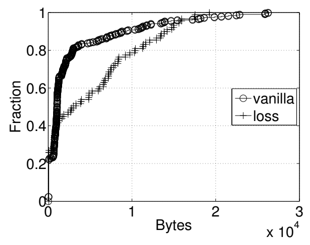
In this section, we evaluate FlowDiff from three perspectives: effectiveness, robustness, and scalability. We start by studying the effectiveness of FlowDiff in identifying problems under realistic scenarios. Then, we evaluate the robustness and stability of application and task signatures. Finally, we use simulations to show how our tool scales when the number of applications and the size of the data center increase. We use three experimental setups to evaluate FlowDiff:

Lab data center. We set up a small data center consisting of 25 physical machines and five virtual machines. The servers are connected using seven OpenFlow switches (two hardware-based NEC PF5240 and five software-based running OpenFlow V1.0) and two D-Link traditional switches. We use NOX [17] to control the switches. All traffic between any two servers passes through at least one OpenFlow switch.

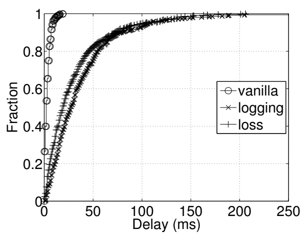
We deploy the following applications in our data center: (1) Petstore, a three-tier retail site, (2) Rubis, a three-tier auction site, (3) Rubbos, a three-tier bulletin board system, and (4) osCommerce, a two-tier online merchant system. In addition, we build a custom three-tier application to allow customization in the application logic (e.g., connection reuse). We use Tomcat and JBoss as application severs and mySQL as database server. Standard *http* client emulators generate traces with different workload for each of these applications.

Amazon EC2. We deploy four virtual machines on different regions of Amazon EC2. One of the VMs is a Ubuntu installation and the other three are Amazon AMI VMs. We add *tcpdump* in the machine start up sequence (boot order) after the networking service starts for logging the flows created during startup. We use this setup to give an example of how FlowDiff is able to identify a specific task, VM startup, in a real world outside-the-lab scenario.

Simulation. To evaluate the scalability of FlowDiff, we simulate a 320-server network, arranged in a tree topology as follows. Each rack of 20 servers connects to a top-of-rack (ToR) switch. Every four ToRs are connected to two aggregation switch. All eight aggregation switches are connected to two core switches.



(a) Byte count



(b) Delay distribution

Fig. 9. Packet loss on a link or utilization increase on a server lead to increases in the (a) byte count of the flows traversing the link and (b) delays between incoming and outgoing flows at the server.

A. Effectiveness

FlowDiff identifies both unknown known operational problems in a data center. First, we run FlowDiff in the wild and attempt to detect a VM startup operation in the EC2 setup. As we do not have access to Amazon's network, we are restricted to build and evaluate task automata based on the traffic on a single VM. We choose the VM startup operation because we can easily collect all the flows that it generates by inserting a *tcpdump* command at the beginning of the startup sequence. For all four VMs, FlowDiff task signatures can successfully detect a startup event using the generated task automata.

We also perform experiments on the lab data center by injecting various operational problems and checking whether and how FlowDiff detects them. Table I shows the list of problems we introduce. For each problem, the second column in the table gives the list of signature components that change. The third column shows the identification of problem type which can be further diagnosed by operator. FlowDiff detects successfully each problem.

To illustrate how signatures change when a problem occurs, consider a four node three-tier application with one web server connected to two application servers, which are then connected to one database server. We introduce two types of faults:

- **network faults**: we inject 1% loss on both links connecting the web and application server. This changes the flow statistics (*i.e.*, byte count) and the latency of the flows traversing the two edges.
- **server faults**: we enable logging on the application server. This increases the request processing time on the application server and implicitly modifies delay difference between the incoming and outgoing flows.

Figure 9 presents the CDFs of the byte count of flows incoming to the application servers and of the delay distribution between the outgoing and incoming flows at the application servers. Intuitively, we expect that introducing loss increases both the byte count (due to the retransmissions) and the delay distribution (because flows are delayed due to retransmissions). Similarly, enabling logging leads to an increase in request processing and subsequently, an increase in the delay distribution. The results in Figure 9 match our intuition: the flow statistics (FS) and delay distribution (DD) change significantly after introducing the problems. By capturing such changes, FlowDiff can identify the problems that causes them.

B. Robustness

For FlowDiff to correctly detect operational problems in data centers, it needs to capture both a stable snapshot of the data center behavior (application signatures) and stable flow sequences of operational tasks (task automata). Next we evaluate how well is FlowDiff able to construct stable and robust application and task automata.

1) Application signatures: To measure the stability of application signatures across various workloads or at different logging times, we use the lab data center experimental setup. We consider five different application deployment case, described in Table II. The server numbers do not have any particular significance, other than helping us identify the servers in the testbed. For each case, we run the applications multiple times, modifying the workloads (by varying the traffic distribution) and, when possible, the application logic (by varying the connection reuse parameters). The traffic logging lasts for 45 minutes for each run. During one run, we ensure that the data center runs smoothly, without any infrastructure or application layer modifications. We evaluate the stability of each application signature component.

Connectivity graph. The variations in workload or application setup do not have any impact on the connectivity graphs captured by FlowDiff. This is because the CG depends on the internal structure of each application (*e.g.*, what application components communicate with each other) which is independent on the properties of the input traffic.

Delay distribution. The delay distribution between flows may be affected by the type of input workload and by the application logic. For example, for each incoming flow, an application may generate a new outgoing flow or may use an existing outgoing connection. Because the flow-based switches send `PacketIn` messages to the controller only when a new flow arrives, connection reuse at the application layer results in incomplete information about dependent flows.

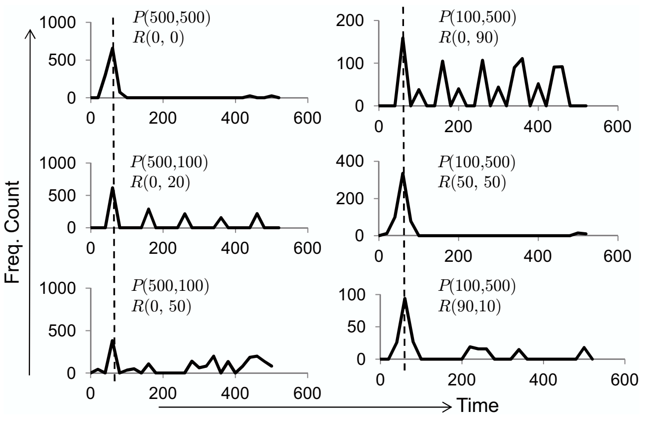


Fig. 10. The robustness of delay distribution for different workload and application logic between communication of S2-S3 and S3-S8 for case 5 shown in Table II.

To assess the stability of delay distribution, we use case 5 in Table II, which represents a custom setup of a three-tier application and gives us flexibility to modify both the

TABLE I
DEBUGGING WITH FLOWDIFF

ID	Problem Introduced	Impact on signatures	Problem Inference
1	Mis-configure to enable “INFO” mode logging on Tomcat	DD	Host or Application Problem
2	Emulate loss using tc on the server	DD, FS	Host network problem, Network congestion
3	High CPU(background process)	DD	Host or Application Problem
4	Application crash	CG, CI	Application Failure
5	Host/VM Shutdown	CG, CI	Host Failure
6	Firewall (port block)	CG, CI	Host or Application Problem
7	Inject Background traffic using Iperf	ISL, FS, PC, DD	Network Congestion Problem

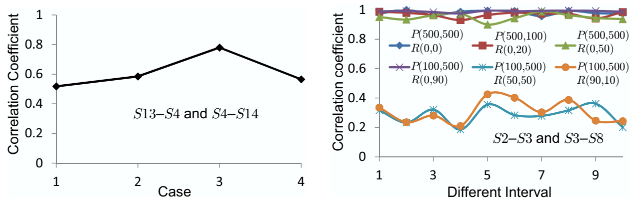
TABLE II
CASE STUDIES: ROBUSTNESS OF APPLICATION SIGNATURE.

Case	List of application groups
1	**Rubbis:** S25 (client) — S13 (web-server) — S4 (app-server) — S14 (db-server) — S15 (slave-db) **Rubbis:** S24 (client) — S12 (web-server)— S10 (app-server) — S20(db-server) **osCommerce:** S23 (client) — S7 (web-server) — S10 (app-server) — S20(db-server)
2	**Rubbis:** S25 (client) — S12 (web-server) — S4 (app-server) — S14 (db-server) — S15 (slave-db) **osCommerce:** S23 (client) — S7 (web-server)— S10 (app-server) — S20(db-server)
3	**Rubbis:** S25 (client) — S12 (web-server) — S4 (app-server) — S14 (db-server) — S15 (slave-db) **Rubbos:** S24 (client) — S12 (web-server) — S10 (app-server) — S20(db-server)
4	**Rubbis:** S25 (client) — S12 (web-server) — S4 (app-server) — S14 (db-server) — S15 (slave-db) **Petstore:** S24 (client) — S16 (web-server) — S25 (app-server) — S19(db-server)
5	**Custom:** S22 (client) — S1 (web-server) — S3 (app-server) — S8 (db-server) **Custom:** S21 (client) — S2 (web-server) — S3 (app-server) — S8 (db-server) **Custom:** S23 (client) — S5 (web-server) — S11 (app-server) — S18 (db-server) **Custom:** S23 (client) — S5 (web-server) — S17 (app-server) — S6 (db-server)

application logic and the input workload, as we show below. Figure 10 shows DD between flows S2-S3 and S3-S8 for various workload distributions and connection reuse ratios. $P(x, y)$ indicates that the workload has a Poisson distribution with statistical mean x and y across web server to the application server for S1-S3 and S2-S3. $R(m, n)$ indicates that $m\%$ and $n\%$ of connections are reused at the application server S3 for the database communication of any incoming request via S1-S3 and S2-S3. We plot the delays with bins of $20ms$ along the x-axis. Even with the varying connection reuse (from 10% to 90%) and with different workloads, the peak value of inter-flow delay persists within $[40, 60]ms$ ($60ms$ is the ground truth).

Partial correlation. Figure 11 shows the partial correlations between the dependent flows S13-S4 and S4-S14 for the application group S25-S13-S4-S14-S15 in cases 1 to 4 (in Table II). The PC values are stable across the runs and do not vary with the workload. To check the stability of PC between the dependent flows when connections are reused, we use case 5 (see Table II). We partition the log into 10 intervals (1.5 minutes each) and compute the partial correlation by varying the application logic (*i.e.*, percentage of connection re-use) and workloads as described above. The results, shown in Figure 11(b), prove that the partial correlation is relatively stable even with connection re-use.

Component interaction. Figure 12 shows the change in the component interaction graph of node S4 in the application group S25-S13-S4-S14-S15 for cases 1 to 4. The bars represent the normalized flow count in (blue bars) and out (red bars) of application server node S4. The values do not vary much across case studies. We also present the χ^2 values considering



(a) (b)

Fig. 11. Change in partial correlation signature between (a) flows $S13$-$S4$ and $S4$-$S14$ of application group $S25$-$S13$-$S4$-$S14$-$S15$ for case 1 to 4, and (b) flows $S2$-$S3$ and $S3$-$S8$ in case 5 for different intervals.

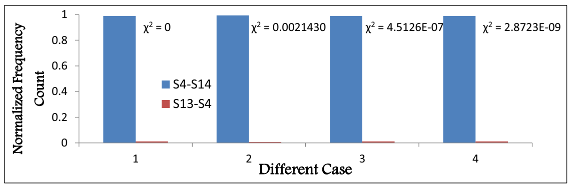


Fig. 12. Change in component interaction graph at application server node S4 of application group S25-S13-S4-S14-S15 for case 1 to 4.

the values at case 1 as the expected values. In many cases, the component interaction may not be stable (*e.g.*, non uniform load balancing at application node S5 in case 5). When this happens, FlowDiff does not consider component interaction as part of the normal stable data center signature.

2) Task signatures: For FlowDiff to accurately identify planned or valid operational changes (i.e., task signatures), the task automata must be stable. We evaluate the stability of task automata in the EC2 setup using *VM startup* as the operator task. We detect the flows that correspond to VM startup for each for the four VMs and build the task signatures. We use 50 runs to ensure that the task automata are stable.

We compute two types of task automata: with masked IPs and without masked IPs. Masking the IP addresses allows a task automata to be more general: instead of matching the startup task on the same VM from whose flows it was constructed, the automata would match the task on *any* VM. We start each VM several times and try to match the startup operation against the previously generated task automata.

Table III presents the results. True positives (TP) are those cases where FlowDiff should find a successful match for the operational task. False positives (FP) are the runs where FlowDiff incorrectly finds a match. The second number in each cell represents how many times we run each experiment by restarting the corresponding VM. When IPs are not masked, there are no false positives: FlowDiff never matches to an incorrect task. However, when we mask IP addresses, the traces become more general and there are a few cases where FlowDiff incorrectly finds a match. Ubuntu VM is never wrongly matched with Amazon AMI VM's whereas Amazon AMI VM's may match with each other (as they have same base OS). FlowDiff achieves an almost perfect true positive rate and and very low false positive rate.

We repeated the experiments using five types of tasks on our lab testbed and obtained similar results. We performed, *VM startup*, *VM stop*, *VM migration*, *mount network storage*, and *unmount network storage*. All these tasks involve flows to/from a single host and their task signatures have unique sequences of connections. We leave experimenting with operator tasks involving connections to multiple hosts (*e.g.*, update VLAN or ACL) to future work.

TABLE III
ACCURACY OF TASK SIGNATURE MATCHING

ID	AMI name	TP (not masked)	TP (masked)	FP (masked)
1	i-$3486634d$ (AMI)	20/20	18/20	1/40
2	i-$5d021f3b$ (AMI)	17/20	14/20	4/40
3	i-$c5ebf1a3$ (Ubuntu)	5/5	5/5	0/60
4	i-$d55066b3$ (AMI)	20/20	19/20	7/40

C. Scalability

We next conduct a scalability study of FlowDiff using simulation. We randomly generate a set of three-tier applications and randomly place their VMs on the network described at the beginning of the section. Within an application, every VM in the same tier communicates with every VM in the next tier. We construct a traffic pattern according to the measurement study [4] performed by Benson *et al.*. In particular, for each communicating VM pair, the traffic follows an ON/OFF pattern, in which both the ON and OFF periods follow log normal distribution with mean 100ms and standard deviation 30ms. For more realism, we consider TCP connection reuse, which is commonly employed in popular protocols such as SQL, JDBC, HTTP. If connections are reused, multiple flows share the same TCP connection, and therefore do not trigger multiple `PacketIn` requests. In the paper, we use a connection reuse probability of 0.6 to simulate the effect.

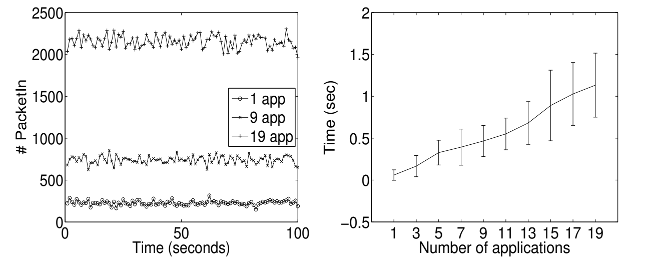


Fig. 13. Scalability test of FlowDiff: (a) Rate of `PacketIn` messages for various applications. (b) The processing time of FlowDiff grows sub-linearly with the number of applications.

Given the above setup, we first generate traffic with $N = 1, 3, 5, \ldots, 19$ applications and collect the `PacketIn` messages. Figure 13(a) shows the number of `PacketIn` requests per second for different numbers of applications. We next feed these traces to FlowDiff and record the processing time. We repeat 90 times and calculate the mean and variance of the processing time. As illustrated in Figure 13(b), the processing time of FlowDiff scales sub-linearly with the number of applications (or the average number of `PacketIn` requests per second at the controller).

Even though, the processing time of FlowDiff scales sub-linearly with the number of applications and requests, there is a threshold beyond which the framework becomes too slow to be useful in detecting problems. Assuming a peak controller rate of $100K$ requests/second [6], FlowDiff can process an 100 seconds of log in under one minute, which is reasonable for an offline analysis tool. However, previous studies show that controllers in large networks (*e.g.*, >100 switches), with new flows arriving every $10\mu s$, may have to precess up to $10M$ `PacketIn` messages per second [4]. Under such scenarios, we have to consider alternative OpenFlow deployment scenarios that reduce the load on the controller while maintaining the expressivity of the control information captured. We discuss these deployments in the next section.

VI. DEPLOYMENT CONSIDERATIONS

Deploying FlowDiff at scale requires an efficient and scalable OpenFlow deployment. We discuss the suitability of various decisions in operating OpenFlow networks and how they affect the functionality of FlowDiff. Previous research [6], [23], [34] and conversations with operators of OpenFlow-based networks reveal several approaches to scaling: distribute controller functionality across multiple machines, proactively set up rules to avoid triggering control traffic, and use wildcard rules to reduce the amount of control traffic.

Distributed controller Distributing the controller does not affect the amount or frequency of control traffic but it simplifies the messages processing task. Using a mechanism similar to FlowVisor [31], we can can capture control traffic and synchronize the information captured across controllers. Network virtualization solutions such as Nicira's NVP [27] use a cluster of controllers to manage large-scale network state.

Wildcard rules Wildcard rules reduce the number of control messages and installed rules because there are fewer new flows without a match in the flow table. This limits the granularity of measurements derived from control packets. For example, if a wildcard rule covers many input ports of a switch, the behavioral model constructed from control measurements would not be able to pinpoint the specific link on which a failure occurred and would be limited to indicate the collection of links corresponding to the wildcard rule.

Proactive deployment When operators install rules proactively, new flows at a switch do not trigger `PacketIn` because they find a matching rule in the flow table. Further, if rules have large timeouts, they take long to expire and fire `FlowRemoved` messages. Some entries may be set up to not trigger `FlowRemoved` when they expire. Although one could easily trigger control traffic (*e.g.*, by injecting packets that are not covered by existing rules), this traffic would capture only infrastructure signatures. Thus, FlowDiff would not be suitable for OpenFlow operational modes that remove or heavily limit the amount of control traffic between switches and controller.

Incremental deployment Deploying FlowDiff on a hybrid data center network where a subset of switches are OpenFlow-enabled and operate reactively is a more realistic proposal and may provide a good trade-off between scalability and accuracy. From conversations with operators, we have learned that hybrid data center networks where the aggregation switches are OpenFlow-enabled are already in production. Similar to the wildcard rules, the granularity of problem detection for hybrid data centers is limited by the granularity of measurements. For example, FlowDiff can localize a performance issue on a path, but not on a link, unless the link is between OpenFlow switches. In addition, a smaller number of switches would also limit the amount of control traffic that controllers process and make the problem detection more amenable.

VII. Conclusion

We present FlowDiff, a flow-based data center diagnosis tool that models data center applications, infrastructure and operator tasks using network flow logs collected from OpenFlow controller. FlowDiff is scalable and efficient in flow monitoring, non-intrusive, easily deployable and usable by the network operators. FlowDiff effectively constructs application and infrastructure signatures that capture data center behavior and compares signature at different points in time to identify changes that cannot be attributed to normal operator tasks. Using FlowDiff, we were able to identify a wide range of behavioral abnormalities related to end-host, network or application performance problems.

References

[1] R. Agrawal and R. Srikant, “Mining sequential patterns,” in *Proc. of ICDE*, 1995.

[2] P. Bahl and R. Chandra et al., “Towards highly reliable enterprise network services via inference of multi-level dependencies,” in *Proc. ACM Sigcomm*, 2007.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, “Using magpie for request extraction and workload modelling,” in *Proc. OSDI*, 2004.

[4] T. Benson, A. Akella, and D. Maltz, “Network Traffic Characteristics of Data Centers in the Wild,” in *Proc. ACM IMC*, 2010.

[5] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, “Fingerprinting the datacenter: automated classification of performance crises,” in *Proc. EuroSys*, 2010.

[6] Z. Cai, A. L. Cox, and T. E. Ng, “Maestro: A System for Scalable OpenFlow Control,” Rice University, Tech. Rep. TR11-07, 2011.

[7] X. Chen and M. Zhang et al., “Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions,” in *Proc. of OSDI*, 2008.

[8] Why Amazon’s cloud Titanic went down. [Online]. Available: http://money.cnn.com/2011/04/22/ technology/amazon_ec2_cloud_outage/index.htm.

[9] I. Cohen, M. Goldszmidt, T. Kelly, and J. Symons, “Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control,” in *Proc. OSDI*, 2004.

[10] Estimated cost of Amazon downtime. [Online]. Available: http://ophir.wordpress.com/2011/04/22/ amazons-cloud-downtime-cost-estimated-at-two-million-dollars/.

[11] “Data center outages generate big losses,” http://www.information week.com/news/hardware/data_centers/ 229500121.

[12] N. Feamster and H. Balakrishnan, “Detecting BGP Configuration Faults with Static Analysis,” in *Proc. NSDI*, 2005.

[13] R. Fonseca and G. Porter et al., “X-Trace: A Pervasive Network Tracing Framework,” in *Proc. NSDI*, 2007.

[14] IBM Launches Beefy Openflow Switch for Data Centers Cloud. [Online]. Available: http://gigaom.com/cloud/ibm-launches-beefy-openflow-switch-for-data-centers-cloud/.

[15] D. Geels, G. Altekar, S. Shenker, and I. Stoica, “Replay debugging for distributed applications,” in *Proc. USENIX ATC*, 2006.

[16] M. Goldszmidt, A. Fox, I. Cohen, J. Symons, S. Zhang, and T. Kelly, “Capturing, indexing, clustering, and retrieving system history,” in *Proc. SOSP*, 2005.

[17] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, “NOX: Towards an Operating System for Networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, 2008.

[18] R. Guerraoui and M. Yabandeh, “Model checking a networked system without the network,” in *Proc. NSDI*, 2011.

[19] J. Han, H. Cheng, D. Xin, and X. Yan, “Frequent pattern mining: current status and future directions,” in *Proc. of DMKD*, 2007.

[20] IBM and NEC Jointly Enable Tervela and Selerity to Transform Their Networks with OpenFlow. [Online]. Available: http://www.wwpi.com/.

[21] S. L. Jackson, *Research Methods and Statistics: A Critical Thinking Approach*. Cengage Learning, 2008.

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

[23] T. Koponen and M. Casado et al., “Onix: A Distributed Control Platform for Large-scale Production Networks,” in *Proc. USENIX OSDI*, 2010.

[24] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, B. Godfrey, and S. T. King, “Debugging the data plane with anteater,” in *Proc. ACM Sigcomm*, 2011.

[25] N. McKeown and T. Anderson et al., “OpenFlow: enabling innovation in campus networks,” *ACM Sigcomm CCR*, vol. 38, pp. 69–74, March 2008.

[26] K. Nagaraj, C. Killian, and J. Neville, “Structured comparative analysis of system logs to diagnose performance problems,” in *Proc. NSDI*, 2012.

[27] “Network Virtualization Platform,” http://nicira.com/en/network-virtualization-platform.

[28] L. Popa, B. G. Chun, and I. Stoica et al., “Macroscope: End-Point Approach to Networked Application Dependency Discovery,” in *Proc. ACM CoNEXT*, 2009.

[29] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, “Pip: detecting the unexpected in distributed systems,” in *Proc. NSDI*, 2006.

[30] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, “OFLOPS: An Open Framework for OpenFlow Switch Evaluation,” in *Proc. PAM*, 2012.

[31] R. Sherwood and G. Gibb et al., “Can the Production Network Be the Test-bed,” in *Proc. USENIX OSDI*, 2010.

[32] M. Yabandeh and N. Knežević et al., “Predicting and preventing inconsistencies in deployed distributed systems,” *ACM Trans. Comput. Syst.*, vol. 28, pp. 2:1–2:49, August 2010.

[33] M. Yu and A. Greenberg et al., “Profiling Network Performance for Multi-Tier Data Center Applications,” in *Proc. of NSDI*, 2011.

[34] M. Yu and J. Rexford et al., “Scalable flow-based networking with DIFANE,” in *Proc. ACM Sigcomm*, 2010.