Detecting Performance Anomalies in Cloud Platform Applications

Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski
Computer Science Department, Univ. of California, Santa Barbara

Abstract—We present Roots, a full-stack monitoring and analysis system for performance anomaly detection and bottleneck identification in cloud platform-as-a-service (PaaS) systems. Roots facilitates application performance monitoring as a core capability of PaaS clouds, and relieves the developers from having to instrument application code. Roots tracks HTTP/S requests to hosted cloud applications and their use of PaaS services. To do so it employs lightweight monitoring of PaaS service interfaces. Roots processes this data in the background using multiple statistical techniques that in combination detect performance anomalies (i.e. violations of service-level objectives). For each anomaly, Roots determines whether the event was caused by a change in the request workload or by a performance bottleneck in a PaaS service. By correlating data collected across different layers of the PaaS, Roots is able to trace high-level performance anomalies to bottlenecks in specific components in the cloud platform. We implement Roots using the AppScale PaaS and evaluate its overhead and accuracy.

Index Terms—Performance anomaly detection, Root cause analysis, Cloud computing

1 Introduction

Cloud computing is a popular approach for deploying applications at scale [1], [2]. This widespread adoption of cloud computing, particularly for deploying web applications, is facilitated by ever-deepening software abstractions. These abstractions elide the complexity necessary to enable scale, while making application development easier and faster. But they also obscure the runtime details of cloud applications, making the diagnosis of performance problems challenging. Therefore, the rapid expansion of cloud technologies combined with their increasing opacity has intensified the need for new techniques to monitor applications deployed in cloud platforms [3].

Application developers and cloud administrators generally wish to monitor application performance, detect anomalies, and identify bottlenecks. To obtain this level of operational insight into cloud-hosted applications, the cloud platforms must support data gathering and analysis capabilities that span the entire software stack of the cloud. However, most cloud technologies available today do not provide adequate application monitoring support. Cloud administrators must therefore trust the application developers to implement necessary instrumentation at the application level. This typically entails using third party, external monitoring software [4], [5], [6], which significantly increases the effort and financial cost of maintaining applications. Developers must also ensure that their instrumentation is both correct, and does not degrade application performance. Nevertheless, since the applications depend on extant cloud services (e.g. scalable database services, scalable in-memory caching services, etc.) that are performance opaque, it is often difficult, if not impossible to diagnose the "root cause" of a performance problem using such extrinsic forms of monitoring.

Further compounding the performance diagnosis problem, today's cloud platforms are very large and complex [3], [7]. They are comprised of many layers, where each layer may consist of many interacting components. Therefore when a performance anomaly manifests in a user application, it is often challenging to determine the exact layer or the component of the cloud platform that may be responsible for it. Facilitating this level of comprehensive root cause analysis requires both data collection at different layers of the cloud, and mechanisms for correlating the events recorded at different layers.

Moreover, performance monitoring for cloud applications must be customizable. Different applications have different monitoring requirements in terms of data gathering frequency (sampling rate), length of the history to consider when performing statistical analysis (sample size), and the performance SLOs (service level objectives [8]) that govern the application. Cloud monitoring should be able to facilitate these diverse requirements on a per-application basis. Designing such customizable and extensible performance monitoring frameworks that are built into the cloud platforms is a novel and challenging undertaking.

To address these challenges, we develop a full-stack, application performance monitor (APM) called *Roots* [9], as a cloud Platform-as-a-service (PaaS) extension. PaaS clouds provide a set of managed services which developers compose into applications, via high-level interfaces (i.e., defined and exported via a software development kit (SDKs)). We design Roots as another PaaS service so that it can be managed automatically and directly capture events and performance data across the PaaS without requiring application code instrumentation.

Prior work outlines several key requirements for cloud APMs [3], [7], which we incorporate into Roots. They are:

Scalability Roots is lightweight, and does not cause any noticeable overhead in application performance. It puts strict upper bounds on the data kept in memory. The persistent data is accessed on demand, and can be removed after their usefulness has expired.

Multitenancy Roots facilitates configuring monitoring policies at the granularity of individual applications. Users can employ different statistical analysis methods to process the monitoring data in ways that are most suitable for their applications.

Complex application architecture Roots collects data from the entire cloud stack (load balancers, app servers, built-in PaaS services etc.). It correlates data gathered from different parts of the cloud platform, and performs systemwide bottleneck identification.

Dynamic resource management Cloud platforms are dynamic in terms of their magnitude and topology. Roots captures performance events of applications by augmenting the key components of the cloud platform. When new processes/components become active in the cloud platform, they inherit the same augmentations, and start reporting to Roots automatically.

Autonomy Roots detects performance anomalies online without manual intervention. When Roots detects a problem, it attempts to automatically identify the root cause by analyzing available workload and service invocation data.

Roots collects data from the logs and the interfaces of internal PaaS components. In addition to high-level metrics including request throughput and latency, Roots measures PaaS service invocations and their duration. It uses batch operations and asynchronous communication to minimize its overhead on request latency.

When Roots detects a performance anomaly in an application, it attempts to identify its root cause by analyzing the workload data and the performance of the internal PaaS services on which the application depends. Roots first determines if the detected anomaly was most likely caused by a change in the application workload (e.g. a sudden spike in the number of client requests), or by an internal bottleneck in the cloud platform (e.g. a slow database query). For the latter, Roots employs a statistical bottleneck identification method that combines quantile analysis, change point detection, and linear regression to identify the root cause bottleneck (i.e. the PaaS component that most likely caused the performance degredation).

We also devise a mechanism for Roots that distinguishes between different paths of execution in the application (control flows). Our approach does not require static analysis but instead uses the runtime data collected by Roots. This mechanism calculates the proportion of user requests processed by each path and uses it to characterize the workload of an application (e.g. read-heavy vs write-heavy workload in a data management application). Using this approach, Roots is able to detect when application workloads change.

We prototype Roots as an extension to the AppScale, open source PaaS [10]. We evaluate the feasibility and the efficacy of Roots by conducting a series of empirical trials using our prototype. We show that Roots is able to detect manually injected faults within 5 minutes of their injection with very low overhead. We also show that Roots is able to scale to tens of thousands concurrent applications.

2 Background

PaaS clouds have been experiencing a rapid growth in popularity in the recent years [11], [12]. They typically host web-accessible (HTTP/S) applications, while providing sandboxed execution, high scalability, and high availability. PaaS clouds are complex distributed systems that provide scalability by automatically allocating resources for applications on the fly (auto scaling), and availability through the execution of multiple instances of each application.

PaaS applications rely on a set of managed, scalable services offered by the underlying cloud platform. We refer to these services as PaaS kernel services. PaaS clouds like Google App Engine [13] and Microsoft Azure [14] export the kernel services via a well-defined set of APIs, that are collectively referred to as the PaaS "software development kit" (SDK). The application servers provide the linkage between application code and the PaaS kernel services. A set of front-end servers expose web application entry points, and provide load-balancing for HTTP/S clients invoking the applications.

By providing most of the functionality that applications require via kernel services, the PaaS model significantly reduces the amount of code that applications developers must write. PaaS clouds also relieve developers of the burden of configuration, deployment, and scaling through platform automation. In combination, the PaaS model significantly increases programmer productivity. However, a downside of this approach is that these features also hide the performance details of PaaS applications. Since the applications spend most of their time executing kernel services [15], it is challenging for the developers to diagnose performance issues given the opacity of the cloud platform's internal implementation.

One way to circumvent this problem is to instrument application code [4], [5], [6], and continuously monitor the time taken by various parts of the application. But such application-level instrumentation is tedious, and error prone thereby misleading those attempting to diagnose problems. Moreover, the instrumentation code may slow down or alter the application's performance. In contrast, implementing data collection and analysis as a kernel service built into the PaaS cloud allows performance diagnosis to be a "curated" service that is reliably managed by the cloud platform.

3 Roots

Roots is a holistic system for application performance monitoring (APM), performance anomaly detection, and root cause analysis. It is operated by the cloud providers as a builtin PaaS service that collects data from all the cloud components user applications interact with. Data collection, storage and analysis all take place within the cloud, and the insights gained are communicated to both the cloud administrators and application developers as needed. The key intuition behind Roots is that, as an intrinsic PaaS service, Roots has visibility into all activities of the PaaS cloud, across layers. Moreover, since the PaaS applications we have observed spend most of their time in PaaS kernel services [15], we hypothesize that we can infer application performance from observations of how the application uses the platform, i.e. by efficiently monitoring the time spent in PaaS kernel services. If we are able to do so, then we can avoid application instrumentation and its downsides, while detecting performance anomalies and identifying their root cause quickly and accurately.

The PaaS model that we assume with Roots is one in which the clients of a web application engage in a "service-level agreement" (SLA) [8] with the "owner" or operator of the application that is hosted in a PaaS cloud. The SLA stipulates a response-time "service-level objective" (SLO) that, if violated, constitutes a breech of the agreement. If the performance of an application deteriorates to the point that at least one of its SLOs is violated, we treat it as an *anomaly*. Moreover, we refer to the process of diagnosing the reason for an anomaly as *root cause analysis*. For a given anomaly, the root cause could be a change in the application workload or a *bottleneck* in the application runtime. Bottlenecks may occur in the application code, or in the PaaS kernel services that the application relies on.

Roots collects performance data across the cloud platform stack, and aggregates it based on request/response. It uses this data to infer application performance, and to identify SLO violations (performance anomalies). Roots can further handle different types of anomalies in different ways. We overview each of these functionalities in the remainder of this section.

3.1 Data Collection and Correlation

We must address two issues when designing a monitoring framework for a system as complex as a PaaS cloud.

1) Collecting data from multiple different layers.
2) Correlating data collected from different layers.

Each layer of the cloud platform is only able to collect data regarding the state changes that are local to it. A layer cannot monitor state changes in other layers due to the level of encapsulation provided by layers. However, processing an application request involves cooperation of multiple layers. To facilitate system-wide monitoring and bottleneck identification, we must gather data from all the different layers involved in processing a request. To combine the information across layers we correlate the data, and link events related to the same request together.

To enable this, we augment the front-end server of the cloud platform. Specifically, we have it tag incoming application requests with unique identifiers. This request identifier is added to the HTTP request as a header, which is visible to all internal components of the PaaS cloud. Next, we configure data collecting agents within the platform to record the request identifiers along with any events they capture. This way we record the relationship between application requests, and the resulting local state changes in different layers of the cloud, without breaking the existing level of abstraction in the cloud architecture. This approach is also scalable, since the events are recorded in a distributed manner without having to maintain any state at the data collecting agents. Roots aggregates the recorded events by request identifier to efficiently group the related events as required during analysis.

Figure 1 illustrates the high-level architecture of Roots, and how it fits into the PaaS stack. APM components are shown in grey. The small grey boxes attached to the PaaS components represent the agents used to instrument the cloud platform. In the diagram, a user request is tagged with the identifier value R at the front-end server. This identifier is passed down to the lower layers of the cloud along with the request. Events that occur in the lower layers while processing this request are recorded with the request

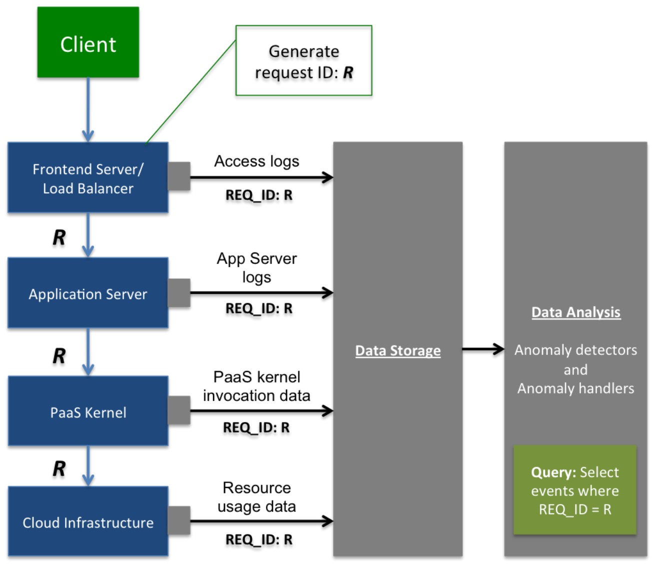

Fig. 1. Roots APM architecture.

identifier R, so Roots can correlate them later. For example, in the data analysis component we can run a filter query to select all the events related to a particular request (as shown in the pseudo query in the diagram). Similarly, Roots can run a "group by" query to select all events, and aggregate them by the request identifier.

The figure also depicts Roots data collection across the PaaS stack (i.e. its full stack monitoring). From the front-end server, Roots collects information related to incoming application requests. It does so by scraping HTTP server access logs, which are exported by most web servers (e.g. Apache HTTPD or Nginx).

At the application server level, Roots collects logs and metrics related to the application runtime from the application servers and operating system. Roots also employs a set of per-application benchmarking processes that periodically probes different applications to measure their performance. These are lightweight, stateless processes managed by the Roots framework. These processes send their measurements to the data storage component for analysis.

Roots collects information about all kernel invocations made by the applications by intercepting kernel invocations at service interface entrypoints. For each PaaS kernel invocation, we capture the following parameters.

- Source application making the kernel invocation
- Timestamp
- A sequence number indicating the order of PaaS kernel invocations within an application request
- Target kernel service and operation
- Execution time of the invocation
- Request size, hash, and other parameters

These PaaS kernel invocation details enable Roots to trace the execution of application requests through the PaaS without instrumenting the application itself.

Finally, at the lowest level Roots collects information related to virtual machines, containers and their resource usage. We gather metrics on network usage by individual components which is useful for traffic engineering use cases.

We also scrape hypervisor and container manager logs to track when resources are allocated and released.

To avoid introducing delays to the application request processing flow, we implement Roots data collecting agents as asynchronous tasks. Agents buffer data locally and periodically write to data storage components using separate background tasks and batch communication operations. These persistence operations must run with sufficient frequency so as to not impede the analysis that Roots employs to detect anomalies soon after they occur.

3.2 Data Storage and Analysis

Roots stores all collected data in a database capable of efficient persistent storage and querying. We facilitate this via indexing data by application ID and timestamp. Roots also performs periodic garbage collection on data that is no longer pertinent to analyses.

The data analysis components consist of two extensible abstractions: *anomaly detectors* and *anomaly handlers*. Anomaly detectors are processes that periodically analyze the data for each deployed application. Roots supports multiple detector implementations, each of which is a statistical method for detecting performance anomalies. Detectors are configured on a per-application basis, making it possible for different applications to use different anomaly detectors. Roots also supports multiple concurrent anomaly detectors for the same application, which can be used to compare the efficacy of different detection strategies concurrently. Each anomaly detector has configurable parameters for execution schedule and sliding window duration. We use a period 60 seconds for the former and the previous hour for the latter, in our prototype and evaluation. Window size impacts the time range of events processed by the detector when invoked. We employ a fixed-size window to bound Roots memory use.

When an anomaly detector detects an anomaly in application performance, it sends an event to a collection of anomaly handlers. The event encapsulates a unique anomaly identifier, timestamp, application identifier and the source detector's sliding window that correspond to the anomaly. Anomaly handlers are configured globally (i.e. each handler receives events from all detectors), but each handler filters events of interest. Handlers can also trigger events, which are delivered to all the listening anomaly handlers. Similar to detectors, Roots supports multiple anomaly handler implementations, e.g., one for logging anomalies, one for sending alert emails, one for updating a dashboard, etc. Additionally, Roots provides two special anomaly handlers: a workload change analyzer and a bottleneck identifier. Communication between detectors and handlers is performed via shared memory.

The ability of anomaly handlers to filter the events they process and to trigger events directly facilitates construction of elaborate event flows with sophisticated logic. For example, the workload change analyzer can run some analysis upon receiving an anomaly event from any anomaly detector. If an anomaly cannot be associated with a workload change, it can trigger a different type of event. The bottleneck identifier, can be configured to execute only when such an event occurs. Using this mechanism, Roots performs workload change analysis first and systemwide bottleneck identification only when necessary.

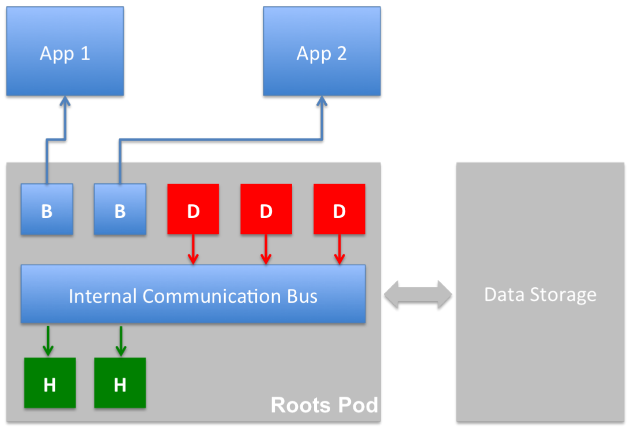

Fig. 2. Anatomy of a Roots pod. The diagram shows 2 application benchmarking processes (B), 3 anomaly detectors (D), and 2 handlers (H). Processes communicate via a shared memory communication bus local to the pod.

3.3 Roots Process Management

Most data collection activities in Roots can be treated as passive – i.e. they happen automatically as the applications receive and process requests in the cloud platform. They do not require explicit scheduling or management. In contrast, application benchmarking and data analysis are active processes that require explicit scheduling and management. This is achieved by grouping benchmarking and data analysis processes into units called Roots pods.

Each Roots pod is responsible for starting and maintaining a preconfigured set of benchmarkers and data analysis processes (i.e. anomaly detectors and handlers). These processes are light enough, so as to pack a large number of them into a single pod. Pods are self-contained entities, and there is no inter-communication between pods. Processes in a pod can efficiently communicate with each other using shared memory, and call out to the central Roots data storage to retrieve collected performance data for analysis. Furthermore, pods can be replicated for high availability, and application load can be distributed among multiple pods for scalability.

Figure 2 illustrates a Roots pod monitoring two applications. It consists of two benchmarking processes, three anomaly detectors and two anomaly handlers. The anomaly detectors and handlers are shown communicating via an internal shared memory communication bus.

4 Prototype Implementation

To investigate the efficacy of Roots as an approach to implementing performance diagnostics as a PaaS service, we have developed a working prototype, and a set of algorithms that uses it to automatically identify SLO-violating performance anomalies. For anomalies not caused by increases in workload (HTTP request rate), Roots performs further analysis to identify the bottleneck component that is responsible for the issue.

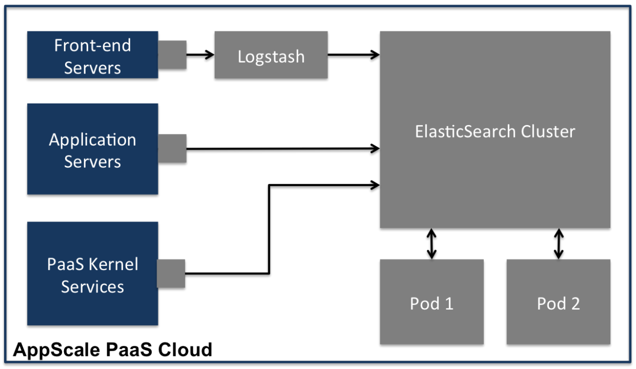

Fig. 3. Roots prototype implementation for AppScale PaaS.

We implement our prototype in AppScale [10], an open source PaaS cloud that is API compatible with Google App Engine (GAE) [13]. This compatibility enables us to evaluate our approach using real applications developed by others since GAE applications run on AppScale without modification. Because AppScale is open source, we were able to modify its implementation minimally to integrate Roots.

Figure 3 shows an overview of our prototype implementation. Roots components are shown in grey, while the PaaS components are shown in blue. We use ElasticSearch [16] as the data storage component of our prototype. ElasticSearch is ideal for storing large volumes of structured and semi-structured data [17]. It can be deployed as a scalable distributed service with sharding and replication. ElasticSearch continuously organizes and indexes data, making the information available for fast and efficient querying. Additionally, it also provides powerful data filtering and aggregation features, which greatly simplify the implementations of high-level data analysis algorithms.

We configure AppScale's front-end server (based on Nginx) to tag all incoming application requests with a unique identifier. This identifier is attached to the incoming request as a custom HTTP header. All data collecting agents in the cloud extract this identifier, and include it as an attribute in all the events reported to ElasticSearch.

We implement a number of data collecting agents in AppScale to gather runtime information from all major components. These agents buffer data locally, and store them in ElasticSearch in batches. Events are buffered until the buffer accumulates 1MB of data, subject to a hard time limit of 15 seconds. This ensures that the events are promptly reported to the Roots data storage while keeping the memory footprint of the data collecting agents small and bounded. For scraping server logs, and storing the extracted entries in ElasticSearch, we use the Logstash tool [18]. To capture the PaaS kernel invocation data, we augment AppScale's PaaS kernel implementation, which is derived from the GAE PaaS SDK. More specifically we implement an agent that records all PaaS SDK calls, and reports them to ElasticSearch asynchronously. Most metrics captured by our prototype are latency-related (e.g. latency of internal RPC calls and latency of cloud SDK calls). We wish to expand Roots' capability to capture a more diverse range of performance metrics in our future work.

We implement Roots pods as standalone Java server processes. Threads are used to run benchmarkers, anomaly detectors and handlers concurrently within each pod. Pods communicate with ElasticSearch via a web API, and many of the data analysis tasks such as filtering and aggregation are performed in ElasticSearch itself. Some of the more sophisticated statistical analysis tasks (e.g. change point detection and linear regression as described below) are implemented in the R language, and the Roots pods integrate with R using the Rserve protocol [19].

4.1 SLO-violating Anomalies

As described previously, Roots defines anomalies as performance events that trigger SLO violations. Thus, we devise a detector to automatically identify when a SLO violation has occurred. This anomaly detector allows application developers to specify simple performance SLOs for deployed applications. A performance SLO consists of an upper bound on the application response time (T), and the probability (p) that the application response time falls under the specified upper bound. A general performance SLO can be stated as: "application responds under T milliseconds p% of the time".

When enabled for a given application, this anomaly detector starts a benchmarking process that periodically measures the response time of the target application. Probes made by the benchmarking process are several seconds apart in time (sampling rate), so as to not strain the application with load. The detector then periodically analyzes the collected response time measurements to check if the application meets the specified performance SLO. Whenever it detects that the application has failed to meet the SLO, it triggers an anomaly event. The SLO-based anomaly detector supports following configuration parameters:

- Performance SLO: Response time upper bound (T), and the probability (p).
- Sampling rate: Rate at which the target application is benchmarked.
- Analysis rate: Rate at which the anomaly detector checks whether the application has failed to meet the SLO.
- Minimum samples: Minimum number of samples to collect before checking for SLO violations.
- Window size: Length of the sliding window (in time) to consider when checking for SLO violations. This acts as a limit on the number of samples to keep in memory. This has to be large enough so that each analysis cycle has enough data points to calculate results with statistical significance.

Together, the window size and sampling rate impose an upper bound on the amount of data that needs to be kept in memory for calculations. Analysis rate governs how often the collected data is aggregated. Cloud administrators and application developers can tune these parameters to meet their specific accuracy and capacity goals.

In order to prevent the detector from needlessly reporting the same anomaly multiple times, we purge all the data from anomaly detector's sliding window whenever it detects an SLO violation. Therefore, the detector cannot check for further SLO violations until it repopulates the sliding window with the minimum number of samples. This implies that each anomaly is followed by a "warm up" period. For instance, with a sampling rate of 15 seconds, and a minimum samples count of 100, the warm up period can last up to 25 minutes.

4.2 Path Distribution Analysis

We have implemented a path distribution analyzer in Roots whose function it is to identify recurring sequences of PaaS kernel invocations made by an application. Each identified sequence corresponds to a path of execution through the application code (i.e. a path through the control flow graph of the application). This detector is able to determine the frequency with which each path is executed over time. Then, using this information which we term a "path distribution," it reports an anomaly event when the distribution of execution paths changes.

For each application, a path distribution is comprised of the set of execution paths available in that application, along with the proportion of requests that executed each path. It is an indicator of the type of request workload handled by an application. For example, consider a data management application that has a read-only execution path, and a read-write execution path. If 90% of the requests execute the read-only path, and the remaining 10% of the requests execute the read-write path, we may characterize the request workload as read-heavy.

Roots path distribution analyzer facilitates computing the path distribution for each application with no static analysis, by only analyzing the runtime data gathered from the applications. It periodically computes the path distribution for a given application. If it detects that the latest path distribution is significantly different from the distributions seen in the past, it triggers an event. This is done by computing the mean request proportion for each path (over a sliding window of historical data), and then comparing the latest request proportion values against the means. If the latest proportion is off by more than n standard deviations from its mean, the detector considers it to be an anomaly. The sensitivity of the detector can be configured by changing the value of n, which defaults to 2.

Path distribution analyzer enables developers to know when the nature of their application request workload changes. For example in the previous data management application, if suddenly 90% of the requests start executing the read-write path, the Roots path distribution analyzer will detect the change. Similarly it is also able to detect when new paths of execution are being invoked by requests (a form of novelty detection).

4.3 Workload Change Analyzer

Performance anomalies can arise either due to bottlenecks in the cloud platform or changes in the application workload. When Roots detects a performance anomaly (i.e. an application failing to meet its performance SLO), it needs to be able to determine whether the failure is due to an increase in workload or a bottleneck that has suddenly manifested. Roots employs a workload change analyzer to detect workload changes. This Roots component is implemented as an anomaly handler, which gets executed every time an anomaly detector identifies a performance anomaly. Note that this is different from the path distribution analyzer, which is implemented as an anomaly detector. While the path distribution analyzer looks for changes in the *type* of the workload, the workload change analyzer looks for changes in the workload *size* or *rate*.

Workload change analyzer uses change point detection algorithms to analyze the historical trend of the application workload. We use the "number of requests per unit time" as the metric of workload size. Our implementation of Roots supports a number of well known change point detection algorithms (PELT [20], binary segmentation and CL method [21]), any of which can be used to detect level shifts in the workload size. Algorithms like PELT favor long lasting shifts (plateaus) in the workload trend, over momentary spikes. We expect momentary spikes to be fairly common in workload data. But it is the plateaus that cause request buffers to fill up, and consume server-side resources for extended periods of time, thus causing noticeable performance anomalies.

4.4 Bottleneck Identification

Applications running in the cloud consist of user code executed in the application server, and remote service calls to various PaaS kernel services. An AppScale cloud consists of the same kernel services present in the Google App Engine public cloud (datastore, memcache, urlfetch, blobstore, user management etc.). We consider each PaaS kernel invocation, and the code running on the application server as separate *components*. Each application request causes one or more components to execute, and any one of the components can become a bottleneck to cause performance anomalies. The purpose of bottleneck identification is to find, out of all the components executed by an application, the one component that is most likely to have caused application performance to deteriorate.

Suppose an application makes n PaaS kernel invocations ($X_1, X_2, ...X_n$) for each request. For any given application request, Roots captures the time spent on each kernel invocation ($T_{X_1}, T_{X_2}, ...T_{X_n}$), and the total response time (T_{total}) of the request. These time values are related by the formula $T_{total} = T_{X_1} + T_{X_2} + ... + T_{X_n} + r$, where r is all the time spent in the resident application server executing user code (i.e. the time spent not executing PaaS kernel services). r is not directly measured in Roots, since that requires code instrumentation. However, in previous work [15] we showed that typical PaaS-hosted web applications spend most of their time invoking PaaS kernel services. We make use of these findings, and assert that for typical, well-designed PaaS applications $r \ll T_{X_1} + T_{X_2} + ... + T_{X_n}$.

Roots bottleneck identification mechanism first selects up to four components as possible candidates for the bottleneck. These candidates are then further evaluated by a weighted algorithm to determine the actual bottleneck in the cloud platform.

4.4.1 Relative Importance of PaaS Kernel Invocations

The purpose of this metric is to find the component that is contributing the most towards the variance in the total response time. We select a window W in time which includes a sufficient number of application requests, and ending at the point when the performance anomaly was detected. Note that for each application request in W, we can fetch the total response time (T_{total}), and the time spent on individual PaaS kernel services (T_{X_n}) from the Roots data storage. Then we take all the T_{total} values and the corresponding T_{X_n} values in W, and fit a linear model of the form $T_{total} = T_{X_1} + T_{X_2} + ... + T_{X_n}$ using linear regression. Here we leave r out deliberately, since it is typically and ideally small.

Occasionally in AppScale, we observe a request where r is large relative to T_{X_n}. Often these events are correlated with large T_{X_n} values as well leading us to suspect that the effect may be due to an issue with the AppScale infrastructure (e.g. a major garbage collection event in the PaaS software). Overall, Roots detects these events and identifies them correctly (cf subsections 4.4.3 and 4.4.4 below), but they perturb the linear regression model. To prevent that, we filter out requests where the r value is too high. This is done by computing the mean (μ_r) and standard deviation (σ_r) of r over the selected window, and removing any requests where $r > \mu_r + 1.65\sigma_r$.

Once the regression model has been computed, we run a relative importance algorithm [22] to rank each of the regressors (i.e. T_{X_n} values) based on their contribution to the variance of T_{total}. We use the LMG method [23] which is resistant to multicollinearity, and provides a break down of the R^2 value of the regression according to how strongly each regressor influences the variance of the dependent variable. The relative importance values of the regressors add up to the R^2 of the linear regression. We consider $1 - R^2$ (the portion of variance in T_{total} not explained by the PaaS kernel invocations) as the relative importance of r. The component associated with the highest ranked regressor is chosen as a bottleneck candidate. Statistically, this is the component that causes the application response time to vary the most.

4.4.2 Changes in Relative Importance

Next we divide the time window W into equal-sized segments, and compute the relative importance metrics for regressors within each segment. We also compute the relative importance of r within each segment. This way we can obtain a time series of relative importance values for each regressor and r. These time series represent how the relative importance of each component has changed over time.

We subject each relative importance time series to change point analysis to detect if the relative importance of any particular variable has increased recently. If such a variable can be found, then the component associated with that variable is also a potential candidate for the bottleneck. The candidate selected by this method represents a component whose performance has been stable in the past, and has become variable recently.

4.4.3 High Quantiles

Next we analyze the individual distributions of T_{X_n} and r. Out of all the available distributions we find the one whose quantile values are the largest. Specifically, we compute a high quantile (e.g. 0.99 quantile) for each distribution. The component, whose distribution contains the largest quantile value is chosen as another potential candidate for the bottleneck. This component can be considered having a high latency in general.

Faulty PaaS Service	L_1 (30ms)	L_2 (35ms)	L_3 (45ms)
datastore	18	11	10
user management	19	15	10

TABLE 1
Number of anomalies detected in guestbook app under different SLOs (L_1, L_2 and L_3) when injecting faults into two different PaaS kernel services.

4.4.4 Tail End Values

Finally, Roots analyzes each T_{X_k} and r distribution to identify the one with the largest tail values with respect to a particular high quantile. For each maximum (tail end) latency value t, we compute the metric P_t^q as the percentage difference between t and a target quantile q of the corresponding distribution. We set q to 0.99 in our experiments. Roots selects the component with the distribution that has the largest P_t^q as another potential bottleneck candidate. This method identifies candidates that contain rare, high-valued outliers (point anomalies) in their distributions.

4.4.5 Selecting Among the Candidates

The above four methods may select up to four candidate components for the bottleneck. We designate the candidate chosen by a majority of methods as the actual bottleneck. Ties are broken by assigning more priority to the candidate chosen by the relative importance method.

5 Results

We evaluate the efficacy of Roots as a performance monitoring and root cause analysis system for PaaS applications. To do so, we consider its ability to identify and characterize SLO violations. For violations that are not caused by a change in workload, we evaluate Roots' ability to identify the PaaS component that is the cause of the performance anomaly. We also evaluate the Roots path distribution analyzer, and its ability to identify execution paths along with changes in path distributions. Finally, we investigate the performance and scalability of the Roots prototype.

5.1 Anomaly Detection: Accuracy and Speed

To begin the evaluation of the Roots prototype we experiment with the SLO-based anomaly detector, using a simple HTML-producing Java web application called "guestbook". This application allows users to login, and post comments. It uses the AppScale datastore service to save the posted comments, and the AppScale user management service to handle authentication. Each request processed by guestbook results in two PaaS kernel invocations – one to check if the user is logged in, and another to retrieve the existing comments from the datastore. We conduct all our experiments on a single node AppScale cloud except where specified. The node itself is an Ubuntu 14.04 VM with 4 virtual CPU cores (clocked at 2.4GHz) and 4GB of memory.

We run the SLO-based anomaly detector on guestbook with a sampling rate of 15 seconds, an analysis rate of 60 seconds, and a window size of 1 hour. We set the minimum sample count to 100, and run a series of experiments with different SLOs on the guestbook application. Specifically, we fix the SLO success probability at 95%, and set the response time upper bound to $\mu_g + n\sigma_g$. μ_g and σ_g represent the mean and standard deviation of the guestbook's response time. We learn these two parameters apriori by benchmarking the application. Then we obtain three different upper bound values for the guestbook's response time by setting n to 2, 3 and 5 and denote the resulting three SLOs L_1, L_2 and L_3 respectively.

We also inject performance faults into AppScale by modifying its code to cause the datastore service to be slow to respond. This fault injection logic activates once every hour, and slows down all datastore invocations by 45ms over a period of 3 minutes. We chose 45ms because it is equal to $\mu_g + 5\sigma_g$ for the AppScale deployment under test. Therefore this delay is sufficient to violate all three SLOs used in our experiments. We run a similar set of experiments where we inject faults into the user management service of AppScale. Each experiment is run for a period of 10 hours.

Table 1 shows how the number of anomalies detected by Roots in a 10 hour period varies when the SLO is changed. The number of anomalies drops noticeably when the response time upper bound is increased. When the L_3 SLO (45ms) is used, the only anomalies detected are the ones caused by our hourly fault injection mechanism. As the SLO is tightened by lowering the upper bound, Roots detects additional anomalies. These additional anomalies result from a combination of injected faults, and other naturally occurring faults in the system. That is, Roots detected some naturally occurring faults (temporary spikes in application latency), when a number of injected faults were still in the sliding window of the anomaly detector. Together these two types of faults caused SLO violations, usually several minutes after the fault injection period has expired.

Next we analyze how fast Roots can detect anomalies in an application. We first consider the performance of guestbook under the L_1 SLO while injecting faults into the datastore service. Figure 4 shows anomalies detected by Roots as events on a time line. The horizontal axis represents passage of time. The red arrows indicate the start of a fault injection period, where each period lasts up to 3 minutes. The blue arrows indicate the Roots anomaly detection events. Note that every fault injection period is immediately followed by an anomaly detection event, implying near real time reaction from Roots, except in case of the fault injection window at 20:00 hours. Roots detected another naturally occurring anomaly (i.e. one that we did not explicitly inject but nonetheless caused an SLO violation) at 19:52 hours, which caused the anomaly detector to go into the warm up mode. Therefore Roots did not immediately react to the faults injected at 20:00 hours. But as soon as the detector became active again at 20:17, it detected the anomaly.

Figure 5 shows the anomaly detection time line for the same application and SLO, while faults are being injected into the user management service. Here too we see that Roots detects anomalies immediately following each fault injection window.

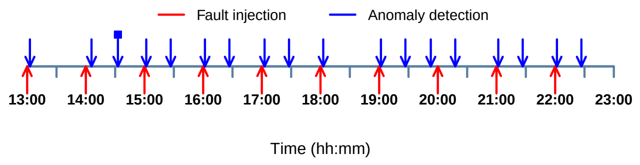

Fig. 4. Anomaly detection in guestbook application during a period of 10 hours. Red arrows indicate fault injection at the datastore service. Blue arrows indicate all anomalies detected by Roots during the experimental run.

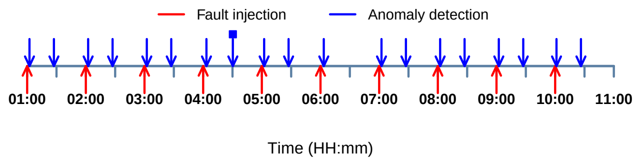

Fig. 5. Anomaly detection in guestbook application during a period of 10 hours. Red arrows indicate fault injection at the user management service. Blue arrows indicate all anomalies detected by Roots during the experimental run.

In all of our experiments, Roots detected the injected anomalies in 158 seconds on average with a maximum time to detection of 289 seconds (i.e. less than 5 minutes). This duration can be further controlled by changing the analysis rate and window size of the detectors.

5.2 Path Distribution Analyzer: Accuracy and Speed

Next we evaluate the effectiveness and accuracy of the path distribution analyzer. For this we employ two different applications.

key-value store This application provides the functionality of an online key-value store. It allows users to store

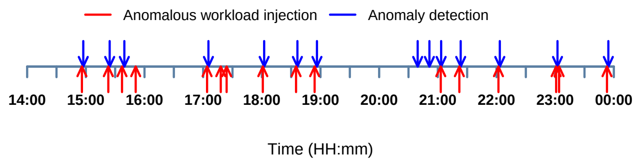

Fig. 6. Anomaly detection in key-value store application during a period of 10 hours. Steady-state traffic is read-heavy. Red arrows indicate injection of write-heavy bursts. Blue arrows indicate all the anomalies detected by the path distribution analyzer.

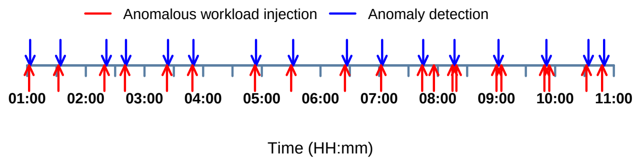

Fig. 7. Anomaly detection in cached key-value store application during a period of 10 hours. Steady-state traffic is mostly served from the cache. Red arrows indicate injection of cache-miss bursts. Blue arrows indicate all the anomalies detected by the path distribution analyzer.

data objects in the cloud where each object is given a unique key. The objects can then be retrieved, updated or deleted using their keys. Different operations (create, retrieve, update and delete) are implemented as separate paths of execution in the application.

cached key-value store This is a simple extension of the regular key-value store, which adds caching to the read operation using the AppScale's memcache service. The application contains separate paths of execution for cache hits and cache misses.

We first deploy the key-value store on AppScale, and populate it with a number of data objects. Then we run a test client against it which generates a read-heavy workload. On average this workload consists of 90% read requests and 10% write requests. The test client is also programmed to randomly send bursts of write-heavy workloads. These bursts consist of 90% write requests on average, and each burst lasts up to 2 minutes. Figure 6 shows the write-heavy bursts as events on a time line (indicated by red arrows). Note that almost every burst is immediately followed by an anomaly detection event (indicated by blue arrows). The only time we do not see an anomaly detection event is when multiple bursts are clustered together in time (e.g. 3 bursts between 17:04 and 17:24 hours). In this case Roots detects the very first burst, and then goes into the warm up mode to collect more data. Between 20:30 and 21:00 hours we also had two instances where the read request proportion dropped from 90% to 80% due to random chance. Roots identified these two incidents also as anomalous.

We conduct a similar experiment using the cached key-value store. Here, we run a test client that generates a workload that is mostly served from the cache. This is done by repeatedly executing read requests on a small selected set of object keys. However, the client randomly sends bursts of traffic requesting keys that are not likely to be in the application cache, thus resulting in many cache misses. Each burst lasts up to 2 minutes. As shown in figure 7, Roots path distribution analyzer correctly detects the change in the workload (from many cache hits to many cache misses), nearly every time the test client injects a burst of traffic that triggers the cache miss path of the application. The only exception is when multiple bursts are clumped together, in which case only the first raises an alarm in Roots.

5.3 Workload Change Analyzer Accuracy

Next we evaluate the Roots workload change analyzer. In this experiment we run a varying workload against the key-value store application for 10 hours. The load generating client is programmed to maintain a mean workload level of 500 requests per minute. However, the client is also programmed to randomly send large bursts of traffic at times of its choosing. During these bursts the client may send more than 1000 requests a minute, thus impacting the performance of the application server that hosts the key-value store. Figure 8 shows how the application workload has changed over time. The workload generator has produced 6 large bursts of traffic during the period of the experiment, which appear as tall spikes in the plot. Note that each burst is immediately followed by a Roots anomaly detection event (shown by red dashed lines). In each of these 6 cases, the increase in workload caused a violation of the application performance SLO. Roots detected the corresponding anomalies, and determined them to be caused by changes in the workload size. As a result, bottleneck identification was not triggered for any of these anomalies. Even though the bursts of traffic appear to be momentary spikes, each burst lasts for 4 to 5 minutes thereby causing a lasting impact on the application performance.

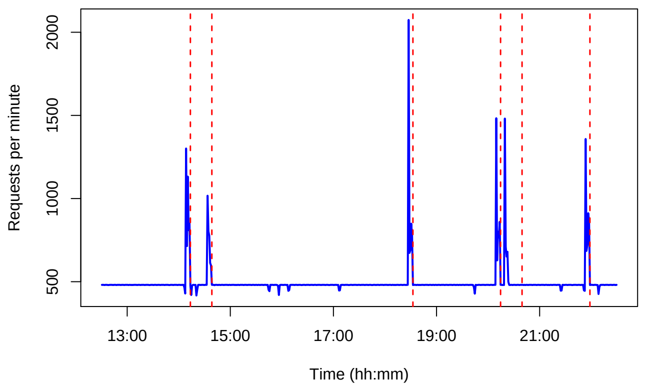

Fig. 8. Workload size over time for the key-value store application. The test client randomly sends large bursts of traffic causing the spikes in the plot. Roots anomaly detection events are shown in red dashed lines.

5.4 Bottleneck Identification Accuracy

Next we evaluate the bottleneck identification capability of Roots. We first discuss the results obtained using the guestbook application, and follow with results obtained using a more complex application. In the experimental run illustrated in figure 4, Roots determined that all the detected anomalies except for one were caused by the AppScale datastore service. This is consistent with our expectations since in this experiment we artificially inject faults into the datastore. The only anomaly that is not traced back to the datastore service is the one that was detected at 14:32 hours. This is indicated by the blue arrow with a small square marker at the top. For this anomaly, Roots concluded that the bottleneck is the local execution at the application server (r). We have verified this result by manually inspecting the AppScale logs and traces of data collected by Roots. As it turns out, between 14:19 and 14:22 the application server hosting the guestbook application experienced some problems, which caused request latency to increase significantly.

Similarly, in the experiment shown in figure 5, Roots determined that all the anomalies are caused by the user management service, except in one instance. This is again inline with our expectations since in this experiment we inject faults into the user management service. For the anomaly detected at 04:30 hours, Roots determined that local execution time is the primary bottleneck. Like earlier, we have manually verified this diagnosis to be accurate.

In order to evaluate how the bottleneck identification performs when an application makes more than 2 PaaS kernel invocations, we conduct another experiment using an application called "stock-trader". This application allows setting up organizations, and simulating trading of stocks

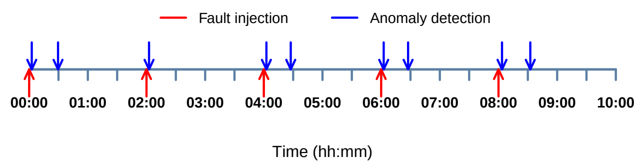

Fig. 9. Anomaly detection in stock-trader application during a period of 10 hours. Red arrows indicate fault injection at the 1st datastore query. Blue arrows indicate all anomalies detected by Roots during the experimental run.

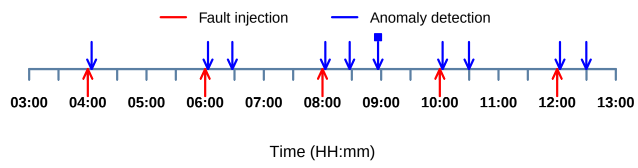

Fig. 10. Anomaly detection in stock-trader application during a period of 10 hours. Red arrows indicate fault injection at the 2nd datastore query. Blue arrows indicate all anomalies detected by Roots during the experimental run.

between the organizations. The two main operations in this application are *buy* and *sell*. Each of these operations makes 8 calls to the AppScale datastore. According to our previous work [15], 8 kernel invocations in the same path of execution is very rare in web applications developed for a PaaS cloud. The probability of finding an execution path with more than 5 kernel invocations in a sample of PaaS-hosted applications is less than 1%. Therefore the stock-trader application is a good extreme case example to test the Roots bottleneck identification support. We execute a number of experimental runs using this application, and here we present the results from two of them. In all experiments we configure the anomaly detector to check for the response time SLO of 177ms with 95% success probability.

In one of our experimental runs we inject faults into the first datastore query executed by the buy operation of stock-trader. The fault injection logic runs every two hours, and lasts for 3 minutes. The duration of the full experiment is 10 hours. Figure 9 shows the resulting event sequence. Note that every fault injection event is immediately followed by a Roots anomaly detection event. There are also four additional anomalies in the time line which were SLO violations caused by a combination of injected faults, and naturally occurring faults in the system. For all the anomalies detected in this test, Roots correctly selected the first datastore call in the application code as the bottleneck. The additional four anomalies occurred because a large number of injected faults were in the sliding window of the detector. Therefore, it is accurate to attribute those anomalies also to the first datastore query of the application.

Figure 10 shows the results from a similar experiment where we inject faults into the second datastore query executed by the operation. Here also Roots detects all the artificially induced anomalies along with a few extras. All the anomalies, except for one, are determined to be caused by the second datastore query of the buy operation. The anomaly detected at 08:56 (marked with a square on top of

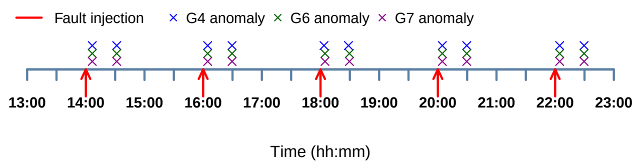

Fig. 11. Anomaly detection in 8 applications deployed in a clustered AppScale cloud. Red arrows indicate fault injection at the datastore service for queries generated from a specific host. Cross marks indicate all the anomalies detected by Roots during the experiment.

the blue arrow) is attributed to the fourth datastore query executed by the application. We have manually verified this diagnosis to be accurate.

In the experiments illustrated in figures 4, 5, 9, and 10 we maintain the application request rate steady throughout the 10 hour periods. Therefore, the workload change analyzer of Roots did not detect any significant shifts in the workload level. Consequently, none of the anomalies detected in these 4 experiments were attributed to a workload change. The bottleneck identification was therefore triggered for each anomaly.

To evaluate the agreement level among the four bottleneck candidate selection methods, we analyze 407 anomalies detected by Roots over a period of 3 weeks. We see that except on 13 instances, in all the remaining cases 2 or more candidate selection methods agreed on the final bottleneck component chosen. This implies that most of the time (96.8%) Roots identifies bottlenecks with high confidence.

5.5 Multiple Applications in a Clustered Setting

To demonstrate how Roots can be used in a multi-node environment, we set up an AppScale cloud on a cluster of 10 virtual machines (VMs). VMs are provisioned by a Eucalyptus (IaaS) cloud, and each VM is comprised of 2 CPU cores and 2GB memory. Then we proceed to deploy 8 instances of the guestbook application on AppScale. We use the multitenant support in AppScale to register each instance of guestbook as a different application ($G1$ through $G8$). Each instance is hosted on a separate application server instance, has its own private namespace on the AppScale datastore, and can be accessed via a unique URL. We disable auto-scaling support in the AppScale cloud, and inject faults into the datastore service of AppScale in such a way that queries issued from a particular VM, are processed with a 100ms delay. We identify the VM by its IP address in our test environment, and shall refer to it as V_f in the discussion. We trigger the fault injection every 2 hours, and when activated it lasts for up to 5 minutes. Then we monitor the applications using Roots for a period of 10 hours. Each anomaly detector is configured to check for the 75ms response time SLO with 95% success rate. ElasticSearch, Logstash and the Roots pod are deployed on a separate VM.

Figure 11 shows the resulting event sequence. Note that we detect anomalies in 3 applications ($G4$, $G6$ and $G7$) immediately after each fault injection. Inspecting the topology of our AppScale cluster revealed that these were the only 3 applications that were hosted on V_f. As a result, the bi-hourly fault injection caused their SLOs to get violated.

Feature	Results Observed in Roots
Detecting anomalies	All the artificially induced anomalies were detected, except when multiple anomalies are clustered together in time. In that case only the first anomaly was detected. Roots also detected several anomalies that occurred due to a combination of injected faults, and natural faults.
Characterizing anomalies as being due to workload changes or bottlenecks	When anomalies were induced by varying the application workload, Roots correctly determined that the anomalies were caused by workload changes. In all other cases we kept the workload steady, and hence the anomalies were attributed to a system bottleneck.
Identifying correct bottleneck	In all the cases where bottleneck identification was performed, Roots correctly identified the bottleneck component.
Reaction time	All the artificially induced anomalies (SLO violations) were detected as soon as enough samples of the fault were taken by the benchmarking process (2-5 minutes from the start of the fault injection period).
Path distribution	All the artificially induced changes to the path distribution were detected.

TABLE 2
Summary of Roots efficacy results.

Other applications did not exhibit any SLO violations since we are monitoring against a very high response time upper bound. In each case Roots detected the SLO violations 2-3 minutes into the fault injection period. As soon as that happened, the anomaly detectors of $G4$, $G6$ and $G7$ entered the warmup mode. But our fault injection logic kept injecting faults for at least 2 more minutes. Therefore when the anomaly detectors reactivated after 25 minutes (time to collect the minimum sample count), they each detected another SLO violation. As a result, we see another set of detection events approximately half an hour after the fault injection events.

We conclude our discussion of Roots efficacy with a summary of our results. Table 2 provides an overview of all the results presented so far, broken down into four features that we wish to see in an anomaly detection and bottleneck identification system.

5.6 Roots Performance Overhead and Scalability

Next we evaluate the performance overhead incurred by Roots on the applications deployed in the cloud platform. We are particularly interested in understanding the overhead of recording the PaaS kernel invocations made by each application, since this feature requires some changes to the PaaS kernel implementation. We deploy a number of applications on a vanilla AppScale cloud (with no Roots), and measure their request latencies. We use the popular Apache Bench tool to measure the request latency under a varying number of concurrent clients. We then take the same measurements on an AppScale cloud with Roots, and compare the results against the ones obtained from the vanilla AppScale cloud. In both environments we disable the autoscaling support of AppScale, so that all client requests are served from a single application server instance. In our prototype implementation of Roots, the kernel invocation events get buffered in the application server before they are sent to the Roots data storage. We wish to explore how this feature performs when the application server is under heavy load.

	Without Roots		With Roots	
App./Concurrency	Mean (ms)	SD	Mean (ms)	SD
guestbook/1	12	3.9	12	3.7
guestbook/50	375	51.4	374	53
stock-trader/1	151	13	145	13.7
stock-trader/50	3631	690.8	3552	667.7
kv store/1	7	1.5	8	2.2
kv store/50	169	26.7	150	25.4
cached kv store/1	3	2.8	2	3.3
cached kv store/50	101	24.8	97	35.1

TABLE 3
Latency comparison of applications when running on a vanilla AppScale cloud vs when running on a Roots-enabled AppScale cloud.

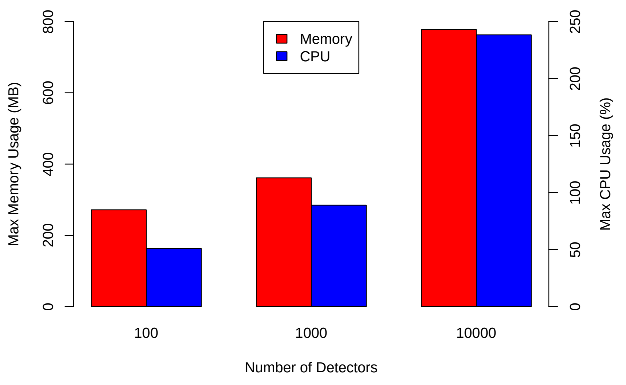

Fig. 12. Resource utilization of a Roots pod.

Table 3 shows the comparison of request latencies. We discover that Roots does not add a significant overhead to the request latency in any of the scenarios considered. In all the cases, the mean request latency when Roots is in use, is within one standard deviation from the mean latency when Roots is not in use. The latency increases with the number of concurrent clients (since all requests are handled by a single application server), but still there is no evidence of any detrimental overhead from Roots even under load. This is due to the asynchronous nature of Roots, which buffers monitoring events in memory, and reports them to ElasticSearch out of the request processing flow.

Finally, to demonstrate how lightweight and scalable Roots is, we deploy a Roots pod on a virtual machine with 4 CPU cores and 4GB memory. To simulate monitoring multiple applications, we run multiple concurrent anomaly detectors in the pod. Each detector is configured with a 1 hour sliding window. We vary the number of concurrent detectors between 100 and 10000, and run each configuration for 2 hours. We track the memory and CPU usage of the pod during each of these runs using the jstat and pidstat tools.

Figure 12 illustrates the maximum resource utilization of the Roots pod for different counts of concurrent anomaly detectors. We see that with 10000 concurrent detectors, the maximum CPU usage is 238%, where 400% is the available limit for 4 CPU cores. The maximum memory usage in this case is only 778 MB. Since each anomaly detector operates with a fixed-sized window, and they bring additional data into memory only when required, the memory usage of the Roots pod generally stays low. We also experimented with larger concurrent detector counts, and we were able to pack up to 40000 detectors into the pod before getting constrained by the CPU capacity of our VM. This result implies that we can monitor tens of thousands of applications using a single pod, thereby scaling up to a very large number of applications using only a handful of pods.

6 Related Work

Roots falls into the category of performance anomaly detection and bottleneck identification (PADBI) systems. PADBI systems observe, in real time, the performance behaviors of a running system or application, collecting vital measurements at discrete time intervals to create baseline models of typical system behaviors [7]. Such systems play a crucial role in achieving guaranteed service reliability, performance and quality of service by detecting performance issues in a timely manner before they escalate into major outages or SLO violations [24]. PADBI systems are thoroughly researched, and well understood in the context of traditional standalone and network applications. Many system administrators are familiar with frameworks like Nagios [25], Open NMS [26] and Zabbix [27] which can be used to collect data from a wide range of applications and devices.

However, the paradigm of cloud computing, being relatively new, is yet to be fully penetrated by PADBI systems research. The size, complexity and the dynamic nature of cloud platforms make performance monitoring a particularly challenging problem. The existing technologies like Amazon CloudWatch [28], New Relic [4] and DataDog [6] facilitate monitoring cloud applications by instrumenting low level cloud resources (e.g. virtual machines), and application code. But such technologies are either impracticable or insufficient in PaaS clouds where the low level cloud resources are hidden under layers of managed services, and the application code is executed in a sandboxed environment that is not always amenable to instrumentation. When code instrumentation is possible, it tends to be burdensome, error prone, and detrimental to the application's performance. Roots on the other hand is built into the fabric of the PaaS cloud giving it full visibility into all the activities that take place in the entire software stack.

Our work borrows heavily from the past literature [3], [7] that detail the key features of cloud APMs. Ibidunmoye et al highlight the importance of multilevel bottleneck identification as an open research question [7]. This is the ability to identify bottlenecks from a set of top-level application service components, and further down through the virtualization layer to system resource bottlenecks. We detail our early investigations into doing so in [9]. The work herein expands upon both the technical detail and empirical evaluation of this initial work. We also present a novel mechanism for detecting changes in application workload patterns by analyzing the request execution paths. Roots is unique in that it supports identifying execution paths and performance bottlenecks using only the set of services provided by the PaaS kernel.

Similar to systems like X-Trace [29] and PinPoint [30], Roots also tags request messages in order to trace their flow through a complex distributed system. X-Trace records network activities across protocols and layers, but does not support root cause analysis. PinPoint traces interactions among J2EE middleware components to localize faults. Roots on the other hand traces the PaaS kernel service calls made by an application while processing requests.

Cherkasova et al developed an online performance modeling technique to detect anomalies in traditional transaction processing systems [31]. They divide time into contiguous segments, such that within each segment the application workload (volume and type of transactions) and resource usage (CPU) can be fit to a linear regression model. Segments for which a model cannot be found, are considered anomalous. Then they remove anomalous segments from the history, and perform model reconciliation to differentiate between workload changes and application problems. While this method is powerful, it requires instrumenting application code to detect different external calls (e.g. database queries) executed by the application. Since the model uses different transaction types as parameters, some prior knowledge regarding the transactions also needs to be fed into the system. The algorithm is compute intensive due to the need for continuous segmentation and model fitting.

Dean et al implemented PerfCompass [32], an anomaly detection and localization method for IaaS clouds. They instrument the VM operating system kernels to capture the system calls made by user applications. Anomalies are detected by looking for unusual increases in system call execution time. They group system calls into execution units (processes, threads etc), and analyze how many units are affected by any given anomaly. Based on this metric they conclude if the problem was caused by a workload change or an application level issue. We take a similar approach in Roots, in that we capture the PaaS kernel invocations made by user applications.

Nguyen et al presented PAL, another anomaly detection and localization mechanism targeting distributed applications deployed on IaaS clouds [33]. Similar to Roots, they also use an SLO monitoring approach to detect application performance anomalies. When an anomaly is detected, they perform change point analysis on gathered resource usage data (CPU, memory and network) to identify the anomaly onset time.

Magalhaes and Silva have made significant contributions in the area of anomaly detection and root cause analysis in web applications [34], [35]. They compute the correlation between application workload and latency. If the level of correlation drops significantly, they consider it to be an anomaly. A similar correlation analysis between workload and other local system metrics (e.g. CPU and memory usage) is used to identify the system resource that is responsible for a given anomaly. They also use an aspect-oriented programming model in their target applications, which allows them to easily instrument application code, and gather metrics regarding various remote services (e.g. database) invoked by the application. This data is subjected to a series of simple linear regressions to perform root cause analysis. This approach assumes that remote services are independent of each other. However, in a cloud platform where kernel services are deployed in the same shared infrastructure, this assumption might not hold true. Therefore we improve on their methodology, and use multiple linear regression with relative importance to identify cloud platform bottlenecks. Relative importance is resistant to multicollinearity, and therefore does not require the independence assumption.

Anomaly detection is a general problem not restricted to performance analysis. Researchers have studied anomaly detection from many different points of view, and as a result many viable algorithms and solutions have emerged over time [36]. Prior work in performance anomaly detection and root cause analysis can be classified as statistical methods (e.g. [33], [35], [37], [38]) and machine learning methods (e.g. [39], [40], [41]). While we use many statistical methods in our work (change point analysis, relative importance, quantile analysis), Roots is not tied to any of these techniques. Rather, we provide a framework on top of which new anomaly detectors and anomaly handlers can be built.

7 Conclusions and Future Work

As the paradigm of cloud computing grows in popularity, the need for monitoring cloud-hosted applications is becoming critical. Application developers and cloud administrators wish to detect performance anomalies in cloud applications, and perform root cause analysis to diagnose problems. However, the high level of abstraction provided by cloud platforms, coupled with their scale and complexity, makes performance diagnosis a daunting problem.

In this paper, we present Roots, an efficient and accurate monitoring framework for applications deployed in a PaaS cloud. Roots is designed to function as a curated service built into the cloud platform. It relieves the application developers from having to configure their own monitoring solutions, or instrument application code. Roots captures runtime data from all the different layers involved in processing application requests. It correlates events across PaaS layers and identifies bottlenecks across the PaaS stack.

Roots monitors applications for compliance with service level objectives (SLOs) and detects anomalies via SLO violations. When Roots detects an anomaly, it analyzes workload data and application runtime data to perform root cause analysis. Roots is able to determine whether a particular anomaly was caused by a change in the application workload, or due to a bottleneck in the cloud platform. Our workload change point detection algorithm distinguishes between different paths of execution though an application. Our bottleneck identification algorithm uses a combination of linear regression, quantile analysis, and change point detection to identify the PaaS service that is the most likely cause of the anomaly.

We evaluate Roots using a prototype built for the AppScale PaaS. Our results indicate that Roots is effective at detecting workload changes and performance bottlenecks within 5 minutes from when they start and introduces no false positives. Our empirical trials also show that the mean latency of the PaaS platform with Roots is within one standard deviation of the mean latency of the cloud platform without Roots, for the workloads we studied.

In our future work, we plan to expand the data gathering capabilities of Roots into the low level virtual machines and containers that host cloud platform services. We intend to tap into the hypervisors and container managers to harvest runtime data regarding the resource usage (CPU, memory, disk etc.) of application components. With that we expect to extend the root cause analysis support of Roots so that it can not only pinpoint the bottlenecked application components, but also the low level hosts and system resources that constitute each bottleneck.

Finally, we acknowledge and thank our research supporters. This work is funded in part by NSF (CCF-1539586, ACI- 1541215, CNS-1218808, CNS-0905237, ACI-0751315), NIH (1R01EB014877-01), ONR NEEC (N00174-16-C-0020), Huawei Technologies, and the California Energy Commission (PON- 14-304).

References

[1] N. Antonopoulos and L. Gillam, *Cloud Computing: Principles, Systems and Applications*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[2] P. Pinheiro, M. Aparicio, and C. Costa, "Adoption of cloud computing systems," in *Proceedings of the International Conference on Information Systems and Design of Communication*, ser. ISDOC '14. New York, NY, USA: ACM, 2014, pp. 127–131. [Online]. Available: http://doi.acm.org/10.1145/2618168.2618188

[3] G. Da Cunha Rodrigues, R. N. Calheiros, V. T. Guimaraes, G. L. d. Santos, M. B. de Carvalho, L. Z. Granville, L. M. R. Tarouco, and R. Buyya, "Monitoring of cloud computing environments: Concepts, solutions, trends, and future directions," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, ser. SAC '16. New York, NY, USA: ACM, 2016, pp. 378–383. [Online]. Available: http://doi.acm.org/10.1145/2851613.2851619

[4] "New relic: Application performance management and monitoring," 2016, https://newrelic.com [Accessed Sep 2016].

[5] "Dynatrace: Digital performance management and application performance monitoring," 2016, https://www.dynatrace.com [Accessed Sep 2016].

[6] "Datadog: Cloud monitoring as a service," 2016, https://www.datadoghq.com [Accessed Sep 2016].

[7] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification," *ACM Comput. Surv.*, vol. 48, no. 1, pp. 4:1–4:35, Jul. 2015. [Online]. Available: http://doi.acm.org/10.1145/2791120

[8] A. Keller and H. Ludwig, "The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services," *J. Netw. Syst. Manage.*, vol. 11, no. 1, Mar. 2003.

[9] H. Jayathilaka, C. Krintz, and R. Wolski, "Performance monitoring and root cause analysis for cloud-hosted web applications," in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW '17. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2017, pp. 469–478. [Online]. Available: https://doi.org/10.1145/3038912.3052649

[10] C. Krintz, "The appscale cloud platform: Enabling portable, scalable web application deployment," *IEEE Internet Computing*, vol. 17, no. 2, pp. 72–75, March 2013.

[11] SearchCloudComputing, 2015, http://searchcloudcomputing.techtarget.com/feature/Experts-forecast-the-2015-cloud-computing-market [Accessed March 2015].

[12] Forbes, 2016, http://www.forbes.com/sites/louiscolumbus/2016/03/13/roundup-of-cloud-computing-forecasts-and-market-estimates-2016 [Accessed Sep 2016].

[13] "App engine - run your applications on a fully managed paas," 2015, "https://cloud.google.com/appengine" [Accessed March 2015].

[14] "Microsoft azure cloud sdk," https://azure.microsoft.com/en-us/downloads/ [Accessed October 2016].

[15] H. Jayathilaka, C. Krintz, and R. Wolski, "Response time service level agreements for cloud-hosted web applications," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: ACM, 2015, pp. 315–328. [Online]. Available: http://doi.acm.org/10.1145/2806777.2806842

[16] “Elasticsearch - search and analyze data in real time,” 2016, ”https://www.elastic.co/products/elasticsearch” [Accessed Sep 2016].

[17] O. Kononenko, O. Baysal, R. Holmes, and M. W. Godfrey, “Mining modern repositories with elasticsearch,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 328–331. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597091

[18] “Logstash - collect, enrich and transport data,” 2016, ”https://www.elastic.co/products/logstash” [Accessed Sep 2016].

[19] S. Urbanek, “Rserve – a fast way to provide r functionality to applications,” in *Proc. of the 3rd international workshop on Distributed Statistical Computing (DSC 2003)*, 2003.

[20] R. Killick, P. Fearnhead, and I. A. Eckley, “Optimal detection of changepoints with a linear computational cost,” *Journal of the American Statistical Association*, vol. 107, no. 500, pp. 1590–1598, 2012.

[21] C. Chen and L.-M. Liu, “Joint estimation of model parameters and outlier effects in time series,” *Journal of the American Statistical Association*, vol. 88, no. 421, pp. 284–297, 1993.

[22] U. Groemping, “Relative importance for linear regression in r: The package relaimpo,” *Journal of Statistical Software*, vol. 17, no. 1, pp. 1–27, 2006. [Online]. Available: https://www.jstatsoft.org/index.php/jss/article/view/v017i01

[23] G. R. Lindeman R.H., Merenda P.F., *Introduction to Bivariate and Multivariate Analysis*. Scott, Foresman, Glenview, IL, 1980.

[24] Q. Guan, Z. Zhang, and S. Fu, “Proactive failure management by integrated unsupervised and semi-supervised learning for dependable cloud systems,” in *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, Aug 2011, pp. 83–90.

[25] R. C. Harlan, “Network management with nagios,” *Linux J.*, vol. 2003, no. 111, pp. 3–, Jul. 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=860375.860378

[26] “Opennms: Network management platform,” 2016, https://www.opennms.org/en [Accessed Sep 2016].

[27] P. Tader, “Server monitoring with zabbix,” *Linux J.*, vol. 2010, no. 195, Jul. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1883478.1883485

[28] “Amazon cloud watch,” 2016, https://aws.amazon.com/cloudwatch [Accessed Sep 2016].

[29] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, “X-trace: A pervasive network tracing framework,” in *Proceedings of the 4th USENIX Conference on Networked Systems Design &#38; Implementation*, 2007.

[30] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, “Pinpoint: Problem determination in large, dynamic internet services,” in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, 2002.

[31] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, “Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change,” in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, June 2008, pp. 452–461.

[32] D. J. Dean, H. Nguyen, P. Wang, and X. Gu, “Perfcompass: Toward runtime performance anomaly fault localization for infrastructure-as-a-service clouds,” in *Proceedings of the 6th USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud’14. Berkeley, CA, USA: USENIX Association, 2014, pp. 16–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=2696535.2696551

[33] H. Nguyen, Y. Tan, and X. Gu, “Pal: Propagation-aware anomaly localization for cloud hosted distributed applications,” in *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, ser. SLAML ’11. New York, NY, USA: ACM, 2011, pp. 1:1–1:8. [Online]. Available: http://doi.acm.org/10.1145/2038633.2038634

[34] J. P. Magalhaes and L. M. Silva, “Detection of performance anomalies in web-based applications,” in *Proceedings of the 2010 Ninth IEEE International Symposium on Network Computing and Applications*, ser. NCA ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 60–67. [Online]. Available: http://dx.doi.org/10.1109/NCA.2010.15

[35] J. a. P. Magalhães and L. M. Silva, “Root-cause analysis of performance anomalies in web-based applications,” in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC ’11. New York, NY, USA: ACM, 2011, pp. 209–216. [Online]. Available: http://doi.acm.org/10.1145/1982185.1982234

[36] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: A survey,” *ACM Comput. Surv.*, vol. 41, no. 3, pp. 15:1–15:58, Jul. 2009. [Online]. Available: http://doi.acm.org/10.1145/1541880.1541882

[37] G. Casale, N. Mi, L. Cherkasova, and E. Smirni, “Dealing with burstiness in multi-tier applications: Models and their parameterization,” *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1040–1053, Sept 2012.

[38] S. Malkowski, M. Hedwig, J. Parekh, C. Pu, and A. Sahai, “Bottleneck detection using statistical intervention analysis,” in *Proceedings of the Distributed Systems: Operations and Management 18th IFIP/IEEE International Conference on Managing Virtualization of Networks and Services*, ser. DSOM’07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 122–134. [Online]. Available: http://dl.acm.org/citation.cfm?id=1783374.1783389

[39] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase, “Correlating instrumentation data to system states: A building block for automated diagnosis and control,” in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI’04. Berkeley, CA, USA: USENIX Association, 2004, pp. 16–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251270

[40] L. Yu and Z. Lan, “A scalable, non-parametric anomaly detection framework for hadoop,” in *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, ser. CAC ’13. New York, NY, USA: ACM, 2013, pp. 22:1–22:2. [Online]. Available: http://doi.acm.org/10.1145/2494621.2494643

[41] K. Bhaduri, K. Das, and B. L. Matthews, “Detecting abnormal machine characteristics in cloud infrastructures,” in *2011 IEEE 11th International Conference on Data Mining Workshops*. IEEE, 2011, pp. 137–144.

Hiranya Jayathilaka Hiranya Jayathilaka is a fifth year PhD candidate at the computer science department of UC Santa Barbara. His research focuses on automated governance for cloud platforms, which addresses a wide range of issues including policy enforcement, performance guarantees, monitoring and troubleshooting. He received a B.Sc. in engineering from University of Moratuwa, Sri Lanka.

Chandra Krintz Chandra Krintz is a Professor of Computer Science (CS) at UC Santa Barbara and Chief Scientist at AppScale Systems Inc. Chandra holds M.S./Ph.D. degrees in CS from UC San Diego. Chandra’s research interests include programming systems, cloud and big data computing, and the Internet of Things (IoT). Chandra has supervised and mentored over 60 students and has led several educational and outreach programs that introduce young people to computer science.

Rich Wolski Dr. Rich Wolski is a Professor of Computer Science at the UC Santa Barbara, and co-founder of Eucalyptus Systems Inc. Having received his M.S. and Ph.D. degrees from UC Davis (while a research scientist at Lawrence Livermore National Laboratory) he has also held positions at the UC San Diego, and the University of Tennessee, the San Diego Supercomputer Center and Lawrence Berkeley National Laboratory. Rich has led several national scale research efforts in the area of distributed systems, and is the progenitor of the Eucalyptus open source cloud project.