PAD: Performance Anomaly Detection in Multi-Server Distributed Systems

Manjula Peiris, James H. Hill
Department of Computer and Information Science
Indiana University-Purdue University Indianapolis
Indianapolis, IN USA
{tmpeiris, hillj}@cs.iupui.edu

Jorgen Thelin, Sergey Bykov, Gabriel Kliot, Christian Konig
Microsoft Research
Redmond, WA USA
{jthelin, sbykov, gkliot, chrisko}@microsoft.com

***Abstract*—Multi-server distributed systems are becoming increasingly popular with the emergence of cloud computing. These systems need to provide high throughput with low latency, which is a difficult task to achieve. Manual performance tuning and diagnosis of such systems, however, is hard as the amount of relevant performance diagnosis data is large. To help system developers with performance diagnosis, we have developed a tool called Performance Anomaly Detector (*PAD*). *PAD* combines user-driven navigation analysis with automatic correlation and comparative analysis techniques. The combination results in a powerful tool that can help find a number of performance anomalies. Based on our experience in applying *PAD* to the Orleans system, we discovered that *PAD* was able to reduce developer time and effort detecting anomalous performance cases and improve a developer's ability to perform deeper analysis of such behaviors.**

***Index Terms*—Performance Diagnostics, Anomaly Detection, Performance Bottlenecks, Distributed Systems, Orleans.**

I. Introduction

Multi-server distributed systems have become popular with the emergence of cloud computing. Most distributed systems nowadays are built from cost effective off-the-shelf multi-server systems instead of high performance single server systems [1]. While cloud technologies provide seamless scalability, achieving high performance in terms of high throughput and low response time is still challenging and important. Good performance improves user experience and helps attract more users, while bad performance results in dissatisfaction. It is therefore important to continually analyze the performance of a cloud-based distributed system to ensure it is executing within its performance requirements. When the required performance is not provided, it is necessary to identify and resolve the performance bottlenecks in a timely manner.

Diagnosing functional problems in distributed applications is *hard*. Performance diagnosis of distributed systems is *harder* [2], as the system may execute without any functional problems while not achieving expected performance goals. Under-achievements of performance goals can be in the form of low throughput or high latency. In such situations, system execution logs might not contain direct clues (*e.g.*, error messages or exceptions) that can be used as a starting point for analysis. Instead, they usually include performance counters that track different aspects of system performance.

Developers and testers typically analyze the performance counters to find system performance anomalies and reason about performance characteristics. Multi-server systems, however, contain hundreds of servers each constantly generating performance data—making manual analysis error prone and time consuming. It is therefore essential to develop techniques and build automatic tools for performance analysis and diagnostics of large multi-server distributed systems using the performance data generated during execution.

To diagnosis performance problems in distributed multi-server systems, we have developed a tool called *Performance Anomaly Detector (PAD)*. The objectives of *PAD* are: (a) give distributed system developers insights about distributed system performance from collected performance data; (b) minimize developer time required to analyze large amounts of performance data generated across hundreds to thousands of servers; and (c) assist system developers and administrators in troubleshooting performance related issues and finding root causes.

To achieve the above goals *PAD* provides:

1) Summary of distributed system performance data using *visualizations* and *summary statistics*;
2) *Threshold analysis* for performance counters;
3) *Correlation analysis* for automatic detection of relationships between performance counters; and
4) *Comparative analysis* for automatic detection of anomalous performance counters.

The capabilities listed above enable a powerful combination of user-driven navigation analysis and automatic analysis. In user-driven navigation analysis, the person troubleshooting the system applies expert knowledge in a semi-manual process assisted by the tool. When this process does not lead to successful problem resolution, automatic correlation and comparative analysis techniques are used to automatically try to find clues for performance problems.

Orleans. The motivation for developing *PAD* started with our experience diagnosing the performance of the *Orleans system* [3]. Orleans is a programming model and runtime for large-scale distributed cloud computing services. Orleans is based on an *actor programming model*. Actors in Orleans are virtual and isolated computation entities that use asynchronous message passing to communicate. The actor model is suitable for interactive request-reply applications (as opposite to MapReduce [4] style models that are suitable for offline batch processing) and is highly scalable due to the independent nature of actors and their interactions.

Although the main design goal of Orleans is to simplify the programming model for cloud applications while providing scalability and reliability, providing efficiency is not less important for cloud applications that pay for consuming cloud resources. It is thus critical for Orleans to provide good performance. To ensure Orleans and its applications are executing within their performance requirements, it is necessary to continually track its performance. When performance requirements are not met, it is necessary to identify and resolve performance bottlenecks in a timely manner.

The above requirements motivated us to create *PAD*. We have used Orleans performance tests data to validate the applicability of our tool. Current applications of *PAD* show that it is capable of supporting root cause analysis of performance problems in Orleans. It is important to note that the applicability of the techniques we have developed as well as the *PAD* tool itself are not limited to Orleans and can be applied to any multi-server distributed system which generates performance data.

Paper organization. The remainder of this paper is organized as follows: Section II provides background information about performance diagnosis and challenges specific to *PAD*; Section III presents two examples that motivated the development of *PAD*; Section IV presents how *PAD* is used by developers to find performance problems and their root causes; Section V describes *PAD* implementation; Section VI discusses how we applied *PAD* to Orleans; Section VII discusses related work and Section VIII provides conclusion and lessons learned.

II. Background Information and Challenges

This section discusses background information related to our work and introduces the challenges we had to overcome while designing and implementing *PAD*.

A. Background on Performance Diagnosis Techniques

Performance diagnosis of software systems can be divided into two main tasks: performance anomaly detection and root cause analysis.

Anomaly detection. An *anomaly* is defined as deviation from a common rule, arrangement, or form [5]. Two prominent approaches, in literature, have been used to detect software performance anomalies:

1) Detecting anomalies based on performance requirements. This method is used when system performance fails to meet its requirements. Performance requirements can either be specified explicitly via *Service Level Agreements (SLAs)* [6], [7], or implicitly reflect internal implementation details (*e.g.*, a desired average server CPU or size of a message queue).

2) Detecting anomalies based on deviations from normal performance. This method is used when there are no specific performance requirements, but performance deviates from the considered norm. For example, performance results of a software release are significantly worse then a prior release; or performance metrics of a particular server deviate from the performance of other identical servers. In these situations, aggregated values (*e.g.*, mean, median, or standard deviation) of abnormal performance data can be compared against the corresponding data of normal performance [8]. A deviation can be defined based on a threshold value, or developer's domain knowledge of the system.

Root cause analysis. *Root cause analysis* [9] is a process of identifying the reasons for software execution failures. In the context of performance, it is the process of identifying the source of performance anomalies. Some examples of root causes of performance anomalies include deadlocks and starvation [10], mis-configurations [11], and software performance anti-patterns [12].

Performing root cause analysis sometimes requires expert knowledge. For example, some systems may need to be configured in a certain way to achieve optimal performance and only system experts may have the knowledge about those configurations. Likewise, detecting performance anti-patterns may require inspecting the systems's source code, but only someone knowledgeable of the source code can do that. It is therefore important to integrate expert knowledge in a practical tool for root cause analysis. In addition, automatic detection of root causes due to mis-configurations and bad software designs is possible as well [11], [12].

B. Performance Counters

When run in production, multi-server distributed system performance is closely monitored. The collected performance data is stored in execution logs in the form of performance counters [13]. *Performance counters* track specific system states or resources during execution, such as CPU, memory, network, and application/framework specific information. Typical large production multi-server distributed systems run on clusters consisting of hundreds to thousands of servers. Each server periodically (typically every couple of minutes) tracks a large number of counters (hundreds in [13]) and stores them in the log.

In Orleans, a typical deployment consists of tens to hundreds of servers each tracking about 200 counters every five minute. The log is either stored separately for each server in its local file system or in a shared cloud storage, such as Azure Table storage[1]. Table I provides examples of different classes of performance counters in Orleans.

Type	**Examples**
Orleans Runtime	CPU usage, Percentage of time in garbage collection
Message Queues	Lengths of the send and receive message queues
Messaging	Number of total messages sent and received
Actors	Number of actors on a server

TABLE I: Examples of different classes of performance counters in Orleans

[1]http://www.windowsazure.com/en-us/develop/net/how-to-guides/table-services/

C. Challenges in Analyzing Performance Counter Data

The approaches used in *PAD* are based on the above mentioned performance diagnosis techniques. In addition, we have tackled a number of unique challenges in our setting that we detail below.

1) Large data volumes. As already mentioned, multi-server distributed systems generate a large amount of performance data, which is impossible to analyze manually. Navigating the vast amount of data is also *hard* as it is not easy to decide how to slide-and-dice it: (1) what set of performance counters to consider and (2) whether to look at the performance counters across different servers at a particular time, particular server across different times, or both.

Developers sometimes have an idea, or clue, about the source of the problem. In such cases, they can manually inspect the relevant counters. For most performance issues, however, it is *hard* to decide what counters are relevant. Incorrect selection can cost valuable developer time at best and/or lead to wrong conclusions at worse. It is therefore important to inspect the performance counters that are more closely related to the problem under investigation.

2) Insufficient training data. One approach for performance diagnosis is to classify the counters into performance crisis situations, as done in [14], [15]. This kind of classification requires many different datasets and known labels (performance crisis situations) in order to apply machine learning based classification techniques. Such labeled datasets, however, are not always available. For example, although Orleans has been used in several projects within Microsoft, we did not have access to any labeled historical data. Because the labeled performance crisis data was unavailable, we could not apply machine learning classification techniques.

3) Time correlation. A distributed nature of the systems we consider poses a major challenge when correlating data collected from different servers across time. Servers are located on different physical machines, each having a different physical clock that may not be always synchronized. Unfortunately, some performance counters are sensitive to time and therefore even a 1 second approximation may give incorrect results.

III. Motivating Examples in Orleans

The need for *PAD* emerged since the early stages of the Orleans project when we occasionally faced non-trivial bugs that required manually looking through large sets of execution logs—literally searching for a needle in a haystack. We describe two such examples and specific data exploration techniques we used.

A. Stuck Random Number Generator

On one occasion, our regression performance test running on a cluster of servers failed after running fine for several hours. The external symptoms were lower than expected throughput and a large number of failed requests. We started by scanning and grepping through the logs with scripts to find a point in time where the number of failed requests suddenly started to grow. We then continued searching for the root cause.

After a laborious process of comparing performance counters across different servers, we discovered that some performance counters started to significantly diverge starting roughly at the time when the requests began timing out. In particular, there was one server that received a much larger number of requests than the other servers. Looking at the logs of this server, we consequently discovered that a disproportionally large number of actors were placed on it compared to other servers. This imbalance kept growing as the time advanced.

In this specific test, the actors were randomly placed on servers and the expectation was the number of actors (and as a result also the number of requests) should be roughly equal across all servers. We now had the evidence that from a certain point on, disproportionally more and more actors were placed on one server only. That lead us to look closer into the placement logic. After a thorough code analysis we discovered that we were using the random number generator (RNG) in a thread-unsafe manner. C# RNG is not thread-safe and, if accessed simultaneously by multiple threads, can get “stuck” returning zero forever. This caused actors to be placed on one server (with index zero) from the moment the RNG got stuck. The fix was to protect access to RNG with a lock.

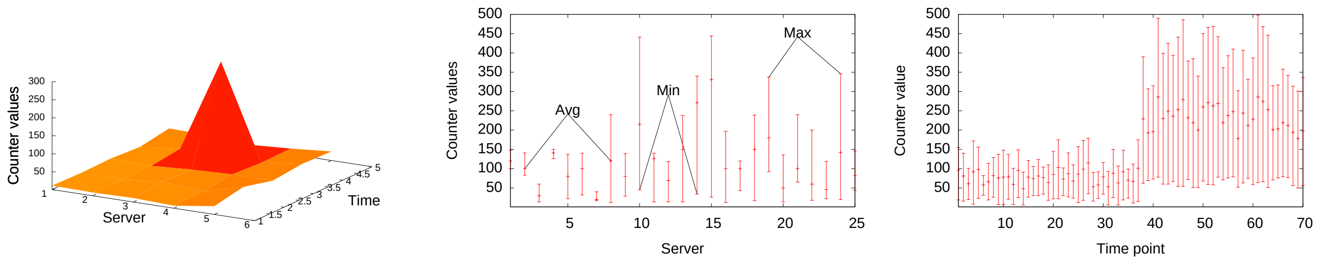
B. Leaking Buffer Pool

In this occasion, we observed decreasing throughput and growing requests latency. By using similar manual techniques like in the RNG case above, we were able to correlate the performance degradation with increasing memory pressure. At approximately the same time as the performance started to degrade, the amount of available memory in the system started to shrink and the overhead of garbage collection activities started to increase. This lead us to suspect a memory leak. Although Orleans is written in a managed language, it uses a custom buffer pool for messages aimed at minimizing the rate of memory allocations and reducing the pressure on the memory subsystem and the garbage collector. Consequently, we found a bug in our buffer pool implementation that caused code acquiring the buffer from the pool to occasionally not release it back to the pool (*i.e.*, leaking memory).

The above two bugs helped us define a number of requirements for *PAD*: (1) ability to visualize performance counters across time and easily find points in time when values start diverging from the norm; (2) automatically find counters that exhibit large variance across different servers (Comparative Analysis within a dataset); (3) automatically correlate one counter that we knew to be a symptom of a problem to other counters that could potentially lead to the root cause of the anomaly (Correlation Analysis).

IV. *PAD*-Assisted Investigation

In this section, we describe how *PAD* assists developers in finding performance problems and anomalies. Developers start the troubleshooting process when they suspect that a performance related problem has occurred. As detailed in



(a) Detailed view (b) Server view (c) Time view

Fig. 1: Three visualization graphs provided by PAD.

Section II, this can be either deviation from explicit performance requirements like SLAs, implicit internal implementation requirements (*e.g.*, CPU time) or deviation from normal performance learned from previous executions.

The developer troubleshooting the system is engaged in a *PAD-assisted investigation process*, which combines their expert knowledge, manual steps, and automatic anomaly detection techniques to find performance problems and its root cause(s). *PAD* helps developers in every step of this process, which typically involves the following five steps: (1) collecting the performance counters data; (2) visualizing the data; (3) threshold analysis; (4) correlation analysis; and (5) comparative analysis. Steps 1, 4, and 5 are completely automated by *PAD*, while steps 2 and 3 are manual steps assisted by *PAD*. We now describe each step in a typical troubleshooting workflow session in detail.

Step 1 - Performance Data Collection. The developer starts by directing *PAD* to gather relevant performance counter data. The developer only needs to provide the location of the log files, or the Azure storage account that holds the logs, and the *PAD* automatically downloads the data, parses it, and stores it in an in-memory compact data structure.

Step 2 - Data Visualization. After data is gathered, the developer typically wants to visualize it. Visualizing the data can sometimes reveal the problem quickly without requiring further complicated analysis. System developers typically suspect certain performance counters, which they prefer to analyze first. The selection of the performance counters is based on developer's knowledge about the system, and the performance diagnosis issue of interest. For example, if the developer suspects that the system is experiencing memory pressure, the developer can use *PAD* to visualize and summarize performance counters related to garbage collection or memory usage. *PAD* provides three different visualizations for different data views:

1) Detailed view. In the detailed view, *PAD* provides a 3D data plot. A 3D plot allows the developer to visualize and compare values both spatially (across all servers) and temporally (across time). Figure 1a illustrates an example plot produced by *PAD* for one performance counter. The X-axis represents server name, the Y-axis represents time, and the Z-axis represents the performance counter value. Any point in the plot captures the value of a performance counter at a particular time and in a particular server. The detailed view provides developers with overall trend information based on time and location. It can also prompt developers to perform further analysis when there are spikes (or anomalies) in the plot.

2) Server view. In the server view PAD visualizes summary statistics (*i.e.,* average, median, standard deviation, minimum, maximum, and quantiles) across time for a selected performance counter in each server. Figure 1b illustrates an example of a server view graph, which is called a *stock type chart* [16], because it shows the average, maximum and minimum values of the performance counter in each server. This view allows developers to quickly compare performance counter summary statistics across all servers.

3) Time view. In the time view, PAD visualizes summary statistics (*i.e.,* average, median, standard deviation, minimum, maximum, and quantiles) across all servers for a selected performance counter at each time point. Figure 1c illustrates an example of a time view graph. The time points are calculated with respect to the start time of system execution. This view allows developers to quickly compare performance counter summary statistics across all times regardless of the server.

By providing visualizations in three different views developers are able to gain more insight about system performance. For example, spike in the server view might be an indication of a hot server that performs more work than others. This allows developers to reduce the problem space into one particular server and concentrate further investigations at this server (like in the case in Section III-A)—eventually saving time. The visualization may not reveal any insights, or may trigger further analysis, including the need to look for other counters or compare certain counter values to predefined thresholds.

Step 3 - Threshold Analysis. In threshold analysis, developers define threshold values for a given counter. *PAD* compares counter values (or their statistical properties, *i.e.*, means, medians, and quantiles) against the predefined threshold values. Performance counters that violate their threshold are reported back. Developers define thresholds using an XML configuration file. Listing 1 illustrates an example for configuring thresholds for different performance counters.

Developers can configure thresholds that apply to the detailed, server, or time view of each performance counter. Developers can specify what statistical property (*e.g.*, mean, median) to apply the rule to, or that the threshold should be compared with all values of the distribution. Likewise, developers can specify an expected percentage with respect to a statistical property. For example, the developer can ask to find all the occurrences of a particular performance counter exceeding more than X% from the median. Finally, threshold analysis in PAD supports Z-score [17] comparisons for each value in the distribution. This helps developers detect outliers when the values are distributed according to a normal distribution.

Threshold analysis is usually used in combination with expert knowledge related to acceptable range of values for certain counters. For example, the developer may want to check if the time spent in garbage collection (GC) has exceed 15% of the CPU time at any point in time. Listing 1 illustrates an example configuration file with two rules: (1) find any time and server that the value of the *PercentOfTimeInGC* counter was greater than 15% ("any in the detailed view") and (2) find any time that the average value of the *NumQueuedMsgs* counter across all servers was greater than 5 ("average in the time view").

```
<ThresholdConfig>
 <PerformanceCounter Name="Runtime.GC.PercentOfTimeInGC">
   <Rule AppliesTo="DetailView">
    <Statistic>Any</Statistic>
    <ExpectedValue>15</ExpectedValue>
    <ComparisonOperator>GreaterThan</ComparisonOperator>
   </Rule>
 </PerformanceCounter>
 <PerformanceCounter Name="MessageQueue.NumQueuedMsgs">
   <Rule AppliesTo="TimeView">
    <Statistic>Average</Statistic>
    <ExpectedValue>5</ExpectedValue>
    <ComparisonOperator>GreaterThan</ComparisonOperator>
   </Rule>
 </PerformanceCounter>
</ThresholdConfig>
```

Listing 1: Example threshold analysis configuration file.

Step 4 - Correlation Analysis. Using the first three steps above, the developers may be able to find what counters behave abnormally. This, however, may not facilitate root cause analysis. For example, imagine the developer has established that a certain server spends more than 15% in GC. The question now is why? What has happened in the system to cause this undesired behavior? The developer may not have a direct answer to this question and may not know the exact counter to look for. In such a situation, the developer can use correlation analysis to find the counters responsible for the root of the problem.

In correlation analysis *PAD* detects a set of counters that can explain the distribution of another performance counter. *PAD* supports two correlation analysis techniques:

1) Pearson Coefficient. The Pearson coefficient is used to check whether any two performance counters have a linear correlation [17]. Pearson coefficient calculates a value in the range [-1, 1]. The closer this value to either endpoint, the greater the correlation between the two performance counters.

2) Spearman Coefficient. The Spearman coefficient is a measure of how well two counters can be described using a monotonic function [17]. Spearman coefficient also provides a value in the range [-1, 1]. When the value is close to either endpoint, the two performance counters can be explained as a monotonically increasing function of the other.

PAD finds all explanatory counters that have a correlation value greater than some X (usually 0.9 in our usage) with the performance counter of interest using Pearson and Spearman correlations. This enables system developers to narrow down reasons for abnormal values in certain performance counters.

An example of using correlation analysis is when the time in GC exceeded the threshold of 15%, the tool found that this spike in GC activity correlated to a spike in a number of queued requests in this server. The server in question was receiving more load than the other servers and failed to keep up. This provided developers with enough information to look into the reason for load imbalance, and helped the developers identify the root cause.

Step 5 - Comparative Analysis. Sometimes the developer may not know what counters to start with. In such situations, using visualization or threshold analysis might be too time consuming, provide too much data that is hard to analyze, and have a low chance of finding the root cause. In such cases *PAD* can help automatically detect anomalous counters based on statistical properties, such as average, median and quantiles, that deviate from other "normal" behavior of this counter.

PAD finds abnormal performance counters using *comparative analysis* [18]. Comparative analysis is a form of exploratory data analysis technique where statistical properties of different view points of a performance counter dataset are compared against each other. More specifically, PAD implements the following comparative analysis methods:

1) Comparative analysis within a dataset. In this analysis, *PAD* uses a given dataset to find performance counters that have abnormal statistical properties either in specific servers, or at different time points. *PAD* uses Equation 1 for comparative analysis within a dataset.

$$X = \frac{\left|GlobalMedian - LocalMedian\right|}{GlobalStandardDeviation} \quad (1)$$

In this equation, we use a global median as a reference point to compare with a local median. A local median is a statistical property of the distribution of the performance counter in a server, or at a time point. *PAD* uses medians instead of averages because they are more robust to noisy data [14] (high and low fluctuations of a performance counter will have little impact on the median). Both metrics will therefore remain stable enough to use as a reference for comparison.

By taking the different between the global and local medians, *PAD* calculates a local counter's deviation with respect to its global value. Finally, *PAD* normalizes the calculated deviation by the standard deviation of the global distribution to account for the fact that different performance counters can have different ranges of values. This provides PAD with

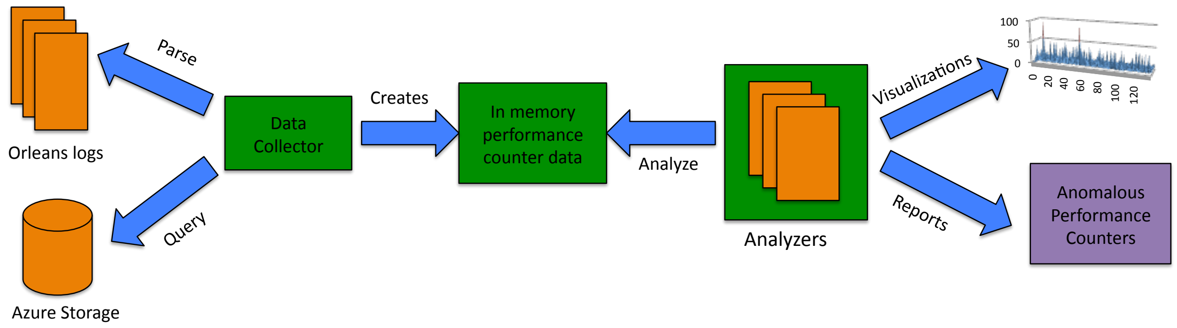


Fig. 2: Design of PAD.

a normalized method for comparing different counters that would be hard to compare using raw values.

2) Comparative analysis between datasets. PAD can also be used to compare different datasets, such as different regression test runs of the same application or different system releases. In this analysis, the developer specifies the reference ("correct") dataset, and *PAD* attempts to detect suspicious performance counters in the anomalous dataset. *PAD* uses Equation 2 for comparative analysis between datasets.

$$X = \frac{|RefDatasetMedian - DatasetMedian|}{RefDatasetStdDev} \quad (2)$$

As illustrated in Equation 2, *PAD* uses global medians and standard deviations of each performance counter in each dataset to calculate the deviation value X. *PAD* automatically performs the comparative analysis for all counters (the developer does not need to specify specific counters as in the threshold or comparative analysis). Once the suspicions counters are found, the developer can use *PAD* for visualization, threshold, and correlation analysis of the performance counters *PAD* has identified.

V. Implementation of PAD

Figure 2 shows the overall workflow of *PAD* and its internal modules. *PAD* can either collect data from Azure storage or log files. The data collection component is decoupled from the analysis components so that new data sources can be integrated without changing the analysis modules. After collecting the data, *PAD* builds an in memory model that is used by all analysis modules.

PAD (by default) implements the analysis techniques described in the previous section. Develoeprs can also extend it with their own analysis techniques through an extensible analysis framework. The different features of *PAD* (*e.g.*, how to collect data, what type of analysis to use, and how to use visualization) are easily configurable via XML. Last, the visualization component of PAD is based on automatically generating Excel charts for selected set of performance counters and uses C# COM interface of Microsoft Excel.

Time correlation. As specified in Section II-C, another challenge in analyzing the performance of Orleans is correlating the counters across different servers at different points in time. This is a common problem in distributed systems, where there is no global clock shared by all servers and where per-server clocks may not be fully synchronized [19].

Prior research efforts have proposed several techniques, such as vector clocks [20], to address this problem. These approaches, however, are intrusive as they requires system instrumentation, and send messages between servers to perform the correlation. *PAD* uses a different approach. *PAD*'s goal is to find the distribution of values of a particular performance counter at time point t_i from all servers. Since servers do not start at exactly the same time, we first find two time points $[t_s, t_f]$ where t_s is the latest starting time point of performance counters recording across all servers and t_f is the earliest finish time point of counters recording across all servers. We consider counter values in all servers during $[t_s, t_f]$ only.

Assuming the configured periodic logging interval of performance counters is d, *PAD* calculates the maximum number of time points N that can be contained inside the time interval $[t_s, t_f]$ using Equation 3.

$$N = \left\lfloor \frac{t_f - t_s}{d} \right\rfloor \quad (3)$$

Since clocks of different servers may not be synchronized, all servers may not have exactly N time points. *PAD* therefore takes the maximal N' time points that are common to each server such that $N' \leq N$. *PAD* then indexes each time point from 1 to N' starting from t_s in each server. Because we have taken the same number of points from each server, it allows *PAD* to correlate performance counter values at similar indices in each server. The distribution of performance counter values at each time point are the correlated values at each index.

VI. Applying PAD to Orleans

This section discusses two applications of *PAD* to analyze the performance counters in Orleans. In these particular scenarios, we used automated Orleans performance tests running on 25 servers machines and a set of client machines as load generators. Each client is configured to send 1 million requests.

A. Unbalanced DHT

In Orleans, actor instances are hosted on all servers. A distributed directory maps actor identities to their locations so incoming requests are brokered to their correct locations.

The actor registry is implemented as a *Distributed Hash Table (DHT)* [21]. Each server is responsible for hosting a portion of the DHT. It is important to keep the DHT balanced so each server handles roughly the same amount of requests related to resolving actors locations.

During one test, Orleans was experiencing lower than expected throughput. We first analyzed Orleans performance counters using *PAD* by performing a comparative analysis within a dataset (Equation 1 in Section IV) on a problematic performance test dataset. *PAD* found two anomalous performance counters in one particular server: *Directory.Registrations.Local* counter and *Directory.Registrations.Remote.Received* counter. This means that the number of local registrations of the DHT in this server was high compared to other servers *(Directory.Registrations.Local)* and that it also received more remote registration requests than other servers *(Directory.Registrations.Remote.Received)*. This was caused by the fact that this server was responsible for a much larger portion of the DHT. *PAD* therefore was able to correctly identify the anomalous performance counters related to this issue. More importantly, *PAD* helped us pinpoint the root cause of performance degradation.

B. Performance Bottleneck and Tuning Analysis

We also used *PAD* to assist us in evaluating various performance optimization techniques in Orleans. As part of this work, we analyzed the impact of the different optimization techniques on performance (*e.g.*, end-to-end throughput and latency) and specific performance counters. For example, we implemented a certain batching algorithm and inspected its impact on end-to-end throughput, number of messages, message size distribution, buffer pool, and number of socket system calls. *PAD* therefore allowed us to quickly assess the effectiveness of various optimization techniques on low-level system components and greatly shortened our trial cycle. Without *PAD*, detailed investigation of a large number of performance counters would be very difficult.

VII. Related Work

A. Approaches that Rely on Historical Performance Data and Known Performance Problems

There are several related works [8], [13], [22], [23] on analyzing performance of large-scale distributed systems similar to Orleans. Similar to *PAD*, these approaches rely on performance counters to detect performance anomalies. Foo *et al.* [22] calculate performance signatures from previous executions and use them as a baseline to compare against performance signatures of new executions. They assume that older executions do not suffer from performance anomalies. This approach is close to regression testing as it validates if anomalies are introduced into newer software versions. They, however, only do comparative analysis, which only provides a Yes/No answer on performance anomalies. In contrast, *PAD* facilitates different types of analysis beyond regression testing.

Nagaraj *et al.* [24] propose a method to compare two system logs, one with good and one with bad performance. After categorizing log messages as events and states, they calculate summary statistics for event timings, event counts, and state variable values used to compare the logs. Their approach is similar to the comparative analysis in *PAD*, but they do not provide other non-comparative techniques.

Bodik *et al.* [14] also propose a signature-based performance anomaly detection. Their method calculates signatures called *fingerprints* from historical performance data collected during a performance crisis. The goal is to quickly identify whether a similar performance crisis has occurred in the past so that known solutions can be applied. This approach is hard to apply when there are no previously known crises.

B. Approaches that Do Not Require Historical Data

Malik *et al.* [13] applied *Principal Component Analysis (PCA)* [25] to reduce the number of counters used to analyze performance anomalies. The main assumption is that counters that have high variance are the ones that represent the performance anomalies. This assumption, however, does not hold in all cases. For example, a system that experiences varying workloads may result in high variances in most performance counters without any actual performance problem.

Attariyan *et al.* [11] proposed a *performance summarization* approach for identifying root causes of performance anomalies based on human errors, such as misconfigurations. They used dynamic binary instrumentation [26] to monitor application as it executes instead of execution logs or performance counters. However, their techniques only focus on misconfigurations and do not help to find root causes for other performance problems, such as bugs in implementation or design, like *PAD*.

Finally, there are other approaches [27]–[29] that use annotated software models to detect performance anti-patterns [30]. These approaches, however, use software model simulations and not real production software. Moreover, these approaches do not rely on statistical analysis, but instead use rules and logical-predicate analysis to detect performance problems.

VIII. Lessons Learned and Conclusions

In this paper, we presented *PAD*, a tool to analyze performance counters in multi-server distributed systems. *PAD* combines user-driven navigation analysis with automatic correlation and comparative analysis techniques. Based on our experience in applying *PAD* to the Orleans framework, we discovered that *PAD* was able to reduce developers' time and efforts in detecting anomalous performance cases and improve developers' ability to perform deeper analysis of such behaviors. Below we detail the lessons learned based on our experience with *PAD*.

1) Visualization and summary statistic is a key part in performance anomaly detection. Visualization provides a quick overview of performance and triggers deeper analysis when needed. We believe that visualization should be the first step in human-based performance anomaly detection. Multiple view points (server or time) as well as summary statistics (*e.g.*, average, median, standard deviation, minimum, maximum, and quantiles) are very helpful in navigating the large amounts of data, and selecting a view for further analysis.

2) Reducing the data size. It is important to reduce the number of performance counters before visualizing data and performing root cause analysis. This saves developers time and effort by focusing their attention on data most relevant to the anomaly. For example, although Orleans has nearly 200 performance counters, we discovered certain performance issues can be summarized using only few counters.

We also tried to apply Principal Component Analysis to reduce the number of performance counters used in the analysis. This approach transformed the original performance counters into a different, smaller dataset with different dimensions. The new counters, however, bared no semantic meaning, could not be correlated back to the actual system, and did not help us with root causes analysis.

3) Fully automated root cause analysis for performance anomalies is hard. Existing research on automating root cause analysis is based on functional failures [9], [31]. As explained in Section VII, expert knowledge is required to analyze the root causes of performance anomalies. This knowledge differs from system to system, which makes it hard to generalize and automate. *PAD* addresses this challenge by combining automatic correlation and comparative analysis with manual user-driven navigation analysis. We still believe that commonalities between different automated root cause analysis processes must be identified and reused. Finally, techniques to formalize the required expert knowledge from different domains are required so developers can begin developing domain-specific automated techniques for root cause analysis.

References

[1] Y. Han, "On the Clouds: a New Way of Computing," *Information Technology and Libraries*, vol. 29, no. 2, pp. 87–92, 2013.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance Debugging for Distributed Systems of Black Boxes," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, 2003, pp. 74–89.

[3] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin, "Orleans: Cloud Computing for Everyone," in *2nd ACM Symposium on Cloud Computing*, 2011, p. 16.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[5] P. Barford, N. Duffield, A. Ron, and J. Sommers, "Network Performance Anomaly Detection and Localization," in *IEEE INFOCOM*, 2009, pp. 1377–1385.

[6] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. P. Pazel, J. Pershing, and B. Rochwerger, "Oceano-SLA based Management of a Computing Utility," in *International Symposium on Integrated Network Management*, 2001, pp. 855–868.

[7] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar, "Comprehensive QOS Monitoring of Web Services and Event-based SLA Violation Detection," in *4th ACM International Workshop on Middleware for Service Oriented Computing*, 2009, pp. 1–6.

[8] H. Malik, H. Hemmati, and A. E. Hassan, "Automatic Detection of Performance Deviations in the Load Testing of Large Scale Systems," in *IEEE International Conference on Software Engineering*, 2013, pp. 1012–1021.

[9] S. Hangal and M. S. Lam, "Tracking Down Software Bugs using Automatic Anomaly Detection," in *24th ACM International Conference on Software Engineering*, 2002, pp. 291–301.

[10] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues, "A Study of the Internal and External Effects of Concurrency Bugs," in *International Conference on Dependable Systems and Networks (DSN)*, 2010, pp. 221–230.

[11] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating Root-cause Diagnosis of Performance Anomalies in Production Software," in *Operating Systems Design and Implementation*, 2012, pp. 307–320.

[12] V. Cortellessa, A. Martens, R. Reussner, and C. Trubiani, "A Process to Effectively Identify guilty Performance Antipatterns," in *Fundamental Approaches to Software Engineering*. Springer, 2010, pp. 368–382.

[13] H. Malik, Z. M. Jiang, B. Adams, A. E. Hassan, P. Flora, and G. Hamann, "Automatic Comparison of Load Tests to Support the Performance Analysis of Large Enterprise Systems," in *Europ. Conference on Software Maintenance and Reengineering*, 2010, pp. 222–231.

[14] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the Datacenter: Automated Classification of Performance Crises," in *5th ACM European Conference on Computer Systems*, 2010, pp. 111–124.

[15] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, Modeling, and Generating Workload Spikes for Stateful services," in *ACM Symposium on Cloud Computing*, 2010, pp. 241–252.

[16] Microsoft, "Available Chart Types," http://office.microsoft.com/en-us/excel-help/available-chart-types-HA010342187.aspx.

[17] R. C. Sprinthall and S. T. Fisk, *Basic Statistical Analysis*. Prentice Hall Englewood Cliffs, NJ, 1990.

[18] D. C. Hoaglin, F. Mosteller, and J. W. Tukey, *Understanding Robust and Exploratory Data Analysis*. Wiley New York, 1983, vol. 3.

[19] H. Kopetz and W. Ochsenreiter, "Clock Synchronization in Distributed Real-Time Systems," *IEEE Transactions on Computers*, vol. 100, no. 8, pp. 933–940, 1987.

[20] M. Singhal and A. Kshemkalyani, "An efficient implementation of vector clocks," *Information Processing Letters*, vol. 43, no. 1, pp. 47–52, 1992.

[21] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," in *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, 2001, pp. 149–160.

[22] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora, "Mining Performance Regression Testing Repositories for Automated Performance Analysis," in *10th IEEE International Conference on Quality Software (QSIC)*, 2010, pp. 32–41.

[23] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automated Performance Analysis of Load Tests," in *IEEE International Conference on Software Maintenance (ICSM).*, 2009, pp. 125–134.

[24] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Symposium on Networked Systems Design and Implementation*, 2012, pp. 26–26.

[25] I. Jolliffe, *Principal Component Analysis*. Wiley Online Library, 2005.

[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *ACM Sigplan Notices*, vol. 40, no. 6, 2005, pp. 190–200.

[27] C. Trubiani and A. Koziolek, "Detection and Solution of Software Performance Antipatterns in Palladio Architectural Models," in *ICPE*, 2011, pp. 19–30.

[28] V. Cortellessa, A. Di Marco, and C. Trubiani, "Performance Antipatterns as Logical Predicates," in *15th IEEE International Conference on Engineering of Complex Computer Systems*, 2010, pp. 146–156.

[29] J. Xu, "Rule-based Automatic Software Performance Diagnosis and Improvement," *Performance Eval.*, vol. 67, no. 8, pp. 585–611, 2010.

[30] C. Smith and L. Williams, "New Software Performance Antipatterns: More Ways to Shoot Yourself in the Foot," in *CMG*, vol. 2, 2003, pp. 667–674.

[31] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.