# *CauseInfer*:Automatic and Distributed Performance Diagnosis with Hierarchical Causality Graph in Large Distributed Systems

Pengfei Chen, Yong Qi, Pengfei Zheng, Di Hou
School of Electronic and Information Engineer, Xi'an Jiaotong University
Email: fly.bird.sky@stu.xjtu.edu.cn, {qiy,houdi}@mail.xjtu.edu.cn, p.f.zheng@stu.xjtu.edu.cn

***Abstract*—Modern applications especially cloud-based or cloud-centric applications always have many components running in the large distributed environment with complex interactions. They are vulnerable to suffer from performance or availability problems due to the highly dynamic runtime environment such as resource hogs, configuration changes and software bugs. In order to make efficient software maintenance and provide some hints to software bugs, we build a system named *CauseInfer*, a low cost and black-box cause inference system without instrumenting the application source code. *CauseInfer* can automatically construct a two layered hierarchical causality graph and infer the causes of performance problems along the causal paths in the graph with a series of statistical methods. According to the experimental evaluation in the controlled environment, we find out *CauseInfer* can achieve an average 80% precision and 85 % recall in a list of top two causes to identify the root causes, higher than several state-of-the-art methods and a good scalability to scale up in the distributed systems.**

## I. Introduction

Modern applications especially cloud-based and cloud-centric applications always consist of many components running in the large distributed environment with complex interactions. They are vulnerable to suffer from performance or availability problems due to the highly dynamic factors such as resource hogs, configuration changes and software bugs. Manual performance diagnosis is daunting, time-consuming and frustrating in large distributed environment due to the huge cardinality of potential cause set. Moreover with the rapid development of cloud computing, more applications are deployed in the cloud to provide services through network. This exacerbates the difficulty of performance diagnosis because of the inherent elastic resource sharing and dynamic management mechanism in the cloud computing. Therefore performance diagnosis becomes a big challenge in these systems.

Numerous previous work has been done in the performance diagnosis area but they mainly put their emphasis on locating anomaly coarsely (e.g. at service level [1], [2] or VM level [3], [4]) instead of identifying the real reasons causing performance problems. However we argue that performance diagnosis should not only cover coarsely anomaly locating but also cause inference in order to make efficient software maintenance or provide some hints to software bugs. If the performance problems were not correctly diagnosed, wrong actions may be taken to maintain the system leading to resource waste and revenue loss.

The large cardinality of suspicious cause set hinders us to uncover the actual culprits precisely and completely. Therefore it's impractical to propose a silver bullet to resolve all the performance problems. In this paper we limit our diagnosis on a subset of performance problems. From previous studies [3]–[5], we find out performance problems are partly caused by the runtime environment changes (e.g. resource hogs [3], [4] and configuration changes [5], [6]). And after reviewing the bugs of several open source systems, we observe that large number of bugs can cause performance problems. Here we only take into account the bugs relevant to the abnormal consumption of physical resources (e.g. CPU) or logical resources (e.g. lock). The reasons of choosing these bugs are: these metrics can be readily collected at runtime without instrumenting the source code; large number of these bugs exist in the software (see Figure 1). Our objective is to attribute the root causes of performance problems to the performance metrics mentioned above. Although we will not directly identify the software bugs, we provide some hints to software bug. For instance, if the root cause is attributed to the violation of lock number it probably indicates a concurrent bug occurs in the system.

To achieve the proposed objective, we build an automatic, black-box and online performance diagnosis system named *CauseInfer*. The basic idea of *CauseInfer* is to establish a causality graph by capturing the cause-effect [7] relationships and then infer the root causes along the causal paths in the causality graph . To fulfill this task, *CauseInfer* automatically constructs a two layered hierarchical causality graph: a coarse-grained graph with the purpose of locating the causes at service level and a fine-grained graph with the purpose of finding the real culprits of performance problems.

Once an SLO (Service Level Objective) violation in the front end servers occurs, the inference procedure is triggered. We first locate the performance anomaly at specific service(s) (e.g. tomcat) by detecting the violations of SLO metric then find out the root cause(s) by detecting the violations of other performance metrics in a local node To further strengthen the robustness of the diagnosis we introduce a new change point detection method based on Bayesian theory which is better than conventional detection methods based on structure changes like CUSUM [4]. Via the experimental evaluation in two benchmarks: Olio and TPC-W, we find out *CauseInfer* can pinpoint the root causes in an average 80% precision and a 85% recall and readily scale up in a large distributed system. Finally, *CauseInfer* works in a completely distributed manner

978-14799-3360-0/14/$31.00 ©2014 IEEE

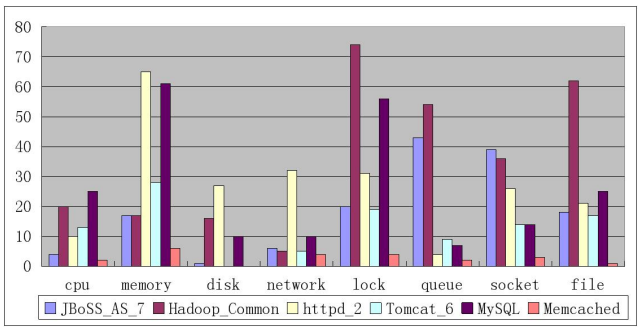


Fig. 1. The number of bugs relevant to logical and physical resources. The data is obtained through searching the relevant words like "cpu","memory" in the bug repository and then manually check whether the bug is indeed correlated with the searched words.

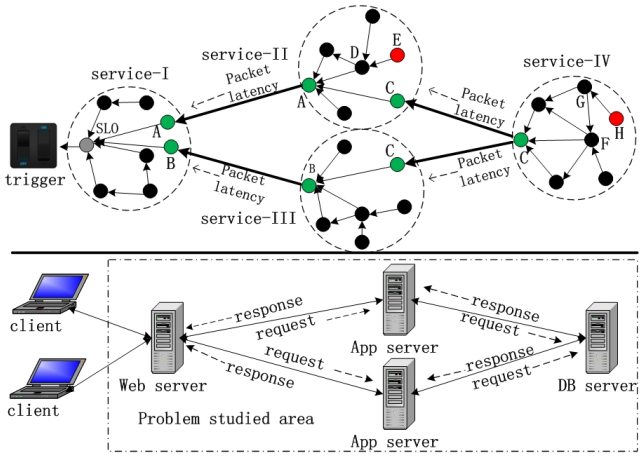


Fig. 2. The basic structure and the work flow of *CauseInfer*. The bottom is the physical topology of a three-tiered transaction processing system; the top is the abstracted service and metric causality graph. In the causality graph,the big dashed circle denotes service, the red node denotes the root cause, the black node denotes performance metric, the green node denotes SLO metric, the arc denotes the causality or dependency relationship and the arrow denotes the direction of anomaly propagation .

and largely reduces the data exchange between hosts during performance diagnosis.

The contributions of this paper are four-fold:

- We introduce a new BCP change point detection method which is more robust than CUSUM to find the change points in the long-term data series.
- We propose a novel light-weighted service dependency discovery method through analyzing the traffic delay between two services combining the new properties of modern operating system. It is very efficient to locate the performance anomaly at service level.
- We provide a new causality graph building method based on the original PC-algorithm. Using this causality graph, we can precisely pinpoint the root causes of performance problems at performance metric level.
- We design and implement *CauseInfer* to infer the root causes of performance problems . *CauseInfer* can hit the real culprits of performance problems in a high precision and recall with low cost.

The rest of this paper is organized as follows. Section II presents the overview of our system. Section III depicts the details of the system design. In Section IV, we will evaluate our system from several aspects in the controlled environment. And in Section V, we will compare our work with previous related work. Section VI concludes this paper.

## II. System Overview

In this section, we depict the umbrella of the *CauseInfer* system and show the work flow of this system via a simple case. The core modules of *CauseInfer* are a causality graph builder and an inference engine. The causality graph builder automatically constructs a two layered hierarchical causality graph. The inference engine is in charge of finding the culprits of performance problems with the causality graph. In the target distributed system, *CauseInfer* is deployed in each node and works in a distributed manner. Therefore in every node, there exists a causality graph. The inference is triggered by an SLO violation in the front end then iteratively goes to the back end services along the paths in the service dependency graph. If an SLO violation is detected in one node, the fine-grained inference is conducted according to the metric dependency graph. Figure 2 demonstrates the basic structure and the work flow of *CauseInfer* in a typical three-tiered system. In Figure 2, we assume metric E in service II node is the root cause. When an SLO violation of service I is detected, the cause inference is triggered. After a cause inference in service I node, we locate the performance anomaly at metric A indicating the SLO violation of service II. Therefore the cause inference in service II node is triggered and executed according to the metric causality graph stored in that node. Finally, we find the root cause, metric E. The whole inference path is: SLO $\rightarrow$ A $\rightarrow$ D $\rightarrow$ E. It should be noticed that the inference result may contain multiple metrics due to the statistical error. Hence a root cause ranking procedure is necessary to decrease the false positive and select the most probable root causes.

## III. System Design

In this section, we describe the details of the design and methodology of *CauseInfer*. As demonstrated in Figure 3, *CauseInfer* system mainly contains two procedures: an offline causality graph building procedure and an online cause inference procedure. The online procedure consists of two modules: data collection and cause inference. The data collection module collects the runtime performance metrics from multiple data sources. The cause inference module is in charge of traversing and ranking the root causes according to the causality graph generated by the causality graph builder. The offline procedure contains two modules:change point detection module and causality graph building module. The change point detection module converts every metric to a binary data series using a Bayesian change point detection. The causality graph building module uses the binarized metrics to construct a two layered hierarchical causality graph.

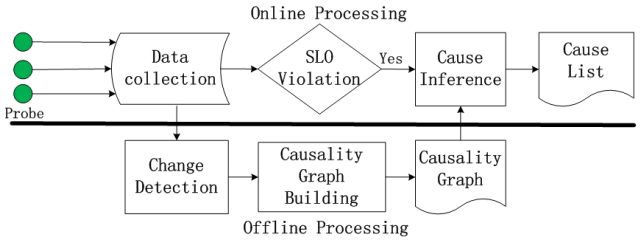


Fig. 3. The main modules of the *CauseInfer* system.

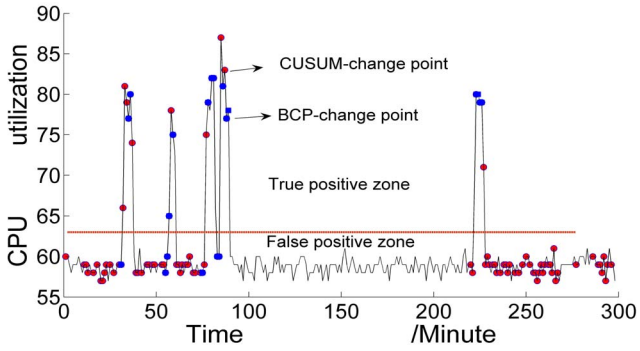


Fig. 4. The change point detection result using BCP and CUSUM.

### A. Data collection

The data collection module collects high dimensional run-time information from multiple data sources across different software stacks covering application [2], process and operating system. In the causality graph building phase, we need the SLO metric of one application. However not all the applications report SLO metric explicitly (e.g. Mysql,Hadoop) and it is variant in different applications, hence we propose a new unified SLO metric, tcp request latency (abbreviated as TCP_LATENCY). TCP_LATENCY is obtained by measuring the latency between the last inbound packet (i.e. request) and the first outbound packet (i.e. response) passing through a specific port. Although this metric is simple, it works well in our system. According to our observations, most of applications use TCP protocol as their fundamental transmission protocol like Mysql, Httpd,etc. Hence TCP_LATENCY can be adopted to represent the SLO metric of most applications.

### B. Change point detection

According to Pearl's cause-effect [7] notion, if two variables have causal relationship, the changes of one variable will cause the changes of the other. So before causality graph building, we first identify the changes in the time series. A conventional CUSUM [4] is always adopted to detect the change points. But due to the high sensibility to noise, CUSUM is hard to detect the long-term changes leading to a high false positive in an offline analysis. Therefore we introduce a more effective method based on Bayes theory, named Bayesian Change Point detection (abbreviated as BCP) [8].

The basic idea of BCP is to find an underlying sequence of parameters which partitions the time series into contiguous blocks with equal parameter values and locate the position of change point which is the beginning of each block. Given a sequence of observations: $X = (x_1, x_2, \cdots, x_n)$, the aim is to find a partition: $\rho = (P_1, P_2, P_3, \cdots, P_{n-1})$, where $P_i = 1$ indicates a change occurs at position $i + 1$, else $P_i = 0$. For the detailed theoretical analysis of BCP, we recommend the reader to refer to the paper [8]. Compared with the prevalent CUSUM method, BCP doesn't need to set the maximal number of change points and the group size in the raw time series. Owing to the sound Bayesian statistical inference, BCP is more effective than CUSUM to identify a change point. To compare the effectiveness of CUSUM and BCP, we use these two methods to detect the change points in the CPU_TOTAL metric obtained from a CPU fault injection experiment depicted in Section IV. There are 16 change points in the CPU_TOTAL metric confirmed by manually checking. From the detection results demonstrated in Figure 4, we can see there are so many false positives using CUSUM (group size=2) but very few using BCP (600 iterations). Therefore we conclude BCP is better than CUSUM to analyze a long sequence of time series. However BCP will not work in an online mode due to the requirement of long historical data, so in the online cause inference we still adopt CUSUM as our change point detection method.

### C. Causality graph building

In this section, we describe the details of a two layered hierarchical causality graph (i.e. service dependency graph and performance metric causality graph) building procedure. Before that we give a short introduction of *causality*. Different from the concepts of association or correlation, causality is used to represent the cause-effect relationship. The formal definition has been described in Pearl's work [7], here we just give a qualitative description. Given two variables X and Y, we say X is a cause of Y if the changes of X can affect the distribution of Y but not vice versa, denoted by $X \rightarrow Y$. Or in other words, X is a parent of Y, denoted by $X \in pa(Y)$. In the collective variables, if all the parents of Y have been fixed, the distribution of Y will be fixed and not affected by other variables. And in this causal relationship, it's not allowed two variables cause each other. So finally, all the causal relationships can be encoded by a DAG (Directed Acyclic Graph)

**Service Dependency Graph**

Our method has the similar assumption and methodology to Orion [1] that is the traffic delay between dependent services often exhibits "typical" spikes that reflect the underlying delay for using or providing these services. But the primary differences are: a. Our method focuses on a limited set of applications which use TCP as their underlying transmission protocol although it can be extended with extra effort. b. Our method relies on the new properties of the modern operating system such as network statistical tools and *kprobe* [9] used

[2]In this paper, we make no differences between application and service and use both terms interchangeably.

to probe the system call. c. We leverage traffic delay to determine the dependency directions rather than determine the dependency structure in the dependency graph which reduces the risk of wrong dependent relationships and computational complexity (from $O(n^2)$ to $O(n)$). According to our observations, TCP protocol takes a dominated position amongst all the protocols facilitated by common applications like Mysql, Tomcat,etc and almost all mainstream Linux operating systems have integrated with network statistic tools such as netstat, tcpdump and kprobe. Therefore our method can be used in most of distributed systems running on Linux operating system.

Different from Orion, we use a two-tuple *(ip, service_name)* instead of three-tuple *(ip, port, proto)* to denote a service considering that a service may utilize multiple ports. For example, in a three tiered system, a web server may access the application server through a random port. So if we adopt *port* as an attribute of a unique service, the dependency graph becomes dramatically huge even though the requests are all issued by the same service. In a distributed system, *ip* denotes a unique host and *service_name* denotes a unique service running in the host. We follow the definition of service dependency in Orion system that is: if service $A$ requires service $B$ to satisfy certain requests from its clients, $A \rightarrow B$. For instance, a web service needs to fetch the content from a database kept by a database service , so we say the web service depends on the database service. And in this paper, we are also only concerned about client-server applications which are dominant in modern applications.

The first step of our method is to use the connection information to construct a skeleton of the service dependency graph. Executing an off-the-shelf tool, *netstat*, in a host, we get a list of all the connection information including protocol, source, destination and connection state. We extract the source and destination information which connected by TCP protocol. Each of the connection is organized in the format $source\_ip : port \rightarrow destination\_ip : port$, we call it a *channel*. The channel is very close to the service dependency pair except one point: it doesn't contain a service name but a port. The following trivial work is to map a service port to a service name. To get the service name with respect to a local port is easy by querying the port information. But to a remote port, *CauseInfer* in the local host need to send a query to the *CauseInfer* in the remote host. After the mapping procedure, the skeleton of a service dependency graph in a local host is established. But one problem stays unresolved. The transmission between client and server is bidirectional which means we may get an opposite service dependency when observing in different hosts. For instance, when observing in host 192.168.1.117, we get the service connection (192.168.1.117, httpd) $\rightarrow$ (192.168.1.115, tomcat); but in host 192.168.1.115, we get (192.168.1.115, tomcat) $\rightarrow$ (192.168.1.117, httpd). To address this issue, we use and improve the traffic delay assumption mentioned above.

In a client-server structure, a common observation is the packets sent by the server change with the ones sent by the client. Therefore we use the lag correlation of the send traffic between two services to distinguish the dependency direction. To get the send traffic of a specific service, we count the number of packets transmitted by a specific process through probing the function *netdev.transmit* triggered when the network device wants to transmit a buffer. Assuming $X$ is the send traffic of service $A$, $Y$ is the send traffic of service $B$, the lag correlation between $X$ and $Y$ is defined as

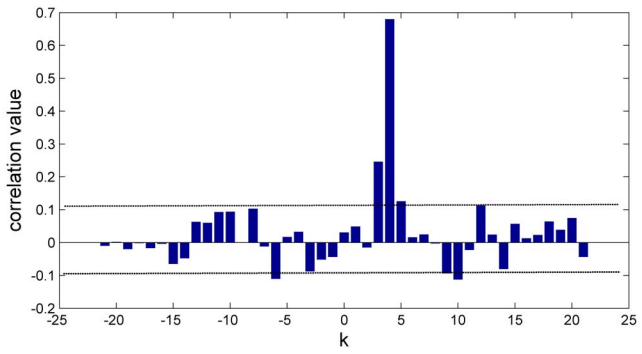


Fig. 5. The lag correlation of send traffic between Httpd and Memcached.

$$\rho_{XY}(k) = \frac{\sum_{t=0}^{N-1}(Y_t - \bar{Y})(X_{t-k} - \bar{X})}{\sqrt{\sum_{t=0}^{N-1}(X_t - \bar{X})^2 \sum_{t=0}^{N-1}(Y_t - \bar{Y})^2}} k \in Z \quad (1)$$

Where $k$ is the lag value, it can be positive and negative. In our system we set the absolute value of $k$ at 30 which can capture almost all the traffic delay. Our objective is to find a best $k$ which maximize $\rho_{XY}(k)$ namely

$$k^* = \{argmax(|\rho_{XY}(k)|), k \in [-30, 30]\} \quad (2)$$

According to the sign of $k^*$, the dependency direction is determined. If $k^* > 0$, $A \rightarrow B$; else $B \rightarrow A$. Figure 5 demonstrates the lag correlation between httpd and memcached applications. The result shows that $k^* = 4$ implies $httpd \rightarrow memcached$ confirmed in reality.

**Metric Causality Graph**

Our method is established on PC-algorithm [10]. The basic idea of PC-algorithm is to build a DAG in the collective variables based on the causal Markov condition [10], [11] and D-separation [10], [11]. To keep self contained, we first give some definitions and preliminaries. Given a graph $G = (V, E)$, $V$ is a set of variables and $E$ is a set of edges. If $G$ is directed, for each $(i, j) \in V$, $(j, i) \notin V$, denoted by $i \rightarrow j$. A DAG is a directed graph without any circle. PC-algorithm makes two assumptions: causal Markov condition and faithfulness [11]. As a fundamental property distinguishing causality from correlation, causal Markov condition is used to produce a set of independence relationships and construct the skeleton of a causality graph. It could be defined as:

**Definition 1.** *Given a DAG $G = (V, E)$, for every $v \in V$, $v$ is independent of the non-descendant of $v$ given its direct $pa(v)$*

In this paper we use a conditional cross entropy based metric $G^2$ [11] to test the independence of $X$ and $Y$ given $Z$, where $X$, $Y$ and $Z$ are disjoint set of variables in $V$. $G^2$ is defined

as

$$G^2 = 2mCE(X, Y|Z) \quad (3)$$

Where $m$ is the sample size, $CE(X, Y|Z)$ is the conditional cross entropy of X and Y given Z. If $G^2$ exceeds a preset significance level, say 0.05, the independence assumption will be accepted, so $X$ is independent of $Y$ given $Z$. The PC-algorithm begins with a completely connected undirected graph, then facilitates $G^2$ to capture all the conditional independences in the set of variables. After that a skeleton of DAG is constructed. The following work is to determine the causal directions using D-separation. The faithfulness assumption guarantees that the independence relationships among the variables in $V$ are exactly those represented by $G$ by means of the D-separation criterion [10]. Due to the limited space of this paper, we don't give the details of PC-algorithm. Please refer to the paper [10] for the details. Based on PC-algorithm, we set up two methods: a *conservative* one and an *aggressive* one. By the word of *aggressive*, we mean it doesn't use any prior knowledge to build the causality graph. while by the the word *conservative*, we mean the graph is initialized with some prior knowledge before construction. The prior knowledge includes which variables have no parents and which variables have no descendants in the graph. In this paper, we set TCP_LATENCY metric of the local service has no descendants because TCP_LATENCY can't cause changes of other metrics; a subset of possible root cause metrics including workload, configuration, TCP_LATENCY of the dependent services have no parents because they can not be changed by other metrics. Both of the algorithms are executed locally in the distributed systems.

The *aggressive* algorithm begins with a metric preparation. For a service without any dependent service such as database service, the causality graph is built using only the local performance metrics mentioned in data collection section. But for the one with some dependent services such as web service, the causality graph is built using not only the local performance metrics but also the TCP_LATENCY metrics of its dependent services. The length of training data is set a default value 200 which will be explained in Section IV . Then we conduct PC-algorithm to construct the causality graph and a DAG is obtained. However this DAG may contain multiple isolated subgraphs, counterintuitive causal relations and bidirectional links due to the lacking of evidence, statistical errors or non-causal relations at all. For example, $M5$ is isolated, the causal relation $M4 \rightarrow M2$ is counterintuitive and the causal relation between $M1$ and $M4$ is bidirectional in Figure 6 (a). So we further select a maximum subgraph from the DAG using the following conditions: a) TCP_LATENCY metric as the final effect metric has no descendants; b) The final effect metric is reachable from every path in the graph. c) There are no parents for the preset root cause metrics; d) For a bidirectional link, we select one direction randomly.

The procedure of *conservative* algorithm is the same as the *aggressive* one except the initialization. We initialize some directions in the graph before executing PC-algorithm. The links between TCP_LATENCY and other metrics are directed and the links between the preset root cause metrics and other metrics are directed. Compared with the *aggressive* algorithm, the *conservative* one can capture more causal relations and reduce the counterintuitive causal relations. For example, the causal relation $M1 \rightarrow SLO$, lost by the *aggressive* algorithm is captured by the *conservative* algorithm and simultaneously the counterintuitive causal relation $M4 \rightarrow M2$ is eliminated in Figure 8(b).

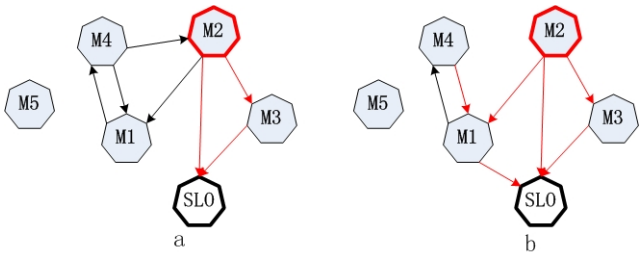


Fig. 6. An example of causality graph constructed by the *aggressive* algorithm (a) and the *conservative* algorithm (b). The red circle denotes a preset root cause and the subgraph with red links is the selected final causality graph.

### D. Cause inference

When an SLO violation in the front end occurs, the cause inference is triggered. We first infer the root causes of local service performance problems using the metric causality graph constructed by the graph building module. If the root causes are located on the SLO metrics of its dependent services, the inference is propagated to the remote dependent services. The process is conducted iteratively until no SLO violations or no service dependencies.

To detect the violations of SLO and other performance metrics online, we adopt two sided CUSUM with a sliding window. Assuming $X$ is a metric, we first initialize the sliding window using a normal data series with a fixed length, say 60 in this paper. Then if a new data $X_t$ arrives, we use CUSUM to check the data series $[X_{t-60}, X_t]$ whether $X_t$ is abnormal. If $X_t$ exceeds the lower control level (LCL) or upper control level (UCL), $X_t$ is abnormal and the sliding window will not move otherwise the new data is filled in the sliding window and the sliding window moves forward. A big advantage of this method is that it can adjust the anomaly threshold adaptively.

To infer the root causes in a specific node, we use a DFS (Depth First Search) method to traverse the metric causality graph. When a node in the graph is traversed, we use the CUSUM method to determine whether it is abnormal. If it is abnormal, we continue to traverse the descendants of this node otherwise we traverse its neighbor node. When there are no descendants for an abnormal node or no violations in all of its descendants we output this node as a root cause. Take Figure 6 (b) for example. We start the inference from node $SLO$ if $SLO$ is abnormal. Then we detect node $M1$, if it is normal, its neighbor node $M2$ is traversed. If $M2$ is abnormal, we output it as a root cause. The following node is $M3$, if it is abnormal, we continue to detect $M2$. Because $M2$ is abnormal, we output $M2$. Finally we find only one root cause $M2$ although $M2$ and $M3$ are both abnormal. However due to the multiple causal

paths, it's possible to get a set of potential root causes in some circumstances. Therefore it is necessary to rank the root causes and select the most probable one. In this paper, a simple z-score based method is employed to measure the violation of a performance metric. The measurement is defined as:

$$violation(X) = \frac{X(t) - \overline{X_{t-60,t-1}}}{\sigma_{t-60,t-1} + \varepsilon}, \varepsilon = 0.001 \qquad (4)$$

Where $\overline{X_{t-60,t-1}}$ and $\sigma_{t-60,t-1}$ are the mean and standard deviation of the sliding window respectively. In case of the situations where the metric may not change in the sliding window, we add a small number $\varepsilon$ to $\sigma_{t-60,t-1}$. According to this score, we get a list of ordered root causes.

## IV. Experimental Evaluation

We have implemented a prototype named *CauseInfer* and deployed it in the controlled environment. To collect the process and operating system performance metrics, we use some off-the-shelf tools such as Hyperic [12]; to collect other metrics, we develop several tools from the scratch. The sample interval in all the data collection tools is 1 minute. In the following, we will give the details of our experimental methodology and evaluation results in two benchmarks: TPC-W and Olio.

### A. Evaluation Methodology

Due to the lack of real operating platforms, *CauseInfer* is only evaluated in a controlled distributed system. But we believe it works well in a real system without exceptions. The controlled system contains five physical server machines hosting the benchmarks and four client machines generating the workload. Each physical server machine has a 8-core Xeon 2.1 GHZ CPU and 16GB memory and is virtualized into five VM instances including domain 0 by KVM. Each VM has two vcpu and 2GB memory and runs a 64-bit CentOS 6.2.

TPC-W is a transaction processing benchmark which is used to emulate online book shopping. In our controlled environment, we employ Apache Httpd, Apache Tomcat and Mysql as the web, servlet application and database service respectively and these services run in dedicated VM instances. We adopt Siege [13] to generate the HTTP requests randomly. To mimic the real performance problems, we inject several faults in the benchmark. For the performance problems caused by runtime environment changes, we inject the following faults: 1) CpuHog: a CPU-bound application co-locates with web server competing for CPU resource; 2) MemLeak: a memory-bound application continually consumes memory of the application server; 3) DiskHog: we use a disk-bound program to generate a mass of disk reads and writes on the web server; 4) NetworkJam: we use "tc", a traffic control tool in Linux, to mimic the packet loss on database server; 5) Overload: we increase the the number of request until the TCP_LATENCY of web service becomes anomaly; 6) ConfChanges: we change the configuration item such as the "KeelAliveTimeout" in Httpd configuration file and then restart Httpd service. For the performance problems caused by software bug, we inject the following faults: 1) CpuBug: we write a php file which largely consumes CPU resource and then access this file through HTTP protocol; 2) MemBug: a code snippet is injected in the "TPCW_home_interaction" class. $4KB$ memory will be leaked once this class is called.

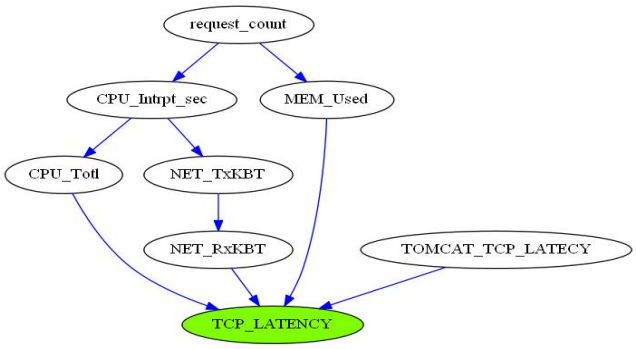


Fig. 7. Patrt of causality graph for Httpd service in TPC-W

Olio is a web 2.0 toolkit to help evaluate the suitability, functionality and performance of web technologies. We employ the same fault injection methodology to mimic the performance problems caused by environment changes as TPC-W. But for software bugs, we inject the following faults: 1) MemBug: we comment the "do_slabs_free" function call in the source code of memcached and recompile the code; 2) LockBug: a daemon program locks "PERSON" table in olio database periodically.

Each of the faults mentioned above will be repeated for more than 20 times and last 5 minutes. And multiple faults may be simultaneously injected in multiple nodes. To get the ground truth, we will log the fault injection time and types. We leverage two commonly used metrics: precision and recall [3], [4] and the rank assigned to the real root cause proposed in the paper [5] to evaluate the effectiveness of our system. By the rank metric, we mean the position of the real cause in the ranking list.

### B. Effectiveness Evaluation

Our system strongly relies on the causality graph, so we first present the causality graph using *conservative* algorithm. Figure 7 demonstrates part of the causality graph built for Httpd service in TPC-W benchmark. From the figure, we can see all the relations are reasonable except one: $NET_TxKBT \rightarrow NET_RxKBT$. Intuitively as a server, the send traffic (NET_TxKBT) changes with the receive traffic (NET_RxKBT). But we get an opposite result here. The most possible reason is that our system runs in a close loop which means the workload generator issues a new request only when it gets response from the Httpd server. This is a "back pressure" which is stated in [3]. And it is a reason to bias our result. But in a real open system, this phenomenon is scarce.

Figure 8 and Figure 9 demonstrate the diagnosis precision and recall of our *conservative* algorithm setting $rank = 1$ which means the real cause is the top one of the ordered list in TPC-W and Olio benchmarks, with only one fault injected every time. From Figure 8, we observe that most of

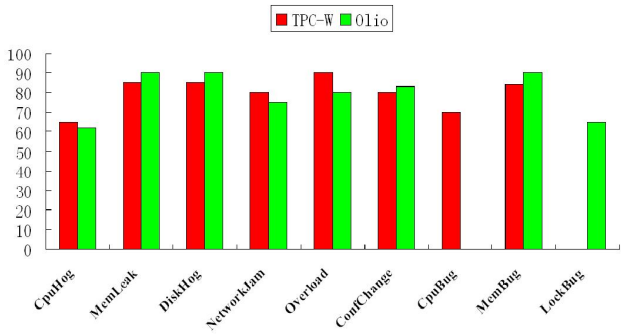


Fig. 8. The diagnosis precision using "conservative" when $rank = 1$

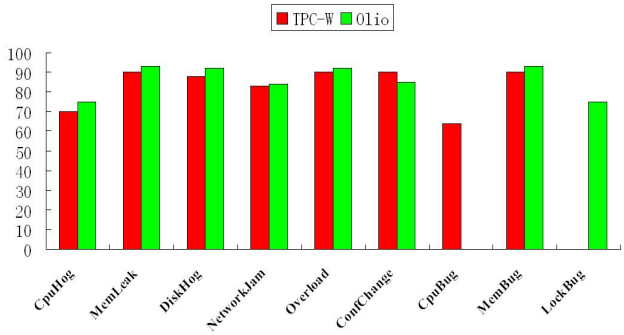


Fig. 9. The diagnosis recall using "conservative" when $rank = 1$

the precisions fall in the range 80%-90% except three cases: CpuHog, CpuBug and LockBug. Through a manual inspections of the whole work flow of *CauseInfer*, we find the anomaly detection procedure with CUSUM has a high false positive. The reason is that the SLO metric of the benchmark is sensitive to these faults, they can cause the SLO metric fluctuates frequently. While for other faults like DiskHog, they are not easy to cause SLO violations. From Figure 9, we observe the recall falls between 70% and 93% and shares a similar characteristic with the precision. Due to the similar structures of TPC-W and Olio, we observe very similar diagnosis results on these two benchmarks. Therefore in the following, we only show the diagnosis results in the TPC-W benchmark.

### C. Comparison

We conduct several comparisons with previous studies. 1) TAN: TAN (Tree Augment Bayesian Network) is adopted to diagnose performance problems in paper [14]. For comparison, we substitute PC-algorithm with TAN to construct the dependency graph in a local node; 2) NetMedic [5]: although the original approach is designed to build the component dependency graph, it can also be used to build the metric dependency graph. We compare it with our system at both of component level and metric level diagnosis; 3) PAL [4]: it is a propagation-aware performance diagnosis method. For comparison, we implement it to diagnose the injected faults; 4) FChain [3]: it shares the same idea with PAL even though it

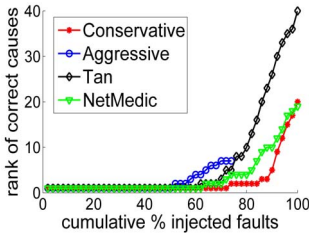


Fig. 10. The comparison using rank

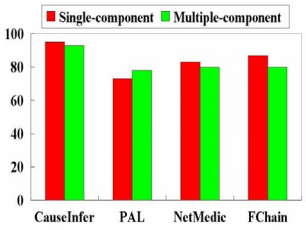


Fig. 11. Single VS Multiple

leverages a new anomaly detection method. To reduce the bias caused by the implementation deviation of these methods, we guarantee the injected faults can make significant violations of SLO metrics.

For metric level diagnosis, Figure 10 demonstrates the result of *CauseInfer*, TAN and NetMedic conducted in TPC-W benchmark with only one fault injected every time. From this figure, we observe that the *conservative* algorithm always, about 80%, puts the root cause in the top two causes showing a high diagnosis precision. But using the *aggresive* algorithm, we only get a 50% precision when $rank = 2$ due to the causal relation loss. Actually, we have only nine metrics connected in the final causality graph which means the algorithm can attribute root causes to at most eight metrics. That's the reason why the curve is truncated when the percentage of cumulative injected faults exceeds 75%. Compared with the other two methods, our system achieves better diagnosis result when the rank is set. In the graph constructed by TAN, we observe that each node only has one parent if not counting the SLO node. Therefore many causal relations are lost. And what makes it worse is that every node in the graph connects with SLO node leading to many fault positives. In the graph constructed by NetMedic, we observe that there are many redundant links and circles which makes the inference unstable and inaccurate as NetMedic constructs causality graph based on correlation rather than Markov conditional independence. We investigate what's the worst case and observe that TAN and NetMedic achieve very low precision to diagnose overload faults, about 15%. Because these faults can cause violations of many other metrics simultaneously. TAN and NetMedic put the the most violated metrics in the first position but not workload while *CauseInfer* can always put the workload violation on the top.

Figure 12 and Figure 13 show diagnosis results of *CauseInfer*, TAN, NetMedic with multiple simultaneous injected faults ranging from 2 to 6. Due to the multiple faults, we use precision and recall instead of the rank metric to evaluate them. From these two figures wen can see, there is some degradation in the effectiveness compared with single fault injection for *CauseInfer*, but it is still better than other methods. We observe that although TAN method has a lower precision, it has a high recall due to the full connections from SLO metric to other metrics in the dependency graph.

For node level diagnosis, Figure 11 demonstrates the diagnosis precision of *CauseInfer* ($rank = 1$), PAL, NetMedic and FChain under fault injection in single component and

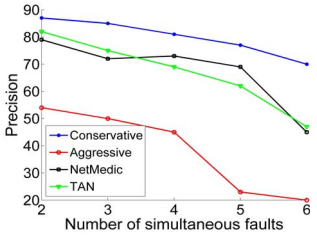


Fig. 12. Precision of diagnosis

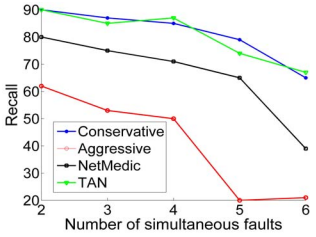


Fig. 13. Recall of diagnosis

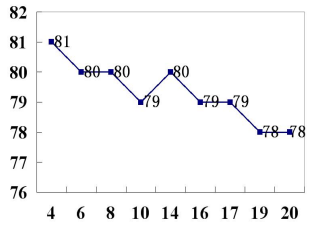


Fig. 14. Precision vs service number

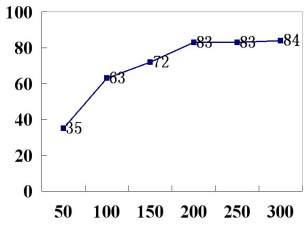


Fig. 15. Precision vs data length

TABLE I
THE OVERHEAD OF OUR SYSTEM

| System Module | CPU cost |
|---|---|
| Data collection | 10% CPU utilization |
| Change point detection (one metric) | 0.003 second |
| Service dependency graph (one service pair) | 0.05second |
| Metric causality graph (100 metrics,200 samples) | 10 second |
| Cause inference (one node,one fault,100 metrics) | 5 second |

multiple components. The result shows that our system has an extraordinary precision compared with other systems. The underlying reason is that we leverage the application specific information to construct the service dependency graph rather than inferring the dependency relations by analyzing the history correlation or sequences of change points of system metrics.

But compared with PAL and FChain, *CauseInfer* brings a high overhead to the host. Table I shows the overhead of our system. Data collection module takes about 10% CPU utilization as we collect over one hundred metrics from multiple data sources. The metric causality graph building module and cause inference module are highly computation-intensive. But overall, it is still a light-weighted tool. In the future work, we will select fewer but effective metrics and improve the cause inference algorithm to reduce the cost.

### D. Discussion

**scalability**:*CauseInfer* works in a distributed manner. Most of the computation is done locally and the data exchange between hosts is very small only including the SLO information and the send traffic of a specific service. Therefore our system is easy to scale up no matter adding new services or machines in a large distributed system. To test the scalability of our system, we deploy more services such as Memcached and Tomcat. From Figure 14 we can see our system has only three percent degradation in precision from 4 services to 20 services showing a good scalability.

**Sensibility**: We have conducted several experiments to evaluate the sensibility to the data length in the metric causality graph building procedure. Figure 15 demonstrates the precision changes with the data length increasing. When the data length is small, a large number of circles and isolated nodes exist in the graph constructed by the original PC-algorithm due to the lack of evidence. So many faults are generated using the causality graph obtained by our method. But when the data length reaches 200, the causality graph becomes stable and the precision doesn't change significantly. That's the reason why we choose 200 in this paper.

## V. RELATED WORK

Performance diagnosis in the large distributed system with many components is a daunting and frustrating work. To pinpoint the causes of performance problems, a large body of researchers have devoted themselves to this area. We present the relative work from the following aspects.

**Log-based method**: These methods use the log information reported by the system to investigate the causes of performance problems [15]–[17]. A general idea is to train the invariant patterns from the history data and then use these patterns to detect anomalies. Some of these work also use graph model [15], [17] to express the patterns, but they are event causal graph instead of performance metric causal graph. Actually our system is also able to analyze the log data sets if the events are transformed into a 0-1 binary data series. Although they can discovery more informational causes, they are hard to work online.

**Trace-based method**: Many famous tools fall into this class such as Magpie [18], X-Trace [19], and Pinpoint [20]. These tools can precisely record the execution path information and locate the abnormal code through instrumenting the source code. It's very helpful to debug the distributed applications. But the overhead brought by these tools are significant which hinders them to be widely used in modern applications. And deploying these tools is also a tough job requiring the administrators to understand the code well. Compared to them, *CauseInfer* can be easily deployed and used without instrumenting the source code. Although it can't detect the real software bugs directly, it indeed provides some hints to them.

**Signature-based method**: These methods employ supervised learning algorithms such as Bayesian classifier or K-NN to classify performance anomalies under several typical scenarios. CloudPD [21] uses a layered online learning approach including K-NN, HMM and k-Means to deal with frequent reconfiguration and high rate of faults; Fingerprint [22] can automatically classify and identify the performance crises using a simple statistical method in large data centers; Cherkasova et al. [23] proposes a new performance signature method based on the resource utilization of requests; Fa [24] uses a new technique named "anomaly based clustering" to make failure

signatures robust to the noisy monitoring data in production systems, and to generate reliable confidence estimates. Most of these methods require labeled data sets or problem tickets and show weakness at discovering new anomalies. While *CauseInfer* is able to capture new anomalies readily due to its unsupervised nature.

**Dependency graph-based method**: Recently dependency graph based performance diagnosis becomes a surge. Many state of the art systems are proposed like WISE [6], Constellation [2], Orion [1], NetMedic [5], FChain [3] and PAL [4]. The emphasis of FChain [3], PAL [4] and NetMedic [5] is put on the performance diagnosis at component or service level. They are not effective enough to infer the root causes at performance metric level which is stated in Section IV. The aim of Constellation [2] and Orion [1] is to locate the anomaly at service level instead of metric level. WISE [6] adopts a similar causal graph to ours. But it is used to predict the effect of possible configuration and deployment changes rather than infer the root causes. TAN [14] is adopted to infer the performance problems at metric level which is the first close to ours, but it is also not effective enough due to the lack of causality.

## VI. Conclusion

Towards automatic performance diagnosis in large distributed systems, this paper designs and implements the *CauseInfer* system. *CauseInfer* can not only pinpoint the root causes caused by runtime environment changes but also provide some hints to software bugs. To fulfill the diagnosis procedure, *CauseInfer* automatically builds a two layered hierarchical causality graph and infers the root causes along the paths in the graph. The service dependency graph is built using a novel light-weighted traffic delay method combining the new properties of modern operating systems; the metric causality graph is built on Pearl's cause-effect notion. To strengthen the effectiveness of our system, we introduce a Bayesian change point detection method which is much better than prevalent CUSUM method. The experimental evaluation in TPC-W and Olio benchmarks shows that *CauseInfer* can achieve a high precision and recall for performance diagnosis and scale up readily in large distributed systems.

## Acknowledgment

The work is sponsored by National Natural Science Foundation of China under Grant No.(60933003 ,61272460),the National High Technology Research and Development Program of China(863 Program) under grant No.2012AA010, and Ph.D. Programs Foundation of Ministry of Education of China under Grant No.20120201110010.

## References

[1] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl, "Automating network application dependency discovery: Experiences, limitations, and new solutions." in *OSDI*, vol. 8, 2008, pp. 117–130.

[2] P. Barham, R. Black, M. Goldszmidt, R. Isaacs, J. MacCormick, R. Mortier, and A. Simma, "Constellation: automated discovery of service and host dependencies in networked systems," *TechReport, MSR-TR-2008-67*, 2008.

[3] H. Nguyen, Z. Shen, Y. Tan, and X. Gu, "Fchain: Toward black-box online fault localization for cloud systems," in *Distributed Computing Systems (ICDCS), 2013 IEEE 33nd International Conference on*. IEEE, 2013.

[4] H. Nguyen, Y. Tan, and X. Gu, "Pal: P ropagation-aware a nomaly l ocalization for cloud hosted distributed applications," in *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*. ACM, 2011, p. 1.

[5] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl, "Detailed diagnosis in enterprise networks," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 243–254, 2009.

[6] M. Tariq, A. Zeitoun, V. Valancius, N. Feamster, and M. Ammar, "Answering what-if deployment and configuration questions with wise," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 99–110.

[7] J. Pearl, *Causality: models, reasoning and inference*. Cambridge Univ Press, 2000, vol. 29.

[8] D. Barry and J. A. Hartigan, "A bayesian analysis for change point problems," *Journal of the American Statistical Association*, vol. 88, no. 421, pp. 309–319, 1993.

[9] R. Krishnakumar, "Kernel korner: kprobes-a kernel debugger," *Linux Journal*, vol. 2005, no. 133, p. 11, 2005.

[10] M. Kalisch and P. Bühlmann, "Estimating high-dimensional directed acyclic graphs with the pc-algorithm," *The Journal of Machine Learning Research*, vol. 8, pp. 613–636, 2007.

[11] P. Spirtes, C. Glymour, and R. Scheines, *Causation, prediction, and search*. The MIT Press, 2000, vol. 81.

[12] "Hyperic." [Online]. Available: http://www.hyperic.com/

[13] "Siege." [Online]. Available: http://www.joedog.org/siege-home/

[14] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons, "Correlating instrumentation data to system states: A building block for automated diagnosis and control." in *OSDI*, vol. 4, 2004, pp. 16–16.

[15] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 2009, pp. 149–158.

[16] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 117–132.

[17] J.-G. Lou, Q. Fu, S. Yang, J. Li, and B. Wu, "Mining program workflow from interleaved traces," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2010, pp. 613–622.

[18] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling." in *OSDI*, vol. 4, 2004, pp. 18–18.

[19] 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 & implementation*. USENIX Association, 2007, pp. 20–20.

[20] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 595–604.

[21] B. Sharma, P. Jayachandran, A. Verma, and C. R. Das, "Cloudpd: Problem determination and diagnosis in shared dynamic clouds," in *IEEE DSN*, 2013.

[22] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: automated classification of performance crises," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 111–124.

[23] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, "Automated anomaly detection and performance modeling of enterprise applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 27, no. 3, p. 6, 2009.

[24] S. Duan, S. Babu, and K. Munagala, "Fa: A system for automating failure diagnosis," in *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. IEEE, 2009, pp. 1012–1023.