Microscope: Pinpoint Performance Issues with Causal Graphs in Micro-service Environments

Jinjin Lin, Pengfei Chen(✉), and Zibin Zheng

School of Data and Computer Science, Sun Yat-sen University, Guangzhou, China
linjj23@mail2.sysu.edu.cn
{chenpf7,zhzibin}@mail.sysu.edu.cn

Abstract. Driven by the emerging business models (e.g., digital sales) and IT technologies (e.g., DevOps and Cloud computing), the architecture of software is shifting from monolithic to microservice rapidly. Benefit from microservice, software development, and delivery processes are accelerated significantly. However, along with many micro services running in the dynamic cloud environment with complex interactions, identifying and locating the abnormal services are extraordinarily difficult. This paper presents a novel system named "Microscope" to identify and locate the abnormal services with a ranked list of possible root causes in Micro-service environments. Without instrumenting the source code of micro services, Microscope can efficiently construct a service causal graph and infer the causes of performance problems in real time. Experimental evaluations in a micro-service benchmark environment show that Microscope achieves a good diagnosis result, i.e., 88% in precision and 80% in recall, which is higher than several state-of-the-art methods. Meanwhile, it has a good scalability to adapt to large-scale micro-service systems.

Keywords: Microservice · Kubernetes · Root cause analytics
Cloud computing

1 Introduction

Nowadays, driven by the emerging business models (e.g., digital sales) and IT technologies (e.g., DevOps and Cloud computing), the architecture of software is shifting from monolithic to microservice [20] architecture rapidly. With microservice architecture, an application is decoupled into many loosely distributed fine-grained services with complex interactions. Usually, these services are connected by some light-weight network protocols such as REST and RPC protocols. But each of them has simple and independent functions following the SRP (Single Responsibility Principle) [20]. The microservice architecture has enabled software systems with new properties such as strong scalability, agile development, fast delivery, and so on. Even though, performance problems are not uncommon

C. Pahl et al. (Eds.): ICSOC 2018, LNCS 11236, pp. 3–20, 2018.
https://doi.org/10.1007/978-3-030-03596-9_1

in microservice systems due to external (e.g., configuration changes) and internal (e.g., software bugs) impairments [7], which brings significant impacts on enterprise revenues. According to [14], Amazon experiences 1% decrease in sales for additional 100 ms delay in response time per request while Google reports a 20% drop in traffic due to 500 ms delay in response time.

To keep microservices running continuously and reliably, it is necessary to detect undesirable performance problems and pinpoint potential root causes. However, it is notoriously difficult to achieve that in microservice environments due to the following challenges:

- **Complex network dependencies**. With a microservice architecture, an application is decoupled into many fine-grained components with an extraordinarily complex network topology. Moreover, to connect different micro services wrapped in a container, an overlay network such as *flannel* is always adopted, which further increases the complexity of performance diagnosis.
- **Continuous integration and delivery.** A microservice system is evolving all the time with continuous integration and delivery technologies. According to a DevOps report from Puppet [1], an enterprise may have 1600 updates a year. That means the anomaly detection and root cause diagnosis procedure should adapt to these changes in order to achieve better results.
- **Dynamic run-time environment.** A microservice system often runs in a containerized environment where the states of containers change frequently. The highly dynamic environment exacerbates the difficulty of performance diagnosis.
- **A large volume of monitoring metrics.** Since so many services co-exist in micro-service systems, the volume of monitoring metrics (e.g., response time) of these services is very large. The paper [23] states that Netflix, Uber and OpenStack has 2,000,000 metrics, 500,000,000 metrics, and 17608 metrics respectively to monitor. How to pinpoint the root causes = from these data is a challenging problem.

Extensive studies have been done to resolve performance diagnosis problems in distributed systems. However, they either (e.g., X-Trace [12], Roots [15]) require to modify the source code of applications or platforms to obtain the service dependencies or cannot adapt to the dynamics of microservice environments (e.g., CauseInfer [7]). In order to address the aforementioned challenges and shortages of previous work, we propose Microscope, a novel system to identify performance problems and infer root causes with causal graphs towards microservice systems. It basically comprises three procedures, namely data collection, causality graph building, and cause inference. Once an anomaly is detected in front-end services, a causality graph which denotes the anomaly propagation paths is constructed automatically. Then the cause inference procedure is triggered to pinpoint root cause with the causal graph. The causality graph is built with no need domain knowledge and instrumenting the application. Microscope leverages a conditioned graph traversing algorithm to locate the root causes rather than a brute-force search, which significantly reduces the volume of performance metrics to process. Moreover, Microscope works in real-time mode

to adapt to the dynamics of microservice environments. We developed a prototype system and validated its effectiveness in a microservices benchmark, namely Sock-shop [2], managed by Kubernetes. The results show Microscope achieves an average 88% precision for root cause identification outperforming several state-of-the-art methods. Meanwhile, it can be applied in a large distributed system without a significant accuracy lose.

The contributions made by this paper are threefold:

- We propose a novel service dependency discovery method through capturing and parsing the network-related system calls, which works automatically to capture the real service instances dependency in real time.
- We provide a **parallelized** service causality graph building method based on the service dependency and service impact graph co-located in a single machine. We can precisely pinpoint the root causes at service instance level with this causality graph.
- We design and implement a prototype of Microscope to infer the root causes of performance problems without domain knowledge and application instrumentation, and achieve a high precision and recall with a low cost.

The rest of this paper is organized as follows. Section 2 presents the system overview and formulation. Section 3 elaborates the details of Microscope. In Sect. 4, we will evaluate Microscope in the controlled environment. And in Sect. 5 we will compare our work with previous work. We discuss the advantages of Microscope and Sect. 6 concludes this paper.

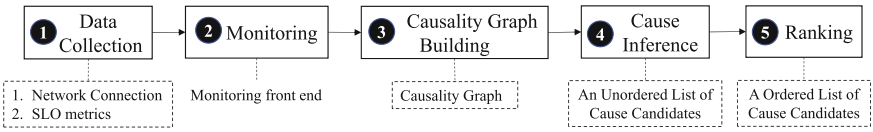

Fig. 1. Workflow of Microscope

2 System Overview and Formulation

Figure 1 shows an overview of Microscope. For data collection, Microscope mainly collects two types of data: network connection information between two service instances and SLO (Service Level Objective) metrics of each service instance. To diagnose system anomalies, Microscope continually monitors the SLO metrics of the front end within a sliding time window. When an SLO violation is detected, the root cause analysis is triggered. In service causality graph building phase, Microscope uses the network connection information and SLO metrics to build a causality graph. Then the cause inference engine starts from the front end and traverses the entire causality graph along with the directed edges. After that, Microscope gets a list of possible root cause candidates. Finally, Microscope calculates a score for each candidate and ranks the candidates with the score.

3 System Design

3.1 Data Collection

In the data collection part, Microscope mainly collects two types of data, namely network connection information between two service instances and SLO metrics of each service instance.

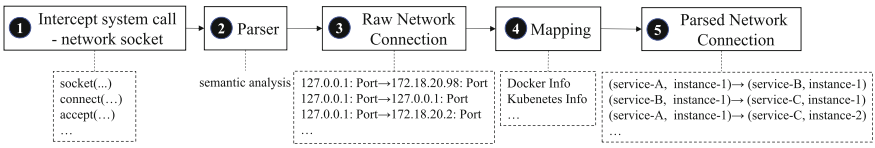

Fig. 2. Workflow of capturing network connection information

Network Connection Information. This type of data is used for causality graph building. Network connection is an important information to represent the real service dependencies. However, most of time, the transmission between client and server is bidirectional. Hence, we will get an opposite connection direction when we observe in different hosts. For example, when we observe in host (192.168.1.2) which sends a request network package, we get a connection (192.168.1.2)→(192.168.1.3). But in host (192.168.1.3) which sends a response network packet, we will get (192.168.1.3)→(192.168.1.2).

In order to address this issue, we introduce a novel method to capture the network connection information by monitoring and intercepting system calls related to network socket, such as *socket()*, *connect()*, *send()* and *recv()*. A network socket is an internal endpoint for sending or receiving data within a computer network. Each socket function relates to a socket variable. When we intercept a socket system call, it parses the socket variable and parameters of this function to get a client IP and a server IP according to the semantic meaning of function name. For instance, if a *connect()* function is intercepted, the local address of socket will be parsed as client IP and the peer address will be parsed as server IP. Another situation is that if a *accept()* function is intercepted, the local address will be parsed as server IP and the peer address will be parsed as server IP. So we can get the direction of network connection and the real dependency between two services.

The complete workflow of capturing network connection information shows in Fig. 2. After the socket parsing, we get the raw network connection whose ends are host IPs rather than service IPs. To know the corresponding service of a host IP, we extract some information from infrastructures such as Docker and Kubernetes. Combining with this information, we map the raw network connection(IP:Port) to parsed network connection ((service name, service instance)). Finally, we get a bunch of records which describe the network connection information from a service instance to another service instance.

SLO (Service Level Objective) Metrics. This type of data is used for detecting whether a service instance is abnormal and ranking root cause candidates. According to our observations, most cloud-native applications that internally generates performance metrics such as throughput for monitoring and maintenance. If these data are not internally available, we can also crawl the service logs to that end. For example, the spring boot framework provides a plug-in of service log for monitoring. Therefore, we can easily get SLO metrics from cloud-native applications in microservice environments. In this paper, we will use a unified SLO metric, namely service request latency which is the service calling time, which exposed by the services themselves. In the future work, we will explore more SLO metrics in microscope for improving the effectiveness. Although it is simple, it works well in Microscope.

3.2 Service Causality Graph Building

In this section, we describe the details of the service causality graph building. The definition of the causality is that 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→Y. In other words, X is a parent of Y, denoted by $X \in pa(Y)$. In a collective variable, if all the parents of Y have been determined, the distribution of Y will be determined and not affected by other variables. In this paper, X and Y represent the SLO metrics of each microservice. Note that in the causality graph, it is not allowed two variables cause each other. So the causality graph can be represented as a DAG (Directed Acyclic Graph).

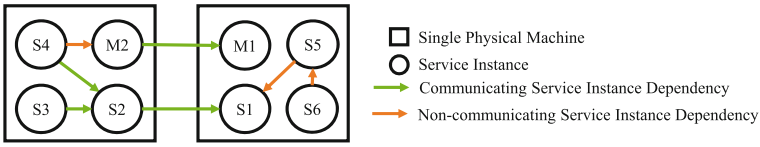

Fig. 3. An example of service causality graph, where S1-S6 and M1-M2 represent unrelated applications with several service instances.

Microscope constructs a causality graph based on the communicating service instance dependencies and non-communicating service instance dependencies. Each node in the causality graph represents a service instance, and the directed edge between two nodes represents a direct "cause-effect" relation between two service instances. Figure 3 shows an example of causality graph generated by Microscope.

Communicating Service Dependency. The first type of dependency represents a dependency relation between two communicating service instances via network. In previous studies, [7,23], they also use the network connection information to construct causality graph. But the direction of dependency between

two service instances are determined by some statistical methods such as Granger Causality [13] and PC-algorithm [16]. So their results strongly depend on the quality of data. Compared with these work, Microscope captures the directed connection information to represent the communicating service instance dependency relation without any statistical error.

To construct communicating service dependency, Microscope uses the *parsed network connection* data directly which already contains two service instances and the direction of dependency in each record. For example, (service-A, instance-1)→(service-B, instance-1) represents (service-A, instance-1) is a cause of (service-B, instance-1). Microscope uses the data collected in the last 10 min, so the causality graph can be updated dynamically, which can exclude some inactive service instances and improve the precision of root cause inference. However, Microscope can also use data for longer periods of time and save static communicating service dependencies, which reduces the cost of building this type of dependencies repeatedly.

Non-communicating Service Dependency. Due to local resource sharing, the service may interfere with other services running in the same node, which is called "non-communication service dependency". For example, if a service instance occupies all the CPU resource of a physical machine, the response time of the other service instances in the same node, especially for computation-intensive ones, is likely affected. Therefore, the SLO metrics changes of other co-located services are also responsible for changes in the SLO metrics of the current concerned services. We construct such relations by a statistical approach.

In one physical machine, the anomalies of Service **A** and Service **B** may be caused by a common anomaly of Service **C**. To model these relations, we adopt causal statistics [22] rather than the pair-wise correlation. Considering the large volume of SLO metrics in microservice environments and the light-weight requirement, we design our algorithm on the basis of PC-algorithm [16], which is more computationally efficient than Bayesian network approaches [11]. To obtain such a DAG, we first construct a skeleton of the DAG, namely an undirected graph. Then we orientate the skeleton with D-separation [16] rules. A causal Markov condition [22] is used to produce a set of independent relations amongst more than two variables and to construct the skeleton of a causality graph. It is 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 leverage a conditional cross-entropy based metric G^2 [22] to qualitatively test whether X is dependent on Y given $\mathbf{Z}$, where X, Y and $\mathbf{Z}$ are disjoint set of variables in V, X and Y are single variables, but $\mathbf{Z}$ can be a set of variables. We choose G^2 instead of other methods like Gaussian independence test [16] as it does not need any assumption on the distribution of each variable. G^2 is defined as

$$G^2 = 2mCE(X, Y|\mathbf{Z}) = \sum_z P(z) \sum_x \sum_y P(x, y|z) log(\frac{P(x, y|z)}{p(x|z) \cdot p(y|z)}), \quad (1)$$

where m is the sample size, $CE(X, Y|\mathbf{Z})$ is the conditional cross entropy of X and Y given $\mathbf{Z}$. As stated in [22], under independence hypothesis, the metric G^2 follows a χ^2 distribution with a degree of freedom equals to

$$(N_X - 1)(N_Y - 1) \prod_{Z' \in \mathbf{Z}} N_{Z'}, \quad (2)$$

where N_X, N_Y and $N_{Z'}$ represent the number of values of variable X, Y and Z' respectively. Hence, via a χ^2 test, we can decide whether the independence hypothesis is accepted. If the *p-value* exceeds the significance level ξ, namely *p-value* $> \xi$ ($\xi = 0.02$ in this paper), the independence hypothesis is accepted otherwise rejected. If X is independent of Y given $\mathbf{Z}$, then $I(X, Y|\mathbf{Z}) = 1$.

PC-algorithm begins with a completely connected undirected graph, then facilitates G^2 to capture all the independence relationships within all variables in pair-wise manner. The following work is to determine the causal directions using D-separation [16] rules, which is demonstrated in our previous work [7] in detail. Due to the limited space of this paper, we cannot show the details of this PC-algorithm. Please refer to the paper [16] for the details of PC-algorithm. On the basis of PC-algorithm, we construct a parallelized algorithm to construct non-communicating service dependencies in micro-service environments more efficiently. When the cardinality of $\mathbf{Z}$ equals 0, namely $|\mathbf{Z}| = 0$, each pair of X and Y is completely independent. Therefore, we leverage "MapReduce" [10] approach to test their independence relations in parallel. Then, we get another undirected graph with significantly reduced edges. If $|\mathbf{Z}| > 0$, the independence tests of X and Y given $\mathbf{Z}$ are independent any more as they share the intermediate results. Under such a condition, we implement a multi-core parallel algorithm in one node. That means we conduct one independence test per core. After the parallelization of PC-algorithm, Microscope can construct the causal relations in real time.

Once an anomaly is detected in the front-end services, the construction procedure of non-communicating service dependency is triggered. We first group all the micro services based on physical machines. Let $(S_1, S_2, \cdots, S_n)$ denote n services, $N_1, N_2, \cdots, N_m$ denote m machines, (S_i, N_j) denote service i locates on machine N_j, so all services are separated into m groups. For each group, we leverage parallelized PC-algorithm to construct service dependency relations. The inputs are time series of SLO metrics of micro services. In this paper, we leverage the mean response time as the SLO metric. Other SLO metrics are also adoptable. 200 data points starting from the abnormal moment are adopted to construct the service dependency graph. The reason why we choose 200 is stated in the sensibility analysis of Sect. 4. One point we observed from our experiments is that some causal directions calculated by PC-algorithm are not consistent with the service dependencies obtained by network analysis. In this scenario, we trust

Algorithm 1. The *parallelized* PC-algorithm

Input: The significance level ξ used to test the conditional independence; the response time metrics of micro services, $\mathbf{R}$=$\{R_1, R_2, R_3, \cdots, R_n\}$, set the maximal cardinality of $\mathbf{Z}$ as 3; the number of CPU cores c;
Output: non-communicating service dependencies DAG, G

/ ∗ ∗ Construct the skeleton of G ∗ ∗ /
Form the complete undirected graph G^u based on $\mathbf{R}$
i=-1
repeat
  i=i+1
  if $i == 0$ **then**
    / ∗ ∗ **Map process** ∗ ∗ /
    Select one pair of (X, Y) from all the combinations
    if $I(X, Y) == 1$ **then**
      Record (X, Y)
    end if
    / ∗ ∗ **Reduce process** ∗ ∗ /
    Collect all pairs of (X, Y) calculated by each Map process
    Remove the edges $X - Y$ recorded by (X, Y) from G^u
    Update G^u
    / ∗ ∗ Calculate service dependencies by multiple process on one machine ∗ ∗ /
  else
    for each $c_i \in (1, 2, \cdots, c)$ **do**
      Fork one process on one machine to conduct in independence tests
      for each $X \in \mathbf{X}$ **do**
        for each $Y \in adj(G^u, X)$ **do**
          / ∗ ∗ $adj(G^u, X)$ represents the set of metrics which are adjacent to X in G^u. ∗ ∗ /
          repeat
            Choose $\mathbf{Z} \subseteq adj(G^u, X) \setminus \{Y\}$ with $|\mathbf{Z}| = i$
            if $I(X, Y|\mathbf{Z}) == 1$ **then**
              Remove $X - Y$ from G^u
              Update G^u
              Make the separate set $\mathbf{S}(X, Y) = \mathbf{Z}$
            end if
          until edge $X - Y$ is removed or all $\mathbf{Z}$ with $|\mathbf{Z}| = i$ have been chosen.
        end for
      end for
    end for
  end if
until $|adj(G^u, X)| \leq i, \forall X$ or i == 3
The skeleton,$G^s = G^u$
/ ∗ ∗ Orient the directions in G^s with D-Separation rules∗ ∗ /
for all pairs of nonadjacent variables X, Y with common neighbor Z **do**
  if $Z \notin \mathbf{S}(X, Y)$ **then**
    Replace $X - Z - Y$ in G^s with $X \rightarrow Z \leftarrow Y$
  end if
end for
Orient $Y - Z$ as $Y \rightarrow Z$ whenever there is an arrow $X \rightarrow Y$
Orient $X - Y$ as $X \rightarrow Y$ whenever there is chain $X \rightarrow Z \rightarrow Y$
Orient $X - Y$ as $X \rightarrow Y$ whenever there are two chains $X - Z \rightarrow Y$ and $X - L \rightarrow Y$
Finally, output G

the result obtained by the latter as it is the ground truth. Therefore, to avoid this scenario, we preset the connections and directions of edges that can be obtained by network analysis in the undirected graph.

For the sake of clarity, we show the pseudo code of our algorithm in Algorithm 1. The computational complexity of Algorithm 1 is dominated by the DAG skeleton construction procedure. The worst case is bounded by $O(nmax\{p^q, p^2\})$ [17], where n is the data length of each metric, p is the number of metrics, q is maximal size of the adjacent sets, i.e., the cardinality of $\mathbf{Z}$, $|\mathbf{Z}|$. When q is large, the complexity increases exponentially. However, from the real data, we

observe that q always stays at a low level $q < 5$. Hence this complexity is affordable. We set $q = 3$ in this paper. The constructed non-communicating service dependencies will be merged with communicating service dependencies to form the final service causal graphs.

3.3 Cause Inference

We summarize the process of cause inference in Algorithm 2. Microscope continually monitors the SLO metrics of the front end. When an SLO metric is detected as abnormal, the cause inference is triggered. Then the cause inference engine starts from this abnormal node in the causality graph and traverses the causality graph along the opposite direction of edges, which represents the dependency between two service instances. When a node is abnormal, the cause inference engine will check its neighbors. If all the neighbors are normal, the current node will be added to the set of root cause candidates and the engine stops traversing its children. If there exist one or more abnormal children, the

Algorithm 2. The *cause inference* algorithm

Input: An original abnormal service instance, rootNode; A causality graph DAG, G;
Output: A ordered list of root cause candidates

```
// Find root cause candidates
stack ← Stack(); candidates ← List()
stack.push(rootNode)
while stack is not empty do
    node ← stack.pop()
    // adj(G, X) represents the neighbors which are adjacent to X in G.
    if adj(G, node) is empty then
        candidates.append(node)
        continue
    end if
    children ← List()
    for each neighbor ∈ adj(G, node) do
        if neighbor is abnormal then
            children.append(neighbor)
            stack.push(neighbor)
        end if
    end for
    if children is empty then
        candidates.push(node)
    end if
end while
// Scoring for each candidates
candidatesScore ← Dict()
for each candidate ∈ candidates do
    candidatesScore[candidate] ← scoring(rootNode, candidate)
end for
return keys of candidatesScore sorted by value
```

cause inference engine will continue to traverse these abnormal children. When the traversal is finished, the engine gets a set of root cause candidates. Then the engine calculates a ranking score for each root cause candidates. Finally, the engine gets an ordered list of root cause with ranking score and the top one in the list is considered to be the real root cause. In this paper, we use *three-sigma rule of thumb* to detect if a service instance is abnormal and use the correlation between the front end and the abnormal service instances as the ranking score.

To detect anomalies, we use a *three-sigma rule of thumb*. The so-called three sigma rule of thumb expresses a conventional heuristic, that is, almost all values are considered to be within the three standard deviations of the mean. Therefore, it is empirically useful to treat the possibility of 99.7% as near certainty. In mathematical notation, this fact can be expressed as $Pr(\mu - 3\sigma < x < \mu + 3\sigma) \approx 0.9973$. If a value of SLO metric is not within the three-sigma interval of the last 10 min, we think this service instance is abnormal. Although it is simple, it works in this paper and can adapt to system changes in real time.

To sort the root cause candidates and get the most possible real root cause, we calculate the pearson correlation coefficient of SLO metrics between the front end and each candidate as the ranking score. According to our observations, if two service instances have a strong dependency relationship, the curves of service request latency of them are very similar. With this method, Microscope has the ability to diagnose real root causes even when several system faults happen at the same time.

4 Experimental Evaluation

Experiment Settings. Microscope is evaluated in a self-constructed distributed system. The controlled system contains four client physical servers that host the benchmark. Each physical server machine has a 12-core 2.40GHz CPU, 64GB of memory and runs with Ubuntu 16.04 OS. We evaluate Microscope in Kubernetes platform. Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications, which is one of best platform for developing and running micro services.

Data Collection. To capture the network connection information, we develop several tools from scratch. The captured network connection information is recorded to a local log file. For forwarding and centralize log files, we use Filebeat which offers a lightweight way to forward and centralize logs and files. Meanwhile, we use elasticsearch to save all the network connection information harvested by Filebeat from each physical nodes. To collect service request latency metrics, we use Prometheus, an open-source systems monitoring and alerting toolkit, to monitor the services instances. The sample interval in service request latency metrics is 1 s.

Benchmark. Sock-shop [2] is a microservices demo that simulates the sale of socks of an e-commerce website which is a widely used micro-service benchmark designed to help demonstrate and test microservices and cloud-native technologies. It provides some key properties (e.g. Polymorphism) that a micro-service

system should have. It contains 13 services in the form of microservices. In each services instance, we configure the CPU resource limited to 1 GHz and the memory resources limited to 1GB. The replicas of service instances are set to 1–3. The total service instances are 36. Furthermore, sock-shop contains a load generator, which defines user behavior, to simulate the query per second(QPS) on a website with simultaneous users, so we adopt this load generator to generate the workload and configure the load to keep the QPS about 5000.

Fault Injection. Our work focuses on locating the root cause service instances. The service instance in Kubernetes is a Pod which contains one or more containers. To mimic the real performance problems, we inject faults to containers in Pods. We inject the following faults: (1) CPU exhausting: we use stress [3], a deliberately simple workload generator for POSIX systems, to exhaust CPU resources in injected containers; (2) NetworkJam: we use "tc", a traffic control tool in Linux, to delay the network packets; (3) ContainerPause: for simulating the status of hangup of a service instance, we pause the container with"PAUSE" command of Docker. We do not kill containers because Kubernetes will recreate a new replicated Pod immediately. For evaluating the effectiveness of Mircoscope, each fault mentioned above will be injected in each service instances more than 5 times and last 1 min. The total number of fault injections is 240.

Evaluation Metric. We use the following metrics to evaluate the effectiveness.

- *Precision at top K*(PR@K) indicates the probability that the root cause appears in the top K of ranking list if the cause inference is triggered. It is important to capture the root cause at a small value of K, thereby resulting in lesser number of service instances to investigate. Here we use K=1,2,3.
- *Recall at top K*(Recall@K) is the fraction of real cause that has been retrieved at top K of the ranking list over the total amount of fault injections. Here we use K=1,2,3.

4.1 Effectiveness Evaluation

Microscope strongly relies on the causality graph, Fig. 4 illustrates a causality graph obtained by Microscope in about 10 min when running a load generator. Different colors represent different applications. Via comparison with the

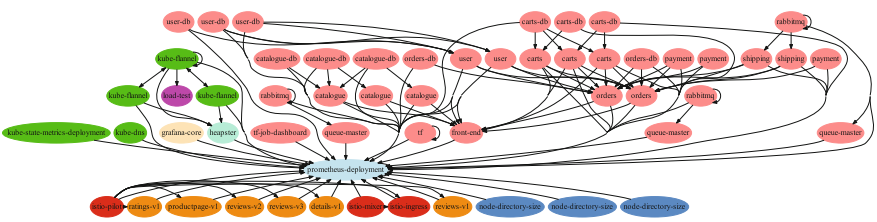

Fig. 4. The sample causality graph generated by Mircoscope in our system

ground truth, we find that all the relations shown in Fig. 4 are indeed the service call relations without any exception, which demonstrates that Microscope can build a reasonable causality graph in real time without domain knowledge and instrumenting the application.

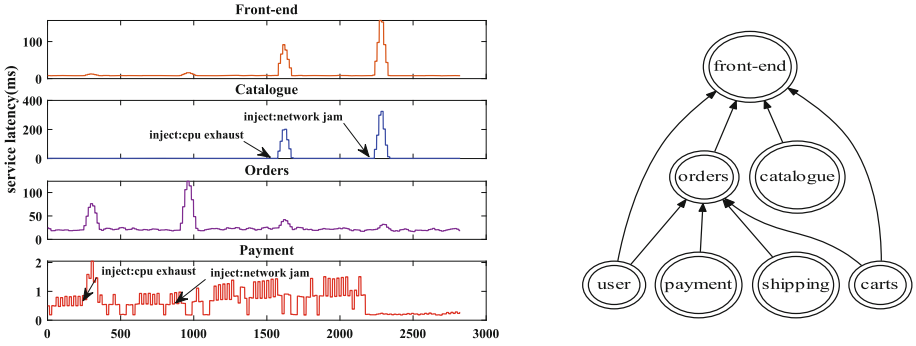

Fig. 5. A view of service latency and causality graph of Sock-Shop

Figure 5 shows the curves of service request latency of four service instances when 4 faults are injected respectively and the dependent relations between service instances of sock-shop. To simplify the description, we keep only one replica of each service. The curve of front-end and catalogue services in Fig. 5 show that if a service is abnormal, it does affect other services which it depends on. But it is not the case for the injected faults in payment service. The orders which the payment depends on is strongly affected. However, it has a very subtle effect on the front end. This is because Kubernetes has a load balancing mechanism. So the cause inference may not be triggered and the precision and recall are low on these types of services.

Figure 6 demonstrates the results of PR@1 and recall on several services of Sock-shop. From Fig. 6, we observe that most of the PR@1 fall in the range 70%–100% in different services and faults, except the shipping and payment service. One of the exceptions is we lack the results of shipping and payment

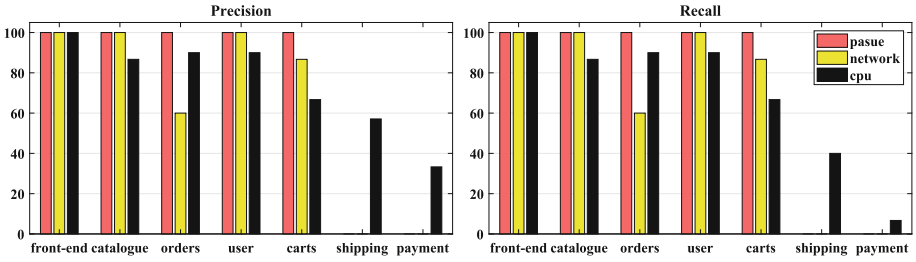

Fig. 6. The results of PR@1 and Recall@1

with respect to network and container pause. This is because (i) the service latency is collected by the process in the Docker container, (ii) these services don't request any other services, so the service latency is returned immediately from the process without passing through the container network, (iii) we inject the network delay to block the network in container rather than the process. So the fault injection doesn't work on these services. The other one of exceptions is the results of CPU exhausting on shipping and payment. This is because (i) the cause inference may not be triggered which mentioned above, (ii) both shipping and payment are not computation-intensive, which the usage of CPU in these service instances is the only 5 mHz. So even though we use *stress* tool to exhaust the resources of CPU, these service instances still have a good response time. Significantly, we get 100% at the fault of container pause, because it causes the service latency missing and it's a strong feature to diagnose.

Table 1. Performance

	Catalogue	Front-end	Carts	Orders	User	Shipping	Payment
CPU exhausting							
PR@1	86.7	100	66.7	90.0	90.0	57.1	33.3
PR@2	93.3	100	66.7	90.0	100	57.1	33.3
Recall@1	86.7	100	66.7	90.0	90.0	40.0	6.7
Recall@2	93.3	100	66.7	90.0	100	40.0	6.7
Network jam							
PR@1	100	100	86.7	60.0	100	-	-
PR@2	100	100	86.7	60.0	100	-	-
Recall@1	100	100	86.7	60.0	100	-	-
Recall@2	100	100	86.7	60.0	100	-	-
Container pause							
PR@1	100	100	100	100	100	-	-
PR@2	100	100	100	100	100	-	-
Recall@1	100	100	100	100	100	-	-
Recall@2	100	100	100	100	100	-	-

Table 1 demonstrate the performance of Microscope in Sock-shop on different service and fault. It shows that Microscope can achieve an average 80%–95% precision and recall for those computation-intensive and network-sensitive services.

4.2 Comparisons

To validate the effectiveness of Microscope thoroughly, we compare it with several state-of-the-art methods including TAN [9], NetMedic [18], Sieve [23], Roots

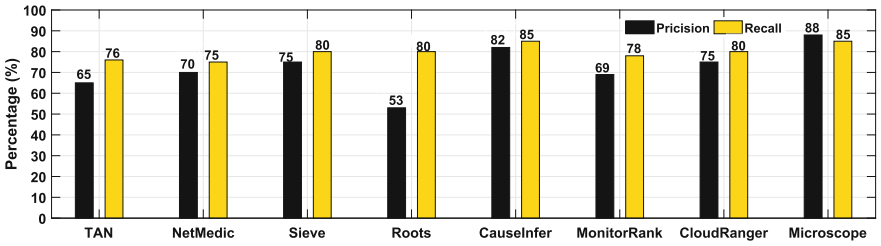

Fig. 7. The comparison results in PR@1 and Recall@1

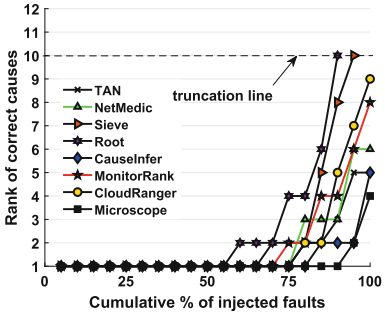

Fig. 8. Rank comparison result

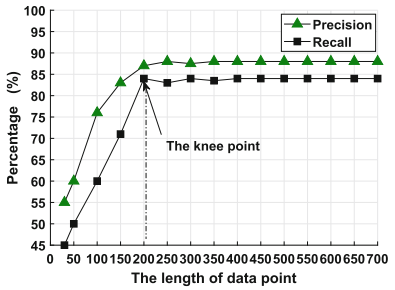

Fig. 9. The Diagnosis Result VS data length

[15], CauseInfer [7], MonitorRank [19], and CloudRanger [24]. To compare with TAN, we replace our service dependency construction approach with Tree Augmented Bayesian Network approach; To compare with NetMedic, we leverage NetMedic's state correlation approach to estimate service dependencies; To compare with Sieve, we adopt *sysdig* to obtain the static service call graph, a bi-directed graph then leverage Granger Causality to obtain the dynamic service dependencies with response time metrics; To compare with Roots, we implement the four root cause identification approaches mentioned in Roots. But since Microscope cannot track each request, we leverage the aggregated response time of requests instead of the response time of one single request to identify root causes; To compare with CauseInfer, we capture the network packets and leverage lag correlation to find service dependencies; To compare MonitorRank, we use a random walk approach to find the root causes which has been implemented in our previous work [24]; To compare with CloudRanger, we leverage PC-algorithm to construct service dependencies with response time metrics of micro services. Figure 7. shows the comparison result in PR@1 and Recall@1. From this figure, we observe that Microscope achieves a significantly better result, 3% higher than CauseInfer, 13% higher than CloudRanger, 35% higher than Roots, in PR@1. Roots identifies the service which contributes the most variance of the abnormal service as the root cause. However, in our experiments, we observe that Roots always finds the first upstream service as the culprit rather

than real root causes. That is why its result is not very good. Similarly, MonitorRank also puts the first upstream service in the first rank. Compared with CauseInfer, CloudRanger, and Sieve, Microscope constructs the service dependencies by analyzing the network connection events rather than calculating the statistical correlations, which is closer to the ground truth. From the perspective of the rank of correct causes, Fig. 8 shows the comparison result between different approaches when $rank <= 10$. From Fig. 8, we observe that Microscope can find 88% of injected faults at Rank 1, which outperforms other approaches.

4.3 Discussion

Overhead. Table 2 shows the overhead of Microscope. Data Collection module takes about 8% CPU utilization as we collect the network connection information and service latency. Overall, Microscope is a light-weight tool for monitoring and pinpointing the abnormal service instances.

Table 2. The overhead of Microscope

System module	CPU Cost
Data collection(Network connection)	$8\% \pm 2\%$ CPU utilization(Single cpu core)
Data collection(SLO metrics)	$4\% \pm 1\%$ CPU utilization(Single cpu core)
Causality graph building	10 s(Single physical node)
Cause inference	2 s(Single physical node)

Sensibility. To evaluate the sensibility to the data length in constructing non-communicating service instances dependency, we conduct several experiments. Figure 9 illustrates the changes in the data length increasing. From this figure, we observe that Precision and Recall stay at a low level when the length of data is lower than 200. Because the service causal graphs calculated by PC-algorithm are not precise enough. However, after "the knee point", the service causal graphs keep constant and the diagnosis results do not change anymore.

Scalability. Microscope is easy to scale up whether we add new service instances or machines in a large distributed system. In order to test the scalability of the system, we deployed more replicas of services instances with Sock-shop. From Fig. 10, we can see that Microscope has only 7% accuracy degradation from 36 service instances to 120 service instances, showing good scalability.

Indeed, the micro-service system provides some fault-tolerance mechanisms such as circuit-breaker, which aim to avoid cascading errors. Also some real microservice infrastructures in kubernetes provide features such as scalability, restart or load balance policies. Considering these, Microscope will resolve these problem in the future work.

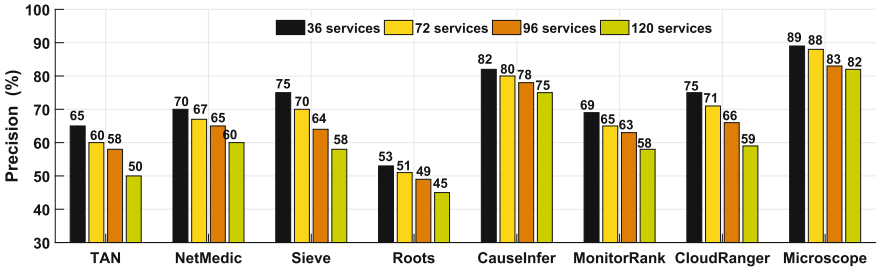

Fig. 10. PR@1 vs serivce instances number

5 Related Work

RCA(Root cause analysis) in large distributed systems with many services is a frustrating task. To pinpoint the root causes of performance problems, a large number of researchers are dedicated to this area. In the following, we present the relative work briefly.

Trace Based Work. Many famous tools fall into this category such as Magpie [4], X-trace [12], Pinpoint [6] and The Mystery Machine [8]. These tools can accurately record the execution path of the program and locate the error by detecting source code or binary code. It's helpful to debug distributed applications. However, deploying these tools is a daunting task and requires administrators to understand the code well. Compared to them, Microscope doesn't need instrumenting the source code and domain knowledge. So it can be easily deployed and used. Although Microscope does not detect true software bugs, it does provide some hints. Roots [15] is a near real-time monitoring and diagnostics framework for web applications deployed in a PaaS cloud without instrumenting the application code. However, Roots only consider web applications and must modify the front-end request server(typically a software load balancer) by instrumenting the request identifier to an HTTP request, which hinders it to be widely used.

Signature Based Work. These methods employ a supervised learning algorithm to classify performance anomalies under several typical scenarios. CloudPD [21] uses a layered online learning approach to deal with the higher occurrence of faults in clouds; Fingerprint [5] provides the basis for automatic classification and identification of performance crises in a data center; All of these methods require labeled data or problem tickets and have limited generalization on new anomalies. While Microscope is an unsupervised method, so it's able to capture new anomalies and have a good generalization.

Dependency Graph Based Work. In recent years, performance diagnosis based on dependency graphs has become a surge. Using graph models, we can not only understand the problem propagation path but also infer the root cause. Sieve [23] infers metrics dependencies between distributed components of the system by using Granger Causality tests. CauseInfer [7] automatically builds a two-layered hierarchical causality graph and the dependency direction is determined

by a lag correlation. This method strongly depends on data and the dependent direction is generated by some statistical method. Compared to them, Microscope uses the network connection information, which is real dependencies, to build the causality graph. TAN [9] is used to infer performance issues at a metric level, but it is not effective enough due to a lack of causality.

Finally, compared to much excellent performance monitoring and analysis tools such as Splunk, CaleCd, and IBM Tivoli Gardens, Microscope provides more advanced analysis capabilities, such as causality graph building and root cause inference techniques, without the need for human intervention.

6 Conclusion and Future Work

This paper designs and implements Microscope, a novel system for helping operators and developers with pinpointing the root causes in microservices environments. Microscope automatically builds a causality graph without domain knowledge and instrumenting the application and infers the root causes along the directed edges in the graph. The experimental evaluation shows that Microscope not only can achieve a high precision and recall for performance diagnosis but also is lightweight and can scale up readily in large distributed systems.

As part of future work, we plan to explore using more types of SLO metrics and methods of ranking candidates to improve the effectiveness and generalization ability of Microscope to adapt more different types of services. We will make Microscope more lightweight in order to work in real-time and validate Microscope in some real microservice systems with more kinds of faults.

Acknowledgment. The work described in this paper was supported by the National Key R&D Program of China (2018YFB1004804), the National Natural Science Foundation of China (61722214) and the Guangdong Province Universities and Colleges Pearl River Scholar Funded Scheme 2016.

References

1. https://puppet.com/blog/2017-state-devops-report-here
2. https://microservices-demo.github.io/
3. https://people.seas.harvard.edu/~apw/stress/
4. Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using magpie for request extraction and workload modelling. In: OSDI, vol. 4, pp. 18–18 (2004)
5. Bodik, P., Goldszmidt, M., Fox, A., Woodard, D.B., Andersen, H.: Fingerprinting the datacenter: automated classification of performance crises. In: Proceedings of the 5th European conference on Computer systems, pp. 111–124. ACM (2010)
6. Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E.: Pinpoint: problem determination in large, dynamic internet services. In: Proceedings of International Conference on Dependable Systems and Networks (DSN 2002), pp. 595–604. IEEE (2002)
7. Chen, P., Qi, Y., Zheng, P., Hou, D.: Causeinfer: automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems. In: INFOCOM, 2014 Proceedings IEEE, pp. 1887–1895. IEEE (2014)

8. Chow, M., Meisner, D., Flinn, J., Peek, D., Wenisch, T.F.: The mystery machine: end-to-end performance analysis of large-scale internet services. In: Proceedings of the 11th symposium on Operating Systems Design and Implementation, pp. 217–231 (2014)
9. Cohen, I., Chase, J.S., Goldszmidt, M., Kelly, T., Symons, J.: Correlating instrumentation data to system states: a building block for automated diagnosis and control. In: OSDI, vol. 4, pp. 16–16 (2004)
10. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
11. Ellis, B., Wong, W.H.: Learning causal Bayesian network structures from experimental data. J. Am. Stat. Assoc. **103**(482), 778–789 (2008)
12. Fonseca, R., Porter, G., Katz, R.H., Shenker, S., Stoica, I.: X-trace: a pervasive network tracing framework. In: Proceedings of the 4th USENIX conference on Networked systems design implementation, pp. 271–284. USENIX Association (2007)
13. Granger, C.W.: Investigating causal relations by econometric models and cross-spectral methods. Econom.: J. Econom. Soc. **37**, 424–438 (1969)
14. Ibidunmoye, O., Hernández-Rodriguez, F., Elmroth, E.: Performance anomaly detection and bottleneck identification. ACM Comput. Surv. (CSUR) **48**(1), 4 (2015)
15. Jayathilaka, H., Krintz, C., Wolski, R.: Performance monitoring and root cause analysis for cloud-hosted web applications. In: Proceedings of the 26th International Conference on World Wide Web, pp. 469–478. International World Wide Web Conferences Steering Committee (2017)
16. Kalisch, M., Bühlmann, P.: Estimating high-dimensional directed acyclic graphs with the PC-algorithm. J. Mach. Learn. Res. **8**(Mar), 613–636 (2007)
17. Kalisch, M., Bühlmann, P.: Robustification of the PC-algorithm for directed acyclic graphs. J. Comput. Graph. Stat. **17**(4), 773–789 (2008)
18. Kandula, S., Mahajan, R., Verkaik, P., Agarwal, S., Padhye, J., Bahl, P.: Detailed diagnosis in enterprise networks. ACM SIGCOMM Comput. Commun. Rev. **39**(4), 243–254 (2009)
19. Kim, M., Sumbaly, R., Shah, S.: Root cause detection in a service-oriented architecture. In: ACM SIGMETRICS Performance Evaluation Review, vol. 41, pp. 93–104. ACM (2013)
20. Newman, S.: Building Microservices, 1st edn. O'Reilly Media Inc., Sebastopol (2015)
21. Sharma, B., Jayachandran, P., Verma, A., Das, C.R.: CloudPD: problem determination and diagnosis in shared dynamic clouds. In: 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 1–12. IEEE (2013)
22. Spirtes, P., et al.: Causation, Prediction, and Search. MIT press, Cambridge (2000)
23. Thalheim, J., et al.: Sieve: actionable insights from monitored metrics in distributed systems. In: Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, pp. 14–27. ACM (2017)
24. Wang, P., et al.: Cloudranger: root cause identification for cloud native systems. In: Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2018). IEEE (2018)