T-Rank:A Lightweight Spectrum based Fault Localization Approach for Microservice Systems

Zihao Ye
School of Computer Science
and Engineering
Sun Yat-sen University
Email: yezh8@mail2.sysu.edu.cn

Pengfei Chen*
School of Computer Science
and Engineering
Sun Yat-sen University
Email: chenpf7@mail.sysu.edu.cn

Guangba Yu
School of Computer Science
and Engineering
Sun Yat-sen University
Email: yugb5@mail2.sysu.edu.cn

***Abstract*—The cloud-native system is shifting from traditional monolithic architecture to microservice architecture because of loosely coupling, better maintainability and availability, faster deployment, and richer ecology brought by it. Except for these advantages, it still has an inevitable weakness–the communication over RPC (Remote Procedure Call) between services makes the system performance more unpredictable. Moreover, the complex interactions amongst services make it hard to reveal the root cause of performance issues. To address this challenge, we propose a lightweight spectrum-based performance diagnosis tool, named T-Rank. T-Rank provides the ranked suspicious score in a list of microservices to localize root causes with very few resources. We demonstrate the high accuracy and the low cost of T-Rank by conducting experiments with the data collected from a real-world production microservice system. Moreover, comparison results show that T-Rank outperforms other state-of-the-art approaches.**

***Keywords*-Tracing; Microservice; Root cause analysis; Spectrum Analysis;**

I. Introduction

Nowadays, it is hard for traditional monolithic software architecture to satisfy the requirement of the IT business since it is time-consuming and laborious to maintain and extend. With the popularity of cloud computing, more and more newborn applications are cloud-native. Driven by these IT tendencies, the microservice architecture has been widely chosen. The microservice architecture decomposes an application into many services that run individually and communicate with each other by network [1], and this brings the application with larger scalability, faster development, and richer language ecology [2]. For microservice irresistible charm, many companies deploy their production as a large-scale microservice system, such as Overleaf [1], Netflix [2], and Uber [3] [3].

However, there are still some problems with microservice. The decomposition of the application makes it easier to extend but also weakens the reliability of connection because of network fault, meaning its reliability is inferior to the application using in-memory calls. And the network latency also degrades the performance of the microservice-based application for the reason that it takes more time in communication. According to the previous study [4], even for a small e-commerce company with a daily sales of \$100, 000, a 1-second page delay could lead to about 7% loss in sales annually. Besides the network delay, internal features such as program bugs and source exhaustion will result in undesirable performance. The costs of modifying services and fixing errors are much lower in the microservice-based system because the “you build it, you run it” principle strengthens its maintainability [1]. But the most time consuming and laborious problems are anomaly detection and root cause localization. The loosely coupled structure and multi-instance services for load balance and tolerance lead to complex system topology. The challenges it brings are as following aspects.

- **Complex service dependency**. In the microservice-based system, the services are decomposed into fine-grained components that communicate by the network. For load balance and availability, most of the services run multiple instances at the same time. The calls between these instances construct a complex service dependency graph.
- **Dynamic system architecture**. The characteristic of the microservice is that its updates are frequent. And the logical dependencies between services also change with updates. New services will be added and outdated services will be abandoned, resulting in a more dynamic architecture. How to adapt to the frequent updates and deployments will be an important breakthrough.
- **Various and large-volume monitoring metric**. To diverse services, there is a diverse set of metric such as configuration metric (e.g., MemoryLimit and CpuLimit), resource metric, and so on [5]. But some metric are helpless when analyzing the root cause. It needs a great deal of preliminary domain knowledge for operators to distinguish the relevant metric from so many kinds of metric, but obviously it is impractical.

Many existing methods try to construct the real-time service dependency graph and analyze the root cause based on it. To build the real-time service dependency graph, many approaches [6]–[8] have proposed different methods in their model based on conditional independence. However, they take plenty of computing resources to construct the service dependency graph, and a little noise during the construction

[1]Overleaf, https://www.overleaf.com/
[2]Netflix, https://www.netflix.com/
[3]Uber, https://www.uber.com/global/zh/sign-in/

may make a great difference in the final result. And the method, such as TABC [9], relies on the static system topology can not adapt to the dynamic and complex dependency in microservice-based system. The neural network method [3] has low robust facing the frequent iteration.

To address the challenges above, we propose a lightweight and fast diagnosis system, namely T-Rank, to localize root causes in microservice systems. T-Rank works based on the spectrum based fault localization (SBFL) algorithm and only relies on the most common metric (i.e., the latency in completing the request), which overcomes the various metric and dependency graph building challenge. And the core of the T-Rank, the SBFL algorithm, is a statistical algorithm that does not need any prior training and the causal graph of the system. T-Rank is a tool that gives a hint to system operators by providing a ranked suspicious score list of service instances helping them to localize the root cause more effectively when an anomaly happens. It takes a little time for T-Rank to give the ranked suspicious score list of all service instances in the systems. T-Rank will update the list constantly along with the system running. Our contributions are as follow.

- We apply the SBFL algorithm in microservice root cause localization and build a diagnosis system, T-Rank, which can analyze the root cause without the service dependency graph in real-time.
- We verify the effectiveness of T-Rank by setting the experiments with the metric data with injected faults collected from a production microservice system. T-Rank can achieve 93% recall when work online at a low cost and it outperforms other state-of-the-art methods.

The organization of the paper is shown as follows. Section 2 is about the background of tracing and SBFL algorithm, and the motivation why we select SBFL to diagnose the performance problems of microservice-based systems. Section 3 is the design of T-Rank. Section 4 is the experiment to verify the effectiveness of the system. In Section 5, we will introduce the related work in the microservice root cause localization. Section 6 concludes the paper.

II. Background and Motivation

A. End-to-end Tracing

Tracing [10] is an excellent method for system operators to understand how the system running and which component joins in the program execution, especially when the system architecture is quite complex. The most common example of tracing is the Linux command *traceroute* [11], which shows the gateways passed between the source host of the packet and the target host. Tracing in the microservice-based system is similar. Utilizing the sidecar [12] model of microservice, we can record some data when a service instance calls another one or the caller's request is done. These recorded data are tracing data and we can know a lot of valuable information that helps us understand the system behavior, such as the call relationship between instances. The extremely famous tracing tools like X-trace [13], Zipkin [14] and Jaeger [15], are well popular and widely used in many systems.

For utilizing the tracing tool, the system developers need to add the call of tracing API to the code. However, if the standards of the tracing API are different in different tracing tools, it will bring great trouble to the system developer for the reason that they have to replace the code of calling tracing API when they try to use another tracing tool. To deal with the possible problem brought by this logical coupling, the open-source project OpenTelemetry [4] of the CNCF provides a single standard and dominates the microservice end-to-end tracing landscape. There are already a lot of tracing tools supporting its standard.

B. Spectrum Based Fault Localization

The Spectrum Based Fault Localization(SBFL) [16]–[18] algorithm is a statistic algorithm utilized in program debugging. Collecting coverage information over different elements during test execution, SBFL calculates the suspicious score of each element with a risk evaluation formula. The spectrum is the coverage of the element in the test case. The basic idea of SBFL is that the element most frequently executed in the failed test cases should be responsible to the program fault. Different test cases have different spectrums. If an element is always in the different spectrums of the failed test cases, it is so suspicious that the programmer should inspect whether there is a bug in it. According to the suspicious score, the elements are sorted into a list. To fix the bug, the programmer excludes the suspicious elements according to this list, which could save them a lot of effort.

C. Motivation

Comparing the architecture of the program and the microservice-based system, we find that they are quite similar. Every component in the microservice-based system provides its special service and call other components when in need, just like the function in the program. The main difference between them is that the components of the microservice-based system communicates by network and the functions by memory. The management of the microservice-based system is growing more complex because of functionally decomposing large systems into a set of independent services [1]. But its nature is still a software, meaning that the fault localization approaches for program debugging may work well on its performance diagnosis. The SBFL algorithm has achieved great success in software debugging [19], [20]. When looking for the appropriate fault localization approach based on tracing data, we notice the SBFL and think it is worth a try. Thus we select a spectrum-based approach to diagnose the performance problems of microservice-based systems.

III. System Design

A. System Overview

Our main purpose is to make use of tracing data between different services in the system to build up a lightweight fault localization tool at the container level. Therefore we propose

[4]OpenTelemetry, https://opentelemetry.io

T-Rank. T-Rank consists of four parts: tracing data collection from the microservice-based system, tracing data preprocess, anomaly tracing data detection and container suspicious score calculation. Fig. 1 shows the architecture of T-Rank. First, we collect the tracing data into the tracing database (DB) when a service instance calls another one. And then, we group the tracing data by the *TraceId*. The same *TraceId* means those tracing data are generated by the same client's service request to the microservice-based system. All the tracing data from the same client's service request form a **tracing chain**. After tracing data integration, we classify the tracing chain by the type and quantity of the containers in it. And then, after calculating its theoretical latency time and comparing the result with its truth latency, we will label the tracing chain as normal if its truth latency time is close to the theoretical one, otherwise it will be labeled as abnormal. Finally, the system will statistic 4 parameters of every container which the SBFL algorithm needs, and then calculate the suspicious score of every container to localize fault containers. The system will provide the ranked suspicious score list of containers to support the system operators to eliminate fault as soon as possible.

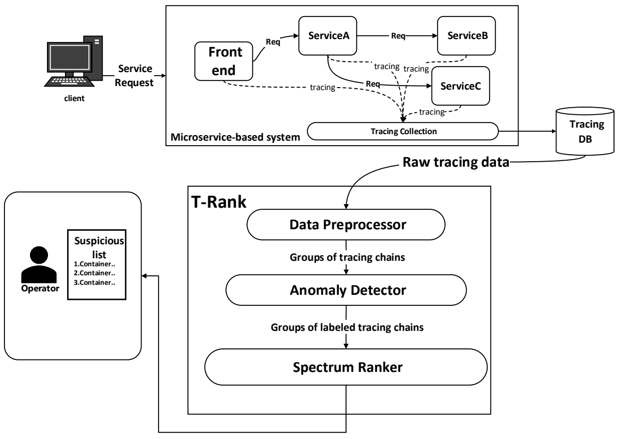

Fig. 1. T-Rank Overview

B. Tracing Collection

In T-Rank, we trace the call between services by deploy a sidecar container for every service instance which is responsible to generate tracing data when an request is launched and done. Once a service calls another service, the sidecar container of caller service will record this call as one piece of tracing data and when the call is done, it records the request completion latency time. Then the generated tracing data will be collected and stored in the Tracing DB. The process of recording tracing data and sending the request to callee service are asynchronous, so it costs little recourse and does not interrupt the system work. The detailed compositions of the tracing data collected are as follow Table. I:

The tracing data consists of *StartTime, ElapsedTime, isSuccess, TraceId, Id, Pid and Cmdb_id*. *TraceId* is the tag we use to correlate the pieces of tracing data scattered on different servers. The initial *TraceId* is always generated by the front-end service instance when a client issues a request to the microservice system. And then, all the request derived by this client request will use it as their *TraceId*. *Id* is the unique symbol that distinguishes one tracing data itself from others in the tracing chain and it is generated when the request happens. *Pid* is the Id of the caller. *Cmdb_id* is the id of the container where the service instance is running. *StartTime* is the 13-bit timestamp that records when the call happens and *ElapsedTime* is the time the call takes to finish the request. *isSuccess* is the label marking whether the call is successfully processed. However, it is always *True* because of system fault tolerance which makes it helpless when detecting the anomaly.

TABLE I
TRACING DATA SAMPLE

StartTime	ElapsedTime	isSuccess	TraceId	Id	Pid	Cmdb_id
2020.05.11.00.12.32	200	TRUE	**47**	**21a**	7b	os_021
2020.05.11.00.12.40	17	TRUE	58	61a	12c	docker_07
2020.05.11.00.13.32	41	TRUE	**47**	80s	**21a**	docker_01
...	...	...	...	...	...	...

In T-Rank, *TraceId, StartTime, ElapsedTime* and *Cmdb_id* are the most important items we rely on. By making use of these items, T-Rank implements the next integration and classification of tracing data.

C. Data Preprocessor

The tracing data is stored in a Tracing DB (e.g., Elasticsearch) and sorted by the timestamp, which means that it is separated and every piece of the data only represents one component state. T-Rank continuously draws the tracing data whose timestamp is in the current time window W from Tracing DB to analyze. However, in the SBFL algorithm, it needs a large number of test cases that cover different components for further analysis. The separated tracing data in DB does not meet SBFL's requirement obviously.

To apply the SBFL algorithm on the tracing data, T-Rank treats every client service request as a test execution. T-Rank integrates the separated tracing data in the current time window W (default 5 minutes) according to their *TraceId*. The tracing data with the same *TraceId* are from the same client service request. T-Rank calls the integrated tracing data with the same *TraceId* as **tracing chain** T. During the integration, T-Rank also filters the useless information like *Id* to reduce cost of following process. The tracing chain has the following property: *Container_Dict*, *StartTime* and *ElapsedTime*. *Container_Dict* is a key-value list storing the type and quantity of containers in tracing chain, such as $\{$*docker_01:5,docker_06:2,os_22:3*$\}$. *StartTime* is the time when the front-end service receive client's service request and *ElapsedTime* is the sum of time to complete client service request.

T-Rank classifies the tracing chains by their *Container_Dict*. In the microservice system, one service is always deployed in several containers at the same time for load balance and fault tolerance. Although their service topologies are the same, two client requests may go through different containers. What is more, most of the time, every service only provides one special function, so every container always completes the request

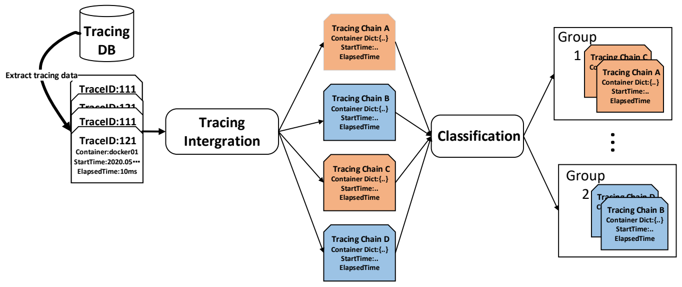

Fig. 2. Data preprocessor.After integration and classification, the raw tracing data become tracing chains and are divided into different group G according their *Container_Dict*.The tracing chains with same color have the same *Container_Dict*

within a time range. But affected by some extra features, for example, the host capacity, some containers providing the same service may have a difference in the latency time. Therefore, the time range of a client service request is more concerned with the containers it passes through.

For above reasons, it is better to classify tracing chains by containers they pass through. The *Container_Dict* of a tracing chain is an important metric to detect the anomaly. In T-Rank, the tracing chains are divided into groups, $G = \{G_1, G_2, ..G_n\}$. The tracing chains which have the same *Container_Dict* are divided into one group G_k, just like the right part of the Fig. 2.

D. Anomaly Detector

In order to calculate different containers' suspicious scores, the SBFL algorithm analyzes the proportion of them in all failed client service requests. A balanced and well-design test-suit makes a great difference in SBFL's effectiveness [21]. Test-suite consists of two parts: the design of test cases (which containers are covered) and their outcomes (which test case fails). In a microservice-based system, it is impossible to design the container coverage of test case which is decided by detailed request and system management. Therefore the only way to enhance the performance of the SBFL algorithm is to provide more precise outcomes.

However, the label *isSuccess* in the tracing data is not reliable and ineffective because it is always *True*, although there is a fault happening. Sometimes some faults in container like network latency will not change the *isSuccess* from *True* to *False* for the reason that the request is finished at last but spends too much time. Therefore besides the *isSuccess* label, it is necessary to detect the anomaly in another way. We compare an attribute of tracing chain, *ElapsedTime*, in the fault injection period with the one in the normal period. From Fig. 4 we find that the *ElapsedTime* obviously increases in some tracing chains when a fault is injected. Considering that the most direct phenomenon when most the faults happen is timeout, we think the tracing chain with an outlier in *ElapsedTime* should be labeled as abnormal.

To find the outlier, T-Rank utilizes a simple and rough anomaly detection based on the *ElapsedTime*. We assume that the *ElapsedTime* of different groups conform to the normal distribution. In the normal distribution, the possibility that $abs(t_i - t_{mean}) > 3 * t_{std}$ is 0.0026. Hence the *ElapsedTime* out of this range can be considered as an anomaly. To confirm the assumption is valid, we label the tracing chains which are out of range by this method. In Fig. 3, $\bigtriangledown$ represents the labeled outlier and almost all outliers are labeled precisely.

For anomaly detection, T-Rank uses the Equ. 1 and Equ. 2 to label the tracing chain. G_k is a group of the tracing chains with same *Container_Dict*. t_i is the tracing chain T_i's *ElapsedTime*. t^k_{mean} is the mean of G_k's *ElapsedTime* and t_{std} is its standard deviation. They are calculated from the historical tracing data. Although the *success* label is ineffective when working alone, it is still worthy of reference. Therefore we get $G_{Abnormal}$, the group of abnormal tracing chains and G_{Normal}, the group of normal tracing chains by the Equ. 3 and Equ. 4.

$$G^k_{Normal} = \{T_i \in G_k | abs(t_i - t^k_{mean}) \leq 3 * t^k_{std}\}, \qquad (1)$$

$$G^k_{Abnormal} = \{T_i \in G_k | abs(t_i - t_{mean}) > 3 * t_{std}\}, \qquad (2)$$

$$G_{Normal} = \sum^{i}_{n} G^i_{Normal} - G_{success=False}, \qquad (3)$$

$$G_{Abnormal} = \sum^{i}_{n} G^i_{Abnormal} + G_{success=False}. \qquad (4)$$

E. Spectrum Ranker

Spectrum Based Fault Localization is an algorithm designed for program debugging. It has achieved great performance in previous studies. SBFL diagnoses the program by providing a list of the suspicious scores. The fault localization component of T-Rank, Spectrum Ranker, is adapted from the SBFL algorithm. Take the Fig. 5 as an example, Req1 and Req2 correspond to tracing chain group G_1 and G_2 respectively. Intuitively, if a great deal of tracing chains in G_1 are abnormal but most ones in G_2 are normal, we will suspect the *docker_01* is a fault container. And if both of G_1 and G_2 have many abnormal tracing chains, there is a high possibility that *docker_04* or *docker_05* has a fault. It is a quite simple thought and Spectrum Ranker is developed from it. Spectrum Ranker formalizes this thought and provides a risk formula to evaluate the container's ratio between the successful requests covering it and the failed ones.

Spectrum Ranker is a simple statistic algorithm collecting the coverage information of containers in client service requests and calculating the containers' suspicious score by the risk evaluation formula $r()$. But to the Spectrum Ranker, the failed case is necessary and it will get nothing if all test cases are successful, therefore the anomaly detection is significant. Spectrum Ranker treats every tracing chain as a test case T and every container c as an element in the test case. The definition of notations about c in Spectrum Ranker is shown as follows:

1) e_f is the number of the failed tracing chain T_f which include the container c, $c \in T_f$.

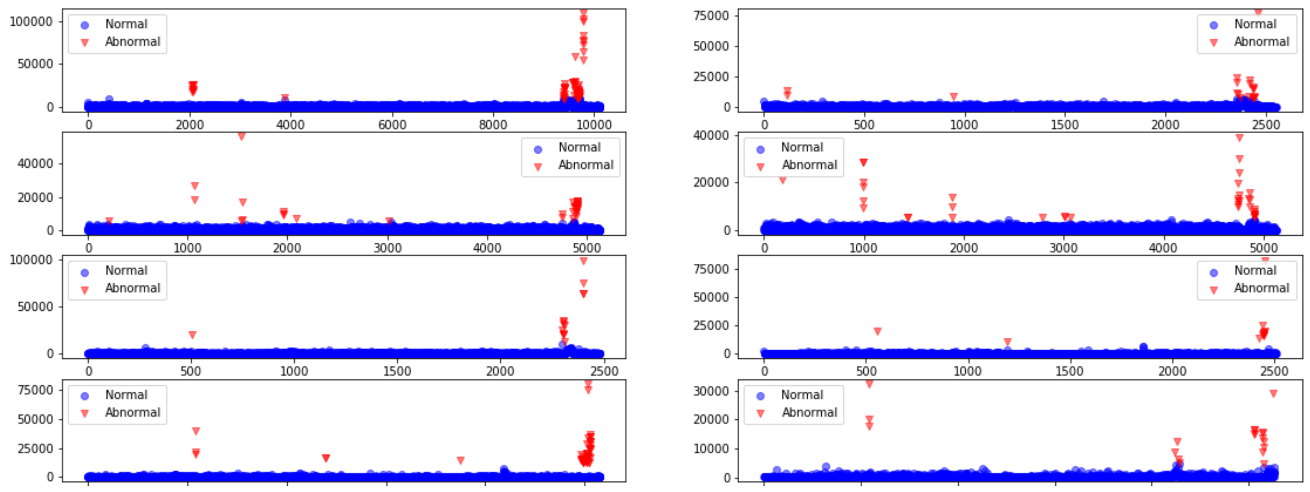

Fig. 3. Tracing chains' *ElapsedTime* in different groups. ▽ is the symbol of an outlier, meaning their *ElapsedTime* is out of normal range.

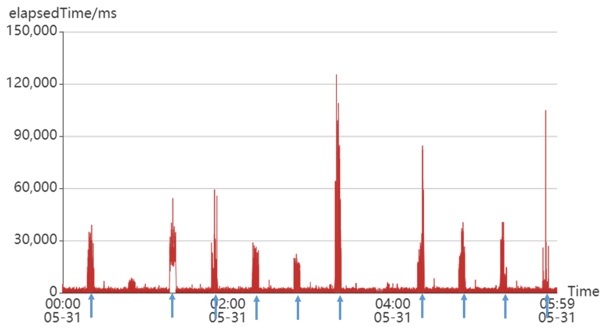

Fig. 4. The *ElapsedTime* of a single trace. The ↑ represents fault injection at this time. When a fault is injected,for example at 2:00, the *ElapsedTime* of some traces is increased significantly.

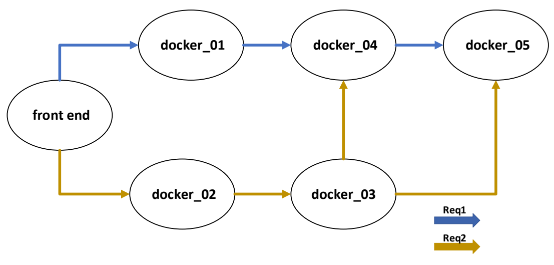

Fig. 5. An example of services dependency graph

2) e_p is the number of the successful tracing chain T_s which include the container c, $c \in T_s$.
3) n_f is the number of the failed tracing chain T_f which don't include the container c, $c \notin T_f$.
4) n_p is the number of the successful tracing chain T_s which don't include the container c, $c \notin T_s$.

According to the definition of notations, we can describe the process of Spectrum Ranker as follows.

The process of Spectrum Ranker is quite simple and the key of Spectrum Ranker is the risk evaluation formula, which

Algorithm 1: SBFL

```
Input: $G$: the groups set of tracing chains
       $r$:the risk evaluation
Output: $L$:the ordered suspicious score list
1 Extract the container set $C$ from $G$
2 Initial suspicious score list $L$
3 for $c$ in $C$ do
4     statistic $e_f, e_p, n_f, n_p$ of $c$ from $G$
5     $L[c] = r(e_f, e_p, n_f, n_p)$
6 end
7 Sort $L$ by suspicious score
```

decides its effectiveness. Review the research in the SBFL algorithm, we can find that there are plenty of formulas [22]. For instance, a popular formula in the SBFL algorithm is Ochiai [23], which is defined as follows.

$$Ochiai(c) = \frac{e_f}{\sqrt{(e_f + n_f) * (e_f + e_p)}}. \tag{5}$$

Let's continue using the Fig.5 as an example. We inject a fault into *docker_01*. If the statistic of the collected tracing data is shown in Table. II. After calculating the suspicious score of containers with Ochiai, the container with the highest suspicious score is *docker_01*, which is the root cause of the fault.

TABLE II
STATISTIC OF COLLECTED TRACING DATA

Container	e_f	e_p	n_f	n_p	Suspicious score
docker_01	100	100	10	190	0.67
docker_02	10	190	100	100	0.06
docker_03	10	190	100	100	0.06
docker_04	110	290	0	0	0.52
docker_05	110	290	0	0	0.52

In T-Rank, we experiment with 25 kinds of spectrum formulas to find out the most effective risk evaluation formula as our default formula.

IV. Experiment

A. Experiment Setup

1) Dataset and Benchmark: A tracing data is provided by a Chinese company, named China Mobile. The tracing data collected from their real-world production microservice systems [5]. From the tracing data, we observe the composition of the microservices system is shown in Table III. All components in the table run as containers and are managed by Kubernetes. The components communicate with each other using gRPC. There are 9 million pieces of tracing items collected in the dataset. The size of the dataset is 1.12 GB. In order to simulate the anomalies, multiple types of faults are injected into the system, such as network delay, network loss, CPU fault, and DB connection limited. Only one fault is injected one time and every fault lasts for 5 minutes.We implement T-Rank in Python3.7 and run it in a computer with 8-core 3.60GHz CPU and 16GB memory.

TABLE III
The Composition of Experiment System

Category	OSB	oracle	redis	docker	virtual machine
Num	1	13	12	8	22

2) Evaluation Metric: To evaluate the effectiveness of the model, we take **Exam Score** (ES) and **Recall& Precision** as the evaluation metric.

Exam Score [21] [24] is the percentage of suspicious containers that need to be inspected by an operator among all candidates before localizing the problem container, reflecting the effectiveness of a fault localization approach. What's more, because most of the time the fault happens alone, ES can better reflect how helpful the tool is when it works online. The lower ES, the better performance the model makes, and the faster the fault is fixed.

Recall & Precision are the metric that evaluates the performance of the model in another way. Precision refers to the percentage of the True fault containers in the fault containers we find. The higher Precision is better, meaning our method can produce less false alarm. Recall refers to the percentage of true fault containers we find in all fault containers. The Recall is higher, meaning our tool can find more fault containers.

B. Different risk evaluation formulas effectiveness

The SBFL algorithm has developed for a long time and previous studies have proposed a lot of formulas. Although their papers have proved all of those formulas perform well in Program Debugging, there may be some difference when SBFL is applied in Microservice root cause analysis. For examining the formula performance, we use 25 risk evaluation formulas to calculate the suspicious score of containers and compare their effectiveness.

[5]dataset, http://iops.ai/competition_detail/?competition_id=15&flag=1

TABLE IV
Different Risk Evaluation Formulas [22]

Ranking metric	**Definition**	**Ranking metric**	**Definition**
Tarantula	$\frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f}+\frac{e_p}{e_p+n_p}}$	Ochiai	$\frac{e_f}{\sqrt{(e_f+e_p)(e_f+n_f)}}$
Jaccard	$\frac{e_f}{e_f+e_p+n_f}$	Ample	$\lvert\frac{e_f}{e_f+n_f} - \frac{e_p}{e_p+n_p}\rvert$
RusselRao	$\frac{e_f}{e_f+e_p+n_f+n_p}$	Hamann	$\frac{e_f+n_p-e_p-n_f}{e_f+e_p+n_f+n_p}$
SϕremsemDice	$\frac{2e_f}{2e_f+e_p+n_f}$	Dice	$\frac{2e_f}{e_f+e_p+n_f}$
Kulczynski1	$\frac{e_f}{n_f+e_p}$	Kulczynski2	$\frac{1}{2}(\frac{e_f}{e_f+n_f} + \frac{e_f}{e_f+e_p})$
SimpleMatching	$\frac{e_f+n_p}{e_f+e_p+n_f+n_p}$	Sokal	$\frac{2e_f+2n_p}{2e_f+2n_p+n_f+e_p}$
M1	$\frac{e_f+n_p}{n_f+e_p}$	M2	$\frac{e_f}{e_f+n_p+2n_f+2e_p}$
RogersTanimoto	$\frac{e_f+n_p}{e_f+n_p+2n_f+2e_p}$	Goodman	$\frac{2e_f-n_f-e_p}{2e_f+n_f+e_p}$
Hamming	$e_f + n_p$	Euclid	$\sqrt{e_f + n_p}$
Overlap	$\frac{e_f}{min(e_f,e_p,n_f)}$	Anderberg	$\frac{e_f}{e_f+2e_p+2n_f}$
Ochiai2	$\frac{e_f n_p}{\sqrt{(e_f+e_p)(n_f+n_p)(e_f+n_p)(e_p+n_f)}}$	Zoltar	$\frac{e_f}{e_f+e_p+n_f+\frac{10000n_f e_p}{e_f}}$
Wong1	e_f	Wong2	$e_f - e_p$
Wong3	$\mathrm{e}_f - h,\ where\, h = \begin{cases} e_P & if\ e_p \leq 2 \\ 2+0.1(e_p-2) & if\ 2 < e_p \leq 10 \\ 2.8+0.01(e_p-10) & if\ e_p > 10 \end{cases}$		

For testing the performance of different SBFL risk evaluation formulas we use different formulas to calculate the suspicious score of containers with fixed sliding time window sizes to compare their online performance. In the online performance experiment, T-Rank only analyzes the tracing data in the last 5 minutes, the sliding time window size. The sliding stride size of the time window is 1 minute. The formula we use in the experiment is cited from the paper [22]. All formulas are show in Table. IV.

Some containers' e_f maybe 0 which leads to the ZeroDivideError. To avoid such a situation, we defined the suspicious score as 0 when $e_f = 0$. Because $e_f = 0$ means that there is no errors in these containers, their suspicious scores should be 0, showing that they work well.

From Fig. 6 we find that when the SBFL works online with fixed window size, different SBFL formulas have a great difference in performance. The group of formulas making the best performance have the lowest ES close to 0, such as Ochiai and Kulcyunski2, meaning that these formulas can localize the fault containers precisely. Based on the suspicious list they provide, the system operator only needs to inspect one or two containers when a fault happens. This helps them save a lot of effort. The formula Wong3 making the worst performance whose ES is up to 0.324, almost 25 times of the best one. Comparing the formulas' structure, we find that those formulas performing badly focus less on the parameter e_f and reduce the importance of it, which may be the reason for their poor performance.

Different formulas have different effectiveness. In this experiment, we find out which formulas are suitable to T-Rank. When faced with different environments and requirements, T-Rank can choose the most suitable risk evaluation formula according to our experiment. This greatly enhanced T-Rank's robustness.

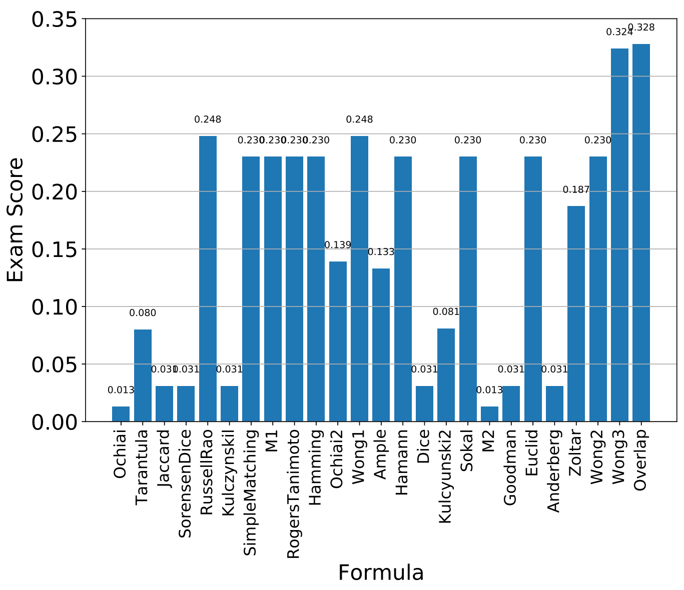

Fig. 6. The performance of different spectrum formulas. Ohiai and M2 achieve best performance

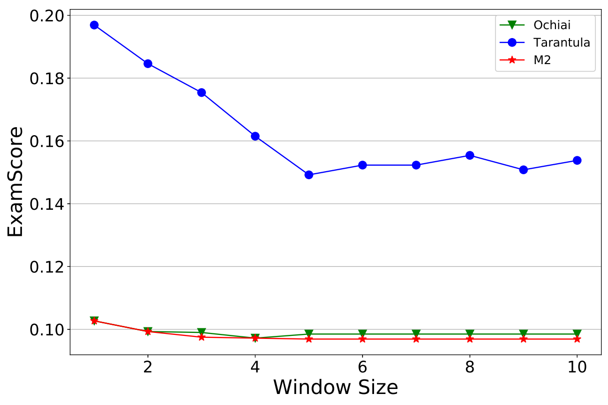

Fig. 7. The ExamScore under different Window Sizes. The improvement of Tarantula's performance is obvious with the increased window size. The improvement of Ochiai's and M2's performance is not so obvious, but they do become better. When the Window Sizes is 4 or 5 minutes, T-Rank performs better.

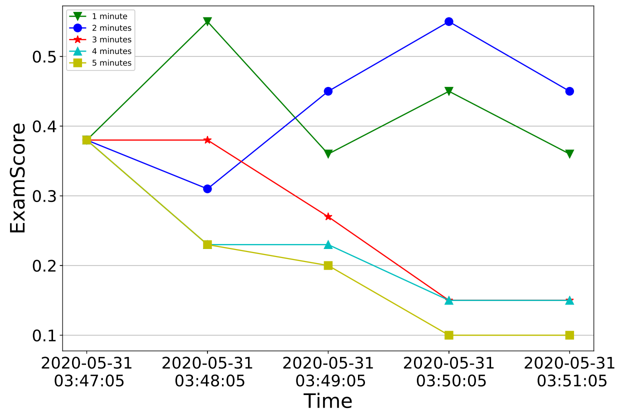

Fig. 8. The ES of different time window sizes. We use *SimpleMatching* risk evaluation formula to experiment and pick up a part of the result of different window sizes. A larger window size gets better results when the fault lasts longer.

C. The Impact of Tracing Data Collection Time Window Size

SBFL is a statistic algorithm. The sliding time window size may affect its performance. When the SBFL model works online, the window size of data collection has a significant impact on its performance for the reason that the window size of data collection makes difference in the amount of the tracing data and failed services, which impact heavily on the final suspicious score list provided by it. With a too-large window size, data collection and processing take too much time, resulting in the model can not catch up with the speed of the tracing data generation. The longer latency of the T-Rank's fault localization weakens its practicality. Eventually, the online SBFL model will degenerate to offline mode, which makes the sliding time window meaningless. On the other hand, too small window size leads to the insufficiency of tracing data and inaccurate localization. The lack of data may cause T-Rank to mistake a normal container as a fault one or even can not find out any fault container.

To balance the data processing speed and data generation speed, we choose minute level granularity to divide data. And the tested window size ranges from 1 to 10 minutes. The SBFL formulas selected in this experiment are Ochiai, Tarantula, and M2 which achieves excellently performance.

From Fig. 7, it is obvious that the ES is reduced with the increase of the window size and stabilizes at low levels at last. When the window size reaches 5 minutes, the Tarantula's ES reaches its minimum. After that, even if we continue to increase the window size, the ES does not decrease any more after that.

To find out the reason why the best sliding time window size is 5 minutes, we pick up a few fault injection period data to analyze different window size's ES change. The Fig.8 show their changing trends. We can know from Fig.8 that over time, the ES of window sizes 1 and 2 have not changed significantly, but the ES of window size 4 and 5 make a great improvement and become 0 at last. When the fault just happens, the useful tracing data to localize the fault is limited, so no matter what the window size is, their performance is close. But when the fault lasts for a while, the larger window size can get more useful information by including more historical tracing data. On account of it, they can provide a more accurate suspicious score list. If the window size keeps increasing and even larger than the fault lasting time, the data before fault happens will be a noise to disturb T-Rank's judgment, resulting in worse performance. And the fault injection lasting time is 4-5 minutes. This is the reason why it's the best that window size is 5. So it is our conclusion that the best window size of the SBFL model should learn from the historical data in practice.

D. Can T-Rank outperform previous work?

To the effectiveness of demonstrate T-Rank, we compare it with several state-of-the-art root cause localization methods.The methods we select include MS-Rank [25], CloudRanger [26], NetMedic [8], MonitorRank [27] and Roots [28]. MS-Rank and CloudRanger are the self-adaptive methods to localize the root cause by utilizing their random walk algorithm on the impact graph they build. NetMedic is a multi-metric diagnosis approach and it establishes the service dependencies by its correlation approach and locates the root cause in the graph it builds. We use the metric extracted from tracing data as its input. Roots mentions four root cause identification approaches. To compare with it, we implement all its approaches. To compare with MonitorRank, we use its batch-mode engine to build the call graph and its random walk algorithm to localize root causes. We compare these methods with ours using metric Recall and Precision.

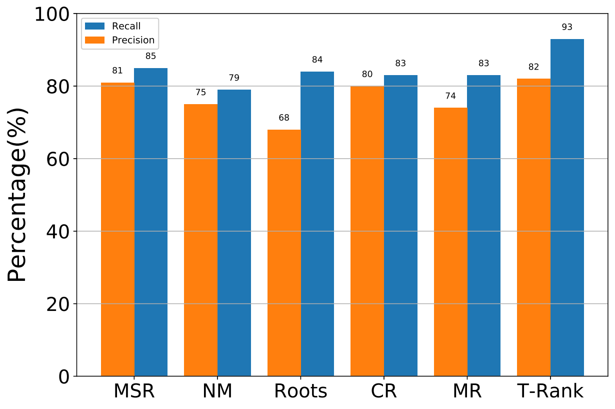

Fig. 9. The Comparison Result with Other Root Cause Localization Methods

The Fig.9 shows the comparison result, where MSR, NM, CR, MR represent MS-Rank, NetMedic, CloudRanger, and MonitorRank respectively. We inject 3 faults into the system and every fault lasts for 5 minutes. The comparison result shows that T-Rank outperforms the previous methods, achieving 1% higher Precision and 8% higher Recall than the best performance method MS-Rank. T-Rank makes a great improvement. T-Rank can achieve a better result because T-Rank only relies on the tracing data and only one metric. The NetMedic and MonitorRank need more data and metrics to more precisely construct the service dependency graph. Lack of data makes them can not find the root cause precisely. Roots prefer to identifies the service close to the front_end service, so it performs badly when a fault is injected in further service. As for CloudRanger and MS-Rank, they can adapt to the metric data, so they can still have a good performance. T-Rank performs better.

But the performance of T-Rank is not as good enough as we expect. After analyzing the detailed experiment result, we think the improvement of the Precision is smaller than the Recall is on account of that there are a few false alarms. T-Rank generates the suspicious score only when the anomaly is detected. And the anomaly detection of T-Rank is too sensitive for its purpose is labeling as much as possible tracing data for localization. But if there is a fault container, T-Rank can localize it precisely. Combined with other anomaly detection tool, T-Rank can achieve better performance. We will make further research in this way.

E. Discussion

In the experiments above, T-Rank has different performance with different risk evaluation formulas and window size. When using the formula like M1 and Ochiai, T-Rank can accurately localize the root cause most time with collection data window size as 5 minutes. From the table V and table VI, we can find that processing the tracing data within 5 minutes only takes 6 seconds at most with one CPU core, which is much less than the window size. And only a little CPU resource is consumed. The advantage of T-Rank is that it is lightweight and needs less data and metric, but has enough high accuracy. However, its limitation does exist. If two or more root causes are existing at the same time, T-Rank maybe only localize one exactly and the container having the second-highest suspicious score may be innocent, just affected by the root cause container in the same tracing chain. And the false alarm should be reduced. But T-Rank as a support tool to help operators exclude the fault is quite effective and practical. Using T-Rank can save a great deal of effort and T-Rank does not occupy too much system resource because of its low cost.

TABLE V
THE COST OF DIFFERENT DATA SIZE

window size/minute	Number of trace	processTime/s
1	41246	1. 163
5	133254	6. 24
10	255602	13. 38
60	1606216	96. 13

TABLE VI
OVERHEAD STATISTICS ON BENCHMARK EXPERIMENT

System Module	Overhead
Tracing collection	2% ± 1% CPU utilization(single core)
Data preprocess and anomaly detection	4% ± 1% CPU utilization(single core)
Root cause localization	8% ± 1% CPU utilization(single core)

V. Related Work

With the popularity of microservice as the enterprise software architecture, the root cause localization attracts so much attention and a lot of researchers focus on addressing it. In this section, we will review some related work in this field.

First, there are many fault localization methods for the microservice system developing from the ones designed for traditional distributed systems. For example, in the traditional distributed network, NetMedic [8] captures all components' state and model the system as a dependency graph. Then it calculates vertexes' score and edges' weight to localize the root cause. MonitorRank [27] stores the collected metric in time partitioned database and generates the call graph of the system periodically. When an anomaly happens, it compares the similarity of sensors' metric pattern to find the relevance of the sensor to the anomaly. Based on this, it provides a random walk algorithm to localize the root cause.

What's more, the statistical analysis of software behavior has been verified as valuable for fault localization in the traditional distributed system. The TABC [9] follows this idea to make further development. TABC detects the anomaly and initializes the anomaly scores of components by comparing the profile to the one learned from historical timing behavior and then utilizes three algorithm variants they proposed to derive anomaly ratings from anomaly scores with the system topology. CauseInfer [29], [30] automatically constructs a two layered hierarchical causality graph based on runtime performance metrics. Then the coarse-grained graph is applied to locate the causes at service level and the fine-grained graph is used to find the real culprits of performance problems by statistical methods. Roots [28] is another statistical methodology for web-application performance diagnosis deployed in Paas clouds. Combining the metadata injection and platform-level instrumentation, Roots tracks the event in the system and analyzes the collected data to localize the root cause of the anomaly.

As for the microservice system, many researches refers to the basic idea of the tradition distributed system fault localization and develops more excellent methods.

Microscope [6] presents a system to identify and locate the abnormal services with a ranked list of the possible root cause. The Microscope system builds the causal graph by collecting data from the socket and provides a parallelized PC-algorithm for service causality graph building. MS-Rank [25] and CloudRanger [26] constructs the causal relationship dynamically and then construct the impact topology of the application system instead of using the given topology. After construction, to identify the culprit services which are responsible for cloud incidents, they put forward a heuristic investigation algorithm based on random walk and generate the ranked list of services.

Microscaler [31], [32] constructs the service dependency graph with the help of Service Mesh, then it locates the performance bottlenecks of microservice systems along the graph by Correlation Coefficient. In the paper Automap [7], they define two notion **"+"** and **"-"** and 3 rules to build the causal graph. By calculating the similarity between the generated causal graph and historical causal graphs, they set the weight of different metrics. And then a heuristic random walk algorithm based on the metric correlation is employed to localize the root cause.

Machine learning techniques also play an important role. The Seer [2] build the neural networks combing the CNN and LSTM. The purpose of CNN is to extract the connection pattern of the system and the LSTM learns the pattern of metric changing over time.

Compared to the above work, T-Rank doesn't rely on the service dependency graph so that we save the time spent in rebuilding the dependency graph, which means faster and lower cost. By analyzing the statistical data of trace to localize the root cause, T-Rank does not need any domain knowledge and spends less time and resources in localization, and can run all the time.

VI. Conclusion

This paper proposes a performance diagnosis system, named T-Rank, for microservice-based system. To address the root cause localization problem, T-Rank collects the tracing data in a sliding time window and integrates them into tracing chains which represent the whole process of a client service request. T-Rank labels the anomaly tracing data by the metric, *ElapsedTime*. The purpose of T-Rank is to offer a ranked suspicious score list of the containers based on the spectrum algorithm. We demonstrate the effectiveness and efficiency of T-Rank with a dataset collected from a real-world production microservice system. T-Rank has high accuracy, low resource cost, and low time consuming, which is quite practical. Moreover, as the scale of a microservice system becomes large, T-Rank can also help operators localize root causes due to its lightweight property.

Acknowledgment

The work was supported by the National Key Research and Development Program of China (2019YFB1804002), the Research Development Plan of Key Areas in Guangdong Province (2020B010165002), the Natural Science Foundation of China (U1811462, 61802448), the Basic and Applied Basic Research of Guangzhou (202002030328), and the Natural Science Foundation of Guangdong Province (2019A1515012229). The corresponding author is Pengfei Chen.

References

[1] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

[2] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. ACM, 2019, pp. 19–33.

[3] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer, "Sieve: actionable insights from monitored metrics in distributed systems," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Las Vegas, NV, USA, December 11 - 15, 2017*. ACM, 2017, pp. 14–27.

[4] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification," *ACM Comput. Surv.*, vol. 48, no. 1, pp. 4:1–4:35, 2015.

[5] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 2019, pp. 683–694.

[6] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *Service-Oriented Computing - 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings*, ser. Lecture Notes in Computer Science, vol. 11236. Springer, 2018, pp. 3–20.

[7] M. Ma, J. Xu, Y. Wang, P. Chen, Z. Zhang, and P. Wang, "Automap: Diagnose your microservice-based web applications automatically," in *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*. ACM / IW3C2, 2020, pp. 246–258.

[8] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl, "Detailed diagnosis in enterprise networks," in *Proceedings of the ACM SIGCOMM 2009 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Barcelona, Spain, August 16-21, 2009*. ACM, 2009, pp. 243–254.

[9] N. Marwede, M. Rohr, A. van Hoorn, and W. Hasselbring, "Automatic failure diagnosis support in distributed large-scale software systems based on timing behavior anomaly correlation," in *13th European Conference on Software Maintenance and Reengineering, CSMR 2009, Architecture-Centric Maintenance of Large-SCale Software Systems, Kaiserslautern, Germany, 24-27 March 2009*. IEEE Computer Society, 2009, pp. 47–58.

[10] A. Parker, D. Spoonhower, J. Mace, B. Sigelman, and R. Isaacs, *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media, 2020.

[11] Traceroute. [Online]. Available: https://wikipedia.org/wiki/Traceroute

[12] Sidecar. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar

[13] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings*. USENIX, 2007.

[14] Zipkin. [Online]. Available: http://zipkin.io/

[15] Y. Shkuro, "Evolving distributed tracing at uber engineering," *Uber Engineering Blog*, 2017.

[16] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*. ACM, 2002, pp. 467–477.

[17] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*. ACM, 2005, pp. 273–282.

[18] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[19] X. Xie, T. Y. Chen, F. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 31:1–31:40, 2013.

[20] T. B. Le, F. Thung, and D. Lo, "Theory and practice, do they match? A case with spectrum-based fault localization," in *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*. IEEE Computer Society, 2013, pp. 380–383.

[21] P. Chatterjee, A. Chatterjee, J. Campos, R. Abreu, and S. Roy, "Diagnosing software faults using multiverse analysis," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*. ijcai.org, 2020, pp. 1629–1635.

[22] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 2014, pp. 191–200.

[23] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006), 18-20 December, 2006, University of California, Riverside, USA*. IEEE Computer Society, 2006, pp. 39–46.

[24] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *CoRR*, vol. abs/1803.09939, 2018.

[25] M. Ma, W. Lin, D. Pan, and P. Wang, "Ms-rank: Multi-metric and self-adaptive root cause diagnosis for microservice applications," in *2019 IEEE International Conference on Web Services, ICWS 2019, Milan, Italy, July 8-13, 2019*. IEEE, 2019, pp. 60–67.

[26] P. Wang, J. Xu, M. Ma, W. Lin, D. Pan, Y. Wang, and P. Chen, "Cloudranger: Root cause identification for cloud native systems," in *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018*. IEEE Computer Society, 2018, pp. 492–502.

[27] M. Kim, R. Sumbaly, and S. Shah, "Root cause detection in a service-oriented architecture," in *ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '13, Pittsburgh, PA, USA, June 17-21, 2013*. ACM, 2013, pp. 93–104.

[28] H. Jayathilaka, C. Krintz, and R. Wolski, "Performance monitoring and root cause analysis for cloud-hosted web applications," in *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*. ACM, 2017, pp. 469–478.

[29] P. Chen, Y. Qi, P. Zheng, and D. Hou, "Causeinfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems," in *2014 IEEE Conference on Computer Communications*. IEEE, 2014, pp. 1887–1895.

[30] P. Chen, Y. Qi, and D. Hou, "Causeinfer: Automated end-to-end performance diagnosis with hierarchical causality graph in cloud environment," *IEEE Transaction Service Computing*, vol. 12, no. 2, pp. 214–230, 2019.

[31] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *2019 IEEE International Conference on Web Services*. IEEE, 2019, pp. 68–75.

[32] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2020.