ModelCoder: A Fault Model based Automatic Root Cause Localization Framework for Microservice Systems

Yang Cai
College of Computer
National University of Defense Technology
Changsha, Hunan, CN
caiyang@nudt.edu.cn

Biao Han*
College of Computer
National University of Defense Technology
Changsha, Hunan, CN
nudtbill@nudt.edu.cn

Jie Li
Department of Computer Science and Engineering
Shanghai Jiaotong University
Shanghai, CN
lijie@cs.sjtu.edu.cn

Na Zhao
College of Computer
National University of Defense Technology
Changsha, Hunan, CN
zhaona@nudt.edu.cn

Jinshu Su
College of Computer
National University of Defense Technology
Changsha, Hunan, CN
sjs@nudt.edu.cn

2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS) | 978-1-6654-1494-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/IWQOS52092.2021.9521318

***Abstract*—Microservice system is an architectural style to develop a single application as a suite of small services running in its process and communicating with lightweight message mechanisms. Although microservice architecture enables rapid, frequent and reliable delivery of large, complex applications, it is increasingly challenging for operational staffs to locate the root cause of a microservice fault, which usually occurs on a service node and propagates to affect the entire system. To this end, in this paper, we first introduce the concept of deployment graph and service dependency graph to depict the deployment status and calling relationship between service nodes. Then we formulate the root cause localization problem in microservice systems based on the constructed graphs, in which fault model is defined to capture the characteristics of a fault's root cause. A fault model based automatic root cause localization framework called ModelCoder is later developed to figure out the root cause of unknown faults by comparing with the predefined fault models. We evaluate ModelCoder on a real-world microservice system monitoring data set spanning 15 days. Through extensive experiments, it is revealed that ModelCoder can localize the fault root cause nodes within 80 seconds on average and improve the root cause localization accuracy (to 93%) by 12% compared with the state-of-the-art root cause localization algorithm.**

***Index Terms*—microservice, fault model, root cause localization**

I. Introduction

Microservice is a widely used architecture dividing a single application into a group of service nodes. Each service node is a separate process, which can be deployed, compiled and run independently. Service nodes coordinate and cooperate with each other through a lightweight message mechanism to provide users with complete service [1].

Microservice can improve the abstraction, modularity and extensibility of web applications [2], but it also brings some new problems, one of which is the root cause localization of faults. Fast and accurate localization of root causes can reduce troubleshooting time and economic loss. Enormous service nodes and monitoring data in microservice systems make it extremely difficult to rapidly and automatically localize the root cause [3]. Existing works applied log analysis tools [4] can only provide very simple assistance and strongly rely on the experience of operational staffs. Therefore, it is an urgent need to design an automatic root cause localization framework.

- First, as a microservice system has hundreds of service nodes [5], the calling relationships among service nodes are complex and dynamic. As a result, existing root cause localization algorithms based on static topology or model are not suitable for microservice systems.
- Second, as a fault often involves multiple service nodes [6], analysis of a single service node can not accurately capture all features of the fault. In addition, the features of faults are diverse, in which the root cause nodes are normal or abnormal in different faults. Therefore, we need to analyze the faults from a multi-nodes scale.

To address these challenges, we first define **deployment graph** and **service dependency graph** to depict the complex relationships among service nodes. The abnormal nodes are classified into **explicit nodes** and **implicit nodes** to further represent the features of faults. According to the above definitions, we formulate the fault root cause localization problem for microservice systems. To solve the problem, we develop a fault model based automatic root cause localization framework called **ModelCoder**, which consists of three modules: data preprocessing, known fault analysis and unknown fault analysis. Based on a novel coding technique, we define **fault model** to represent the feature of faults, which promotes the root cause localization scale from single-node to multiple-nodes.

We evaluate ModelCoder with monitoring data of a real-world microservice system [7]. Compared with the state-of-the-art root cause localization algorithm, ModelCoder improves the localization accuracy by 12%. Extensive experimental results reveal that the root cause localization time of ModelCoder could be greatly reduced without significantly affecting the localization accuracy.

The main contributions of this paper are as follows.

*Corresponding author is Biao Han.

978-0-7381-3207-5/21/$31.00 ©2021 IEEE

978-1-6654-1494-4/21/$31.00 ©2021 IEEE

- As far as we know, we are the first to formulate the fault root cause localization problem in microservice systems, in which our objective is to improve the location accuracy by optimizing the fault feature matching process.
- We propose ModelCoder, a fault model based root cause localization framework, in which the formulated localization problem is solved by automatically matching feature vectors between known and unknown faults.
- We evaluate the root cause localization accuracy and speed of ModelCoder on a microservice system monitoring data. Experimental results show that Modelcoder can achieve about 93% localization accuracy within 80 seconds on average, which outperforms the current root cause localization algorithms in microservice systems.

The rest of this paper is organized as follows. Section II briefly surveys the related work. Section III describes the system model and formulates the root cause localization problem. Section IV provides the architecture of our proposed ModelCoder. Section V introduces the implementation details of ModelCoder. Section VI discusses the evaluation results of ModelCoder. Section VII concludes the paper and points out the future work.

II. Related work

This section reviews the work related to root cause localization and divides them into the following three categories according to their characteristics.

Trace-based methods. Trace-based methods are often used to analyze the monomer program from the event or function level. By tracing the relationship of events or functions, they can construct the system's overview model which aids the root cause analysis. X-Trace [8] is a tracing framework that provides a comprehensive view of the systems combining different applications, administrative domains, network mechanisms. DARC [9] can construct a running program's call-graph using runtime latency analysis and finds root cause paths according to it. Insight [10] can create the execution path of the failed service request within minutes after the fault is detected to help localize the root cause of the fault. In general, these algorithms make a detailed analysis of the system; they bring considerable overhead and require operational staffs to understand the system very well to deploy them.

Static methods. Static methods analyze the system based on static information such as the system model, threshold and graphs. ϵ-Diagnosis [11] determines whether the system is abnormal according to the manually set threshold and starts the fault analysis process. Gao *et al*. [12] propose a transition probability model based on Markov characteristics to describe the correlation between pairs of measurements, thus representing the system's profile. Vscope [13] continuously detects and tracks interaction in the whole process of the application running based on advance generated graphs and advance defined functions. Depending on the information that keeps unchanged for a certain period, these algorithms cannot adapt to the dynamic system.

Dynamic methods. The dynamic methods localize the root cause based on the real-time updating system model. Automap [14] analyzes the fault propagation based on abnormal behavior graphs that are dynamically generated and updated. Furthermore, Causeinfer [15] can balance the granularity and performance of root cause localization by constructing a double-layer causality diagram. However, the root cause localization of these methods is at the single-node level as they only consider the abnormal node itself. In contrast, Weng *et al.* [16] considers multiple nodes on the same physical host for the first time by adding the resource contention relationship between nodes to the system model. Based on the ripple effect, Li *et al.* [17] cluster the abnormal nodes generated by the same root cause to deduces the fault root cause node. These two methods involve the representation of multi-node features, but only for a specific case and lack of scalability.

ModelCoder is different from the above methods. First, it analyzes the calling data between service nodes instead of the events or functions inside service nodes, thus reducing the root cause localization overhead to the system. Second, it constructs the service dependency graph in real-time so that it can respond to the topology changes of the system in time. Third, it analyzes the fault from a multi-node level based on the fault model which can accurately extract the feature of a variety of faults.

III. System model and problem description

A. System Model

In order to represent the resource contention and calling relationships between the service nodes involved in a user's request, the deployment graph and the service dependency graph are defined respectively.

The deployment graph $G_D = \langle V, E \rangle$ is defined to represent the deploy status of service nodes on the physical hosts. V includes service nodes and physical hosts and E is a set of directed edges $\langle v_i, v_j \rangle$ denoting that service node v_i is deployed on physical host v_j where $v_i, v_j \in V$.

The service dependency graph $G_S = \langle N, L \rangle$ is defined to represent the service nodes involved during a user's request and their calling relationship. N is a set of service nodes and L is a set of directed edges $\langle v_i, v_j \rangle$ denoting that service node v_i call service node v_j where $v_i, v_j \in N$. A typical service dependency graph is shown in Fig. 1, in which the circles represent the service nodes, *docker_xxx* are the docker service nodes, *os_xxx* is the host service node, *db_xxx* is the database service node, arrows represent the calls, an arrow starts at the service node initiating the call and ends at the service node responding to the call. The meaning of different kinds of circles and arrows will be explained in §III-B.

We define three categories of node to lay a foundation for the subsequent concepts. In a deployment graph G_D, if there exist edges $\langle v_i, v_k \rangle$ and $\langle v_j, v_k \rangle$, then service nodes v_i and v_j are **bidirectional nodes** of each other as they are deployed on the same physical host v_k. In a service dependency graph G_S, if there exists edge $\langle v_m, v_n \rangle$ and $v_m \neq v_n$, then v_m is the **parent node** of v_n and v_n is the **child node** of v_m.

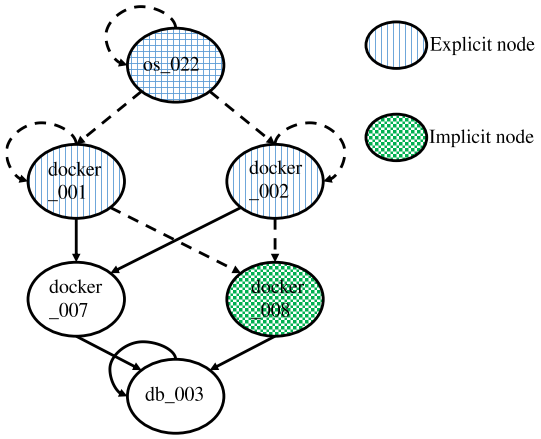

Fig. 1. Service dependency graph with abnormal call and service nodes

B. Abnormal Detection and Representation

From the user's point of view, we identify whether a specific call is abnormal by its elapsedtime which starts with the initialization of this call and ends with getting the return data.

1) Abnormal call: Based on the historical data, the call whose elapsedtime is greater than the normal value is identified as an abnormal call, as shown by the dashed arrows in Fig. 1. The detail of calculating will be described in §IV-C.

2) Abnormal service node: The service nodes in the abnormal calling chains are defined as the abnormal service nodes and are further divided into **explicit node** and **implicit node** as shown in Fig. 1. In a service dependency graph G_S, if there exists abnormal call $\langle v_i, v_j \rangle$, then v_i, v_j are all abnormal service nodes and v_i is an explicit node; furthermore, if there does not exist abnormal call $\langle v_j, v_m \rangle$, then v_j is an implicit node. Normal nodes, explicit nodes and implicit nodes sets are represented by V^{nor}, V^{ex} and V^{im} respectively.

C. Problem Description

According to the above definitions and discussions, we theoretically formulate the root cause localization problem in microservice systems.

When a fault occurs in microservice systems, we can get the information as follows: deployment graph G_D, a set of service dependency graphs G_S during the fault time, node sets of different categories: V^{nor}, V^{ex}, V^{im}, and a set of known faults $F_K = \{f_1, f_2, ..., f_{N_K}\}$. For the known fault $f_j (f_j \in F_K)$, we express its feature as a vector $f_j = \langle x_1, x_1, ..., x_m \rangle$ where m is the dimension of the vector and $x_i (1 \leq i \leq m)$ is in $\{-1, 0, 1\}$.

We need to localize the root cause node of a set of unknown faults $F_U = \{f_1, f_2, ..., f_{N_U}\}$. For the unknown fault $f_i (f_i \in F_U)$, there are a group of candidate root cause nodes $C^i = \{v_1^i, v_2^i, ..., v_{N_i}^i\}$. For the candidate fault root cause node v_k^i $(v_k^i \in C^i)$, we express its feature as a vector $v_k^i = \langle x_1, x_2, ..., x_m \rangle$ just like that of known fault.

***Definition** 3.1 (Distance between the feature vector).* Distance between the feature vectors of candidate fault root cause node v_k^i and the known fault f_j is defined as:

$$d\left(v_k^i, f_j\right) = \sum_{1 \leq n \leq m} \alpha_n compare\left(v_{kn}^i, f_{jn}\right), \tag{1}$$

$$compare(a, b) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{if } a \neq b \end{cases}, \tag{2}$$

where α_n is the weight of the comparison result of each bit of the vector.

when analyzing the unknown fault f_i, we divide its candidate fault root cause nodes C^i into different clusters whose centers are known faults F_K. The cluster center f_k which satisfies:

$$f_k = argmin\left\{d\left(v_k^i, f_j\right) | f_j \epsilon F_K\right\}, \tag{3}$$

is the center of the cluster to which v_k^i belongs.

***Definition** 3.2 (Candidate fault root cause node score)* The distance between v_k^i and f_k is defined as the score of v_k^i.

By ranking candidate fault root cause nodes in C^i according to their score from small to large, we can obtain the analysis result A_{f_i} of the unknown fault f_i. Assuming that the real fault root cause node of f_i is r_{f_i}, $P\left(r_{f_i}, A_{f_i}\right)$ is defined as the location of r_{f_i} in A_{f_i}.

Our goals is:

$$Minimize \sum_{f_i \in F_U} P(r_{f_i}, A_{f_i}). \tag{4}$$

IV. ModelCoder architecture and implement

A. Key Ideas

To solve the problem mentioned in §III, we propose the **node feature group** which includes the target node itself, its parent nodes, child nodes and bidirectional nodes. The distribution of explicit nodes and implicit nodes in the node feature group is defined as **node feature**. The node feature of the fault root cause node is defined as the **fault model** which is different for different faults.

We conclude four fault models from the data set [7]: *docker network fault*, *CPU fault*, *docker deployed host network fault* and *user oriented host network fault*. Due to the limited space, the characteristics of each fault mode are not described in detail.

Node feature code is proposed to formulate the node feature which includes the information of four categories nodes in the node feature group. The details of coding will be explained in §IV-D.

B. System Architecture

Based on the above ideas, we design and implement the ModelCoder which mainly consists of three modules: data preprocessing, known fault analysis and unknown fault analysis.

1) Data preprocessing: Abnormal data is kept and constructed as a set of dependency graphs according to which abnormal service nodes are identified and are divided into explicit nodes and implicit nodes.

2) Known faults analysis: The node features of known faults root cause nodes are coded and stored in the standardized coding library.

3) Unknown fault analysis: Candidate root cause service nodes are determined according to the explicit node and implicit nodes. Their node features are coded and obtained codes are compared with those in the standardized coding library to get the root cause service node.

C. Abnormal Detection

1) Abnormal call judgment: When judging whether a call is abnormal, we select all calls with the same *cmdb_id* and *callType* within a period and calculate the average μ and standard deviation σ of the elapsed time of these calls and set $\mu + 3\sigma$ as the threshold. The call whose elapsedtime is greater than the threshold will be judged as an abnormal call.

2) Explicit and implicit nodes determination: The explicit and implicit nodes are identified based on the set of service dependency graphs. The explicit nodes are the initiating nodes of abnormal calls and the implicit nodes are the response nodes of abnormal calls.

D. Fault Model Encoding

1) Encoding format: The node feature is recorded with an 8 bits code where the first bit reflects the category of the node itself: if the node is an explicit node, the first bit is 1; otherwise, the first bit is 0. The second bit reflects the distribution of explicit nodes in the node's child nodes, 1, 0 and - 1 respectively correspond to three cases: all are explicit nodes, some are explicit nodes and none is explicit nodes. The third bit corresponds to the distribution of implicit nodes in the node's child nodes. The fourth and fifth bits reflect the category of the node's parent node, and the sixth and seventh bits reflect the category of the node's bidirectional nodes.

2) Standard code storage: When dealing with a known fault, the id of fault root cause node, fault category, and the code of fault model is constructed as a triple $\langle cmdb\_id, fault\_type, code\rangle$ and stored in the standard coding library.

E. PSO Optimized Fault Root Cause Node Localization

This section mainly describes how to determine the fault root cause node and fault category based on the coding analysis optimized by the particle swarm optimization algorithm.

1) Candidate fault root cause node searching: Since the fault generally propagates upward from the bottom layer of the service dependency graph, we take the bottom explicit nodes and implicit nodes as the candidate fault root cause nodes.

2) Code comparing: We measure the similarity between two nodes' node feature codes by calculating their similarity score as shown in (5) and (6). As the code contains the category information of four kinds of nodes: the target node itself, its child nodes, its parent nodes, and its bidirectional nodes, we set four coefficients: $\alpha_1, \alpha_2, \alpha_3, \alpha_4$.

$$S(code_1, code_2) = \sum_{1\le i\le 4, 1\le j\le 7} \alpha_i compare(code_{1j}, code_{2j}) \quad \begin{cases} i = 1 \text{ if } j = 1 \\ i = 2 \text{ if } j = 2 \textit{ or } 3 \\ i = 3 \text{ if } j = 4 \textit{ or } 5 \\ i = 4 \text{ if } j = 6 \textit{ or } 7 \end{cases}, \tag{5}$$

$$compare(a, b) = \begin{cases} 1 \text{ if } a = b \\ 0 \text{ if } a \neq b \end{cases}. \tag{6}$$

3) Coefficient optimization: As the space of coefficient value is large and we need to search their appropriate values, in a certain time, we choose to use heuristic algorithm and Particle swarm optimization algorithm [18] is applied finally. The values of four coefficients are taken as the coordinates of the particles, and the root cause localization accuracy of ModelCoder is taken as the return value of the fitness function. We set up a four-dimensional space and make the particles move in it. The specific experimental setup and results are shown in §V-B.

4) Coding analysis: For each candidate fault root cause node, its node feature code is compared with that in the standard coding library to calculate the similarity scores in which the maximum one is taken as the score of this candidate fault root cause node as shown in (7).

$$grade_{node_i} = max\left\{S(code_{node_i}, code) | code \subseteq codes\right\}, \tag{7}$$

where $node_i$ is a candidate fault root cause node, $code_{node_i}$ is the code of $node_i$, $codes$ is the set of codes in the standard coding library, and $grade_{node_i}$ is the score of $node_i$. By ranking candidate fault root cause nodes according to their score from large to small, we can obtain the analysis result of the unknown fault.

For the candidate fault root cause node $node_i$, the triple $\langle cmdb\_id', fault\_type', code'\rangle$ is selected out from the standard coding library in which:

$$code' = argmax\left\{S(code_{node_i}, code) | code \subseteq codes\right\}, \tag{8}$$

where $fault\_type'$ is the the fault category corresponding to $node_i$.

V. Performance Evaluations

A. Dataset and Benchmarks

1) Dataset: The data set is from the 2020 International AIOps Challenge and is collected from the monitoring data

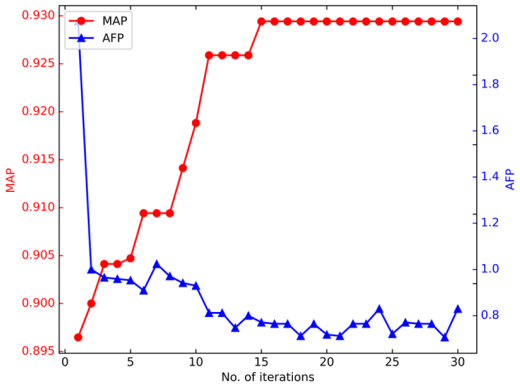

Fig. 2. Particle swarm optimization algorithm optimization results.

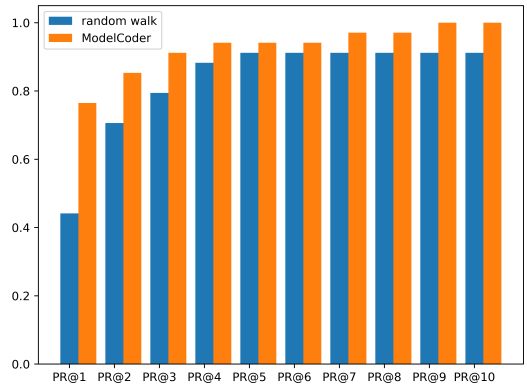

Fig. 3. Scores of PR@K

set of microservice systems in a cloud environment built by Zhejiang Mobile operators.

The abnormal calling data of 81 faults is injected into monitoring data. Each fault lasts for 5 minutes and only contains a fault root cause node. Since the ModelCoder can only deal with the faults that affect the system's response time to the user's request, we choose 63 faults that affect the response time from all faults and the other 18 faults are discarded. In order to ensure that each type of fault contains a sufficient number of faults, so as to generate the standard code and test the accuracy of the algorithm. Among those 63 faults, 29 faults are used in known fault analysis, and 34 faults are used in unknown fault analysis to test the root cause localization accuracy and speed of the ModelCoder.

2) Baseline approach: To illustrate the ModelCoder's root cause localization accuracy, **random walk** is chosen as the baseline approach, which is widely used as a fault cause localization method [14], [16]. When comparing, we first build the service dependency graphs and determine the abnormal nodes. On the basis of that, we run ModelCoder and random walk respectively to compare their performance.

3) Evaluation criteria: We use the $PR@K$, MAP, AFP proposed by [16] and [19] to quantify the performance of each method.

B. Experimental Results

1) Effect of Particle Swarm Optimization Algorithm: We set the parameters of the particle swarm optimization algorithm as follow: the number of particles is 10, the number of iterations is 30, the particle moving range is $[-10, 10]$ and the maximum particle velocity is 1. After each round of iterations, the average fitness function return values of all particles is calculated as the score of ModelCoder in this round. The results are shown in Fig. 2 where the abscissa is the number of iterations and the ordinate is the value of MAP and AFP.

As shown in Fig. 2, the accuracy of ModelCoder gradually increases and reaches the optimal value after 15 iterations. The MAP value increases from 0.896 to 0.929, and the AFP value decreases from 3.835 to about 0.8. Since we take MAP value as the particles' fitness function value, after 15 iterations, particles continue to move in the region with the optimal MAP value causing the AFP value still fluctuates slightly.

2) Root Cause Localization Accuracy: ModelCoder is compared with the random walk to evaluate its root cause localization accuracy. According to the result of the particle swarm optimization algorithm in the previous section, the coefficients $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ are set as [2, 7, 3, 3].

The scores from PR@1 to PR@10 of ModelCoder and random walk are shown in Fig. 3 where the abscissa represents different values of K, and the ordinate represents the corresponding value of PR@K. As shown in Tab. I, when considering all faults, the MAP of ModelCoder is 12% higher than that of random walk, its AFP is 58% lower than that of the random walk.

3) Effect of Service Dependency Graph Density: If we build all service dependency graphs during the fault time, there will be many duplicate service dependency graphs; thus, we choose one from m service dependency graphs and change m, then record the change of ModelCoder's accuracy and speed. We repeat the experiment 10 times for each m value and record the upper bound of 95% confidence interval of the used time. The result is shown in Fig. 4.

It can be seen from Fig. 4 that when m value increasing to 5, the used time is greatly reduced from 85.05s to 51.75s, while the algorithm accuracy can still maintain at a high level, MAP value decreases from 0.93 to 0.86, and AFP value increases from 0.76 to 1.78. When the m value is greater than 5, the time reduction is not obvious, but the accuracy of the algorithm is greatly reduced.

VI. Conclusions

In this paper, we propose ModelCoder, a fault model based automatic root cause localization framework which can adapt to the dynamic microservice system. Compared with the random walk, the accuracy of ModelCoder is improved by 12%, and by optimizing the parameters, the root cause localizing time can be greatly reduced without affecting the accuracy significantly. In the future, we will design a new algorithm to refine the root cause localizing from node level to indicator level.

TABLE I
COMPARISON OF MODELCODER AND RANDOM WALK'S SCORES

	Fault number	Benchmarks	ModelCoder	Random walk
All fault	34	MAP	0.93	0.83
		AFP	0.71	1.71
Docker network fault	15	MAP	0.84	0.77
		AFP	1.60	2.27
CPU fault	7	MAP	1.00	1.00
		AFP	0.00	0.00
User oriented host network fault	4	MAP	1.00	0.93
		AFP	0.00	0.75
Docker deployed host network fault	8	MAP	1.00	0.74
		AFP	0.00	2.63

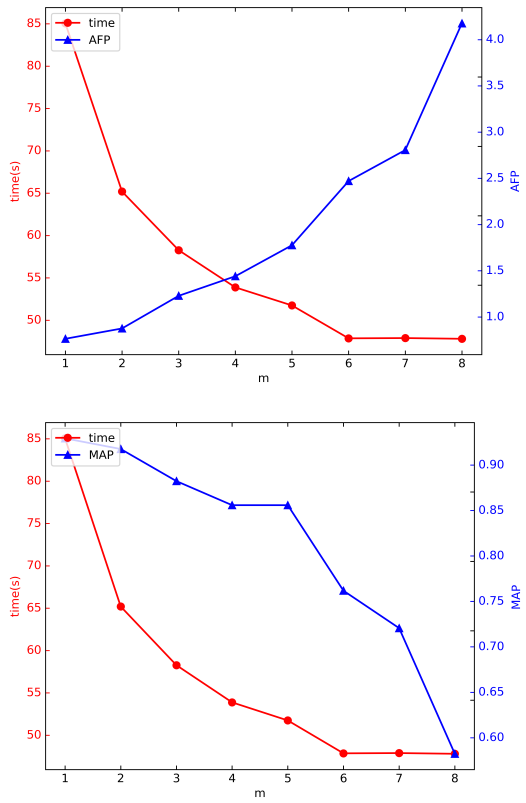

Fig. 4. Effect of service dependency graph density

ACKNOWLEDGMENT

This work was supported in part by the National Natural Science Foundation of China (No.61601483), Hunan Young Talents Grant (No. 2020RC3027) and the Training Program for Excellent Young Innovators of Changsha (No. kq2009027).

REFERENCES

[1] "grpc," [Online], https://www.grpc.io/ Accessed 2020.
[2] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.
[3] R. Li and H. Asaeda, "A blockchain-based data life cycle protection framework for information-centric networks," *IEEE Communications Magazine*, vol. 57, no. 6, pp. 20–25, 2019.
[4] Elasticsearch.Com, "Elasticsearch," [Online], https://www.elastic.co/cn/elasticsearch/ Accessed 2020.
[5] Netflix.Com, "Netflix," [Online], https://www.netflix.com/ Accessed 2018.
[6] P. Liu, Y. Chen, X. Nie, J. Zhu, S. Zhang, K. Sui, M. Zhang, and D. Pei, "Fluxrank: A widely-deployable framework to automatically localizing root cause machines for software service failure mitigation," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 35–46.
[7] Z. Li, "Aiops-challenge-2020-data," [Online], https://github.com/NetManAIOps/AIOps-Challenge-2020-Data Accessed 2020.
[8] R. Fonseca, G. Porter, R. H. Katz, and S. Shenker, "X-trace: A pervasive network tracing framework," in *4th {USENIX} Symposium on Networked Systems Design & Implementation ({NSDI} 07)*, 2007.
[9] A. Traeger, I. Deras, and E. Zadok, "Darc: Dynamic analysis of root causes of latency distributions," in *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2008, pp. 277–288.
[10] H. Nguyen, D. J. Dean, K. Kc, and X. Gu, "Insight: In-situ online service failure path inference in production computing infrastructures," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 269–280.
[11] H. Shan, Y. Chen, H. Liu, Y. Zhang, X. Xiao, X. He, M. Li, and W. Ding, "ϵ-diagnosis: Unsupervised and real-time diagnosis of small-window long-tail latency in large-scale microservice platforms," in *The World Wide Web Conference*, 2019, pp. 3215–3222.
[12] J. Gao, G. Jiang, H. Chen, and J. Han, "Modeling probabilistic measurement correlations for problem determination in large-scale distributed systems," in *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, 2009, pp. 623–630.
[13] C. Wang, I. A. Rayan, G. Eisenhauer, K. Schwan, V. Talwar, M. Wolf, and C. Huneycutt, "Vscope: middleware for troubleshooting time-sensitive data center applications," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2012, pp. 121–141.
[14] M. Ma, J. Xu, Y. Wang, P. Chen, Z. Zhang, and P. Wang, "Automap: Diagnose your microservice-based web applications automatically," in *Proceedings of The Web Conference 2020*, 2020, pp. 246–258.
[15] P. Chen, Y. Qi, and D. Hou, "Causeinfer: automated end-to-end performance diagnosis with hierarchical causality graph in cloud environment," *IEEE transactions on services computing*, vol. 12, no. 2, pp. 214–230, 2016.
[16] J. Weng, J. H. Wang, J. Yang, and Y. Yang, "Root cause analysis of anomalies of multitier services in public clouds," *IEEE/ACM Transactions on Networking*, vol. 26, no. 4, pp. 1646–1659, 2018.
[17] Z. Li, C. Luo, Y. Zhao, Y. Sun, K. Sui, X. Wang, D. Liu, X. Jin, Q. Wang, and D. Pei, "Generic and robust localization of multi-dimensional root causes," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 47–57.
[18] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95-International Conference on Neural Networks*, vol. 4. IEEE, 1995, pp. 1942–1948.
[19] M. Kim, R. Sumbaly, and S. Shah, "Root cause detection in a service-oriented architecture," *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 1, pp. 93–104, 2013.