Delta Debugging Microservice Systems

Xiang Zhou*
Fudan University
China

Xin Peng*†
Fudan University
China

Tao Xie
University of Illinois at Urbana-Champaign
USA

Jun Sun
Singapore University of Technology and Design
Singapore

Wenhai Li*
Fudan University
China

Chao Ji*
Fudan University
China

Dan Ding*
Fudan University
China

ABSTRACT

Debugging microservice systems involves the deployment and manipulation of microservice systems on a containerized environment and faces unique challenges due to the high complexity and dynamism of microservices. To address these challenges, in this paper, we propose a debugging approach for microservice systems based on the delta debugging algorithm, which is to minimize failure-inducing deltas of circumstances (e.g., deployment, environmental configurations) for effective debugging. Our approach includes novel techniques for defining, deploying/manipulating, and executing deltas following the idea of delta debugging. In particular, to construct a (failing) circumstance space for delta debugging to minimize, our approach defines a set of dimensions that can affect the execution of microservice systems. Our experimental study on a medium-size microservice benchmark system shows that our approach can effectively identify failure-inducing deltas that help diagnose the root causes.

CCS CONCEPTS

• **Software and its engineering → Cloud computing**; **Software testing and debugging**;

KEYWORDS

Microservice, Delta Debugging, Testing

*X. Zhou, X. Peng, W. Li, C. Ji, and D. Ding are with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China and the Shanghai Institute of Intelligent Electronics & Systems, China.
†X. Peng is the corresponding author.

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3240730

ACM Reference Format:
Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Wenhai Li, Chao Ji, and Dan Ding. 2018. Delta Debugging Microservice Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3238147.3240730

1 INTRODUCTION

Beyond the implementations of individual microservices, many failures of microservice systems are due to their runtime environments (e.g., containers), communications, or coordinations [19]. Therefore, debugging a failure in microservice systems faces unique challenges due to the high complexity and dynamism of microservices in four dimensions: node, instance, configuration, and sequence. First, numerous microservice instances run on a large number of *nodes* (e.g., physical or virtual machines) and the distribution of microservice instances over nodes is constantly changing, bringing great uncertainties to microservice communication. Second, the *instances* of a microservice may be in inconsistent states and thus behave differently. Third, microservice systems involve complex environmental *configurations* such as memory/CPU limits of microservice and containers, and improper or inconsistent environmental configurations may incur runtime failures. Fourth, microservice invocations are executed or returned in an unexpected *sequence* due to the use of asynchronous invocations (via REST invocations or message queues).

To address the preceding challenges, in this paper, we propose an approach for debugging microservice systems, based on representing microservice system settings as circumstances (specified from various dimensions) such as multi-node and multi-instance deployment. Such representation enables us to conduct delta debugging [18], a technique for simplifying or isolating failure causes (e.g., searching for minimum failure-inducing circumstances) among all circumstances. During delta debugging, a series of delta testing tasks are created to run the test cases with different circumstances. Our experimental study on a medium-size open microservice benchmark system [19] shows that our approach can effectively identify failure-inducing deltas that help identify the root causes.

2 BACKGROUND

Our work is enabled by recent advances in infrastructures and runtime management of microservices. Such advances allow us to manipulate the runtime deployment, configuration, and interactions of microservice systems as required to test the target system with different settings.

Industrial microservice systems usually rely on runtime infrastructures for automating deployment, scaling, and management. Kubernetes [9] is the most widely used runtime infrastructure for microservice systems. It supports the configuration management, service discovery, service registry, and load balancing of microservice systems.

The rise of cloud native applications such as microservice-based ones promotes the introduction of service mesh [11] as a separate layer for handling service-to-service communication. Istio [8] is the most recognized implementation of service mesh for microservices. It supports managing traffic flows between microservices, enforcing access policies, and aggregating telemetry data. Istio can be deployed on Kubernetes. They are combined to provide the required infrastructure for the runtime management of microservices in our work.

3 APPROACH OVERVIEW

Our delta debugging approach for microservice systems can be used when a set of test cases are executed on a microservice system using the same configuration, and at least one of the test cases fails. The approach needs to be run on a containerized environment, allowing the approach to test the target system with different settings. Figure 1 shows an overview of the approach.

The approach includes an infrastructure layer (gray boxes) that automates the deployment and manipulation of microservice systems. The infrastructure layer is built on existing container orchestration platforms (e.g., Kubernetes) and service mesh platforms (e.g., Istio) for microservices. We develop an infrastructure wrapper to provide easy-to-use APIs for applying our delta-debugging approach.

Based on the infrastructure layer, the approach takes as input a set of test cases (including a failing one and some passing ones) and a failure-inducing circumstance, and returns a minimum set of deltas that cause the failure. In particular, a circumstance is defined based on various dimensions (see Section 4.1). The failure-inducing circumstance is the circumstance extracted from the execution of the failing test case. The returned deltas specify a minimum set of differences on the failure-inducing circumstance that can change the testing result of the failing test case and at the same time maintain the testing results of the passing test cases. The approach includes three components: the delta debugging controller, task scheduler, and task executor.

Delta Debugging Controller. The delta debugging controller controls the whole delta debugging process. It first confirms that the failing test case can pass with the simplest circumstance, the one where the value of each dimension is the simplest setting. It then uses the delta debugging algorithm to iteratively search for a minimum set of deltas of the simplest circumstance to make the test case fail. During the process, the controller tests a series of delta sets and for each delta set it creates a delta testing task that runs the test

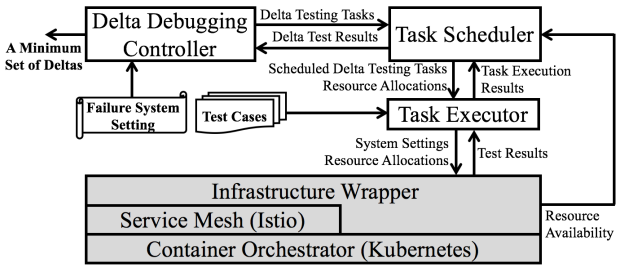

Figure 1: Approach Overview

cases with the circumstance obtained by applying the delta set to the simplest circumstance. To optimize the delta debugging process, the controller dynamically determines the delta testing tasks that need to be executed, and notifies the task scheduler (described next) to add or revoke tasks.

Task Scheduler. The task scheduler schedules the execution of delta testing tasks based on the availability of infrastructure resources (e.g., virtual machines). It maintains a queue of delta testing tasks, and adds or revokes tasks according to notifications from the delta debugging controller. The scheduler monitors the resource availability of the infrastructure and schedules tasks to execute when the required resources are available.

Task Executor. The task executor executes a scheduled delta debugging task on the infrastructure. The executor uses the infrastructure APIs to deploy the target system with the allocated resources and set the environmental configurations and interactions of involved microservices according to the given circumstance. Then the executor runs the test cases and returns test results for further analysis.

The delta debugging controller is the key of the approach and is presented in details in Section 4.

4 DELTA DEBUGGING CONTROLLER

Our delta debugging approach for microservice systems is designed to address unique characteristics of microservices. In particular, the circumstances (each of which is specified from five dimensions) and corresponding deltas considered in our approach reflect the deployment, environmental configurations, and interaction sequences of microservices.

4.1 Dimensions

In general, delta debugging determines circumstances that are relevant for producing a failure [18]. For a microservice system, the relevant circumstances include not only the inputs but also the deployment, environment, and interactions of microservices. A circumstance can be specified from the following five dimensions.

- **Node**. The node dimension specifies the number of nodes (e.g., physical or virtual machines) that can be used by the target system. The more nodes that are provided, the more distributed the instances of the same microservice are. The distributed deployment of the instances of a microservice leads to uncertainties in the network communications with

the microservice, thus incurring failures caused by unexpected network failures or timeout.

- **Instance**. The instance dimension specifies the number of instances of a microservice. Some microservices have explicitly or implicitly defined states. For example, a microservice may store some critical variables in local or remote cache. Without proper coordination, different instances of the same microservice may be in inconsistent states, thus causing failures.
- **Configuration**. The configuration dimension specifies the environmental configurations of a microservice, such as the network configurations and resource (e.g., memory, CPU) limits of microservices or containers. For example, inconsistent configurations of the memory limit of a microservice instance and that of a container where the instance resides may cause the instance to be killed when its memory usage exceeds the memory limit of the container.
- **Sequence**. The sequence dimension specifies the execution and returning sequence of microservice invocations. For a series of asynchronous invocations, the sequence of execution and returning of the invoked microservices is often varying and not consistent with the sequence of requesting. Without proper coordination, the asynchronous invocations may incur unexpected sequences of microservice execution or returning, which in turn cause failures.
- **Input**. The input dimension determines the input of a microservice system. Its influence on a microservice system is similar to the influence of input on an ordinary C program.

Currently we focus on the first four dimensions for reflecting a microservice system's characteristics. The input dimension can be handled in a way similar to the original delta debugging approach [18]. A *circumstance* is a specific combination of the four dimensions involved in test execution. The differences between two circumstances are the *deltas*. The purpose of delta debugging is to isolate the minimum set of failure-inducing deltas with reference to the simplest circumstance. Table 1 shows the values of each dimension in its simplest setting and its general setting. For the first three dimensions, the simplest setting is 1 or the default value, and the general setting can be the values from the given failure-inducing circumstance (i.e., the circumstance derived from the given failing test case). For example, a microservice has 5 instances in the given failure-inducing circumstance, and then its instance number is 1 in the simplest setting and the general setting can be 5. For the sequence dimension, the execution and returning sequence of a series of asynchronous invocations is exactly the requesting sequence of the invocations in the simplest setting, and the general setting can be any other sequences of the invocations. For example, if three microservices are invoked asynchronously in a sequence of S_1, S_2, S_3, then their execution and returning sequence in the simplest setting is also S_1, S_2, S_3, and the general setting can be any other sequence of S_1, S_2, S_3 (e.g., S_3, S_2, S_1).

4.2 Circumstance and Delta Representation

A circumstance is represented as a bit vector that includes one or multiple bits to specify what value to adopt for each dimension. For the node dimension, a bit is used to indicate the number of nodes of the whole system: 0 for adopting the simplest setting (i.e., only one node) and 1 for adopting the number of nodes in the given failure-inducing circumstance. For the instance dimension, multiple bits are used each indicating the number of instances of a microservice: 0 for adopting the simplest setting (i.e., only one instance) and 1 for adopting the number of instances of the microservice in the given failure-inducing circumstance. For the configuration dimension, multiple bits are used each indicating the value of a configuration item: 0 for adopting the simplest setting (i.e., the default value being predefined) and 1 for adopting the value of the configuration item in the given failure-inducing circumstance.

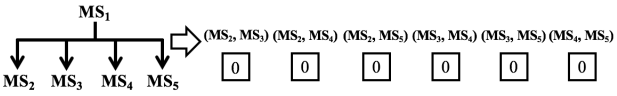

Figure 2: Representation of Execution/Returning Sequence

For the sequence dimension, multiple bits are used to represent the execution/returning sequence of a series of asynchronous invocations, and each bit indicates the order of a pair of invocations: 0 (1) for the order that the first (second) invocation is executed and returned before the second (first) one. Therefore, for n asynchronous invocations, C_n^2 bits are needed to represent the setting of execution/returning sequence. Figure 2 shows an example of the representation of execution/returning sequence. In this example, a microservice MS_1 asynchronously invokes a series of microservice MS_2, MS_3, MS_4, and MS_5, and 6 (C_4^2) bits are used to capture the execution/returning sequence of these invocations. If the four microservices are invoked in the order shown in Figure 2, the simplest setting of execution/returning sequence for this series of asynchronous invocations is [0, 0, 0, 0, 0, 0] based on the pairs defined in the figure. Note that some value combinations of the bits are invalid as these combinations imply cycles in the relative orders of microservice invocations. For the example shown in Figure 2, [0, 1, 0, 0, 0, 0] is an invalid execution/returning sequence as there is a cycle among MS_2, MS_3, and MS_4.

Based on the representation, the simplest circumstance (i.e., the one with each dimension in the simplest setting) can be represented by a bit vector where each bit is set to 0. Thus, an atomic delta based on the simplest circumstance can be represented by a change from 0 to 1 for a bit of the vector, and the purpose of our delta debugging process is to find a minimum set of atomic deltas that cause the failure of a test case.

Note that the representations of the first three dimensions (i.e., node, instance, configuration) can be refined to represent more values. For example, the number of nodes can be any value between 1 and the number of nodes in the given failure-inducing circumstance. To reduce the high cost of delta debugging, we consider only the simplest setting and the general setting from the given failure-inducing circumstance. This strategy can reveal critical deltas in many cases. Note that for the sequence dimension, our representation can cover all the possible execution and returning sequences.

Table 1: Values of Different Dimensions in a Circumstance

Dimension	Target	Simplest Setting	General Setting
Node	the whole system	1	the number of nodes in the given failure-inducing circumstance
Instance	a microservice	1	the number of its instances in the given failure-inducing circumstance
Configuration	a configuration item	the default value	its value in the given failure-inducing circumstance
Sequence	a series of asynchronous invocations	the requesting sequence of the invocations	any other sequences of the invocations

4.3 Delta Debugging Algorithm

Our delta debugging process starts with the confirmation of the testing result with the simplest circumstance. According to the simplest circumstance, all the microservices are deployed on one node; each microservice has only one instance; all the environmental configuration is set to its default value (e.g., unlimited memory); all the asynchronous microservice invocations are executed and returned by the same orders of requests. If the given failing test case still fails with the simplest circumstance, the failure can be thought to be caused by internal faults of related microservices, and further analysis of the root cause can be supported by traditional debugging approaches. Otherwise, the simplest circumstance can be used as the base for delta debugging.

Given the large number of deltas in a microservice system, our aim is to identify *a minimum set of deltas* such that applying the deltas to the simplest circumstance causes the failing test case to produce *failing* results and at the same time causes the passing test cases to maintain *passing* results. In the ideal case, the minimum set contains 1 delta, which can help the developers identify the root cause of the failure. The minimizing delta debugging algorithm [18] is a variant of the original delta debugging algorithm [16], which can be applied to solve our problem. Next, we first present the details on the delta debugging algorithm and then discuss how we apply it in our setting.

Given a failure-inducing circumstance fc and the simplest circumstance sc, let U' be a set of atomic deltas between circumstance fc and sc. In other words, applying all deltas in U' to sc results in fc. In the sequence dimension, multiple bits are used to represent the sequence of a series of asynchronous invocations, and thus we need to use the union of U' and the set of all the atomic deltas in the bits for sequence representation as the universal set of deltas, represented as U.

Let $test(K)$ where $K \subseteq U$ be the testing results of the test cases with the circumstances obtained by applying K to sc. We have $test(\emptyset) = \checkmark$ where $\checkmark$ indicates that all the test cases pass and $test(U) = \times$ where $\times$ indicates that the failing test case fails in the same way of the initial failure and the passing test cases pass. It is possible that $test(K)$ for a subset K results in an unknown case $test(K) =?$, where ? indicates that the failing test case fails in other ways or some passing test cases fail. Formally, the goal is to identify a subset of U, say N, such that $test(N) = \times$ and N is 1-minimal, i.e., $test(N') = \checkmark$ for all $N' \subseteq N$ and $|N'| = |N| - 1$ where $|X|$ is the cardinality of set X. Intuitively, in other words, we would like to find a set of deltas N such that taking away any one of the deltas can change the testing result.

Figure 3 shows the details of the algorithm, denoted as $ddmin(X, n)$, with two inputs. One is a set of deltas denoted as X. Initially X is set to be U. The other is a granularity, denoted as n, for partition

```
1: partition X into n equal subsets △1, ··· △n;
2: for each subset △i do
3:     if test(△i) = × then
4:         return ddmin(△i, 2);
5:     end if
6: end for
7: for each subset △i do
8:     if test(X \ △i) = × then
9:         return ddmin(X \ △i, max(n − 1, 2));
10:    end if
11: end for
12: if n < |X| then
13:    return ddmin(X, min(|X|, 2n));
14: end if
15: return X;
```

Figure 3: DDMin Algorithm: $ddmin(X, n)$

used in the algorithm. Initially, it is set to be 2. At Line 1 of the algorithm, we partition the set of deltas X into n equal-sized partitions $\triangle_1, \cdots, \triangle_n$. Afterwards, we distinguish four cases.

- *Reduce to subset.* If there exists a partition $\triangle_i$ such that $test(\triangle_i)$ fails, we know that $\triangle_i$ is failure-inducing. In such case, we make a recursive call $ddmin(\triangle_i, 2)$ so that we proceed to reduce $\triangle_i$ further. This case yields a "divide and conquer" approach.
- *Reduce to complement.* Otherwise, if there exists a partition $\triangle_i$ such that its complement $X \setminus \triangle_i$ is failure-inducing, i.e., $test(X \setminus \triangle_i)$ fails, we make a recursive call $ddmin(X \setminus \triangle_i), max(n-1, 2))$ so that we proceed to reduce $X \setminus \triangle_i$ further. Note that the second parameter is set to be $n-1$ so that the granularity is not reduced.
- *Increase granularity.* Otherwise, if we can increase the granularity (i.e., $n < |X|$), we recursively call $ddmin(X, min(|X|, 2n))$ so that we can analyze the deltas in X with a finer-grained manner.
- *Done.* Otherwise, we return X as we cannot reduce X further.

The *ddmin* algorithm is designed to reduce the deltas in a way similar to binary search and thus is reasonably efficient (e.g., more efficient compared to the original delta-debugging algorithm [16]). We refer the readers to [18] for detailed discussion on the correctness and complexity of the algorithm. Note that the preceding algorithm assumes that deltas are independent of each other, and it is known [18] that partitioning related deltas in the same partition improves the efficiency of the algorithm.

5 EVALUATION

We conduct an experimental study to evaluate the effectiveness of our approach.

5.1 Settings

We implement our approach itself as a microservice system (including the delta debugging controller, task scheduler, and task executor) running on a containerized environment. We conduct the study based on a medium-size open benchmark microservice system named TrainTicket [19] (with 41 microservices reflecting

real-world industrial practices) after adapting it to the implementation of our infrastructure layer. The environment used in the study includes 13 virtual machines (VMs) provided by a private cloud of Fudan University. Each VM has a 8-core CPU (Intel XEON 3GHz) and 24GB memory, and CentOS 7 installed as the operating system. One of the VMs is used to run our microservice debugging system.

5.2 Results

We conduct an experimental study that uses the approach to debug real microservice failures. The benchmark system TrainTicket includes 11 representative fault cases that are replicated from industrial fault cases identified through an industrial survey. Among the 11 fault cases, we choose 3 fault cases that are related to deployment, environmental configurations, or asynchronous interactions, as shown in Table 2. The 3 fault cases correspond to a dimension (i.e., instance, configuration, or sequence), respectively.

We incorporate the implementations of the 3 fault cases into the benchmark system. For each fault case, we use the corresponding test cases provided by the benchmark system to run the system and produce a failure. We perform a delta debugging process for each fault case with the multiple-cluster setting: 12 VMs are divided into 6 clusters and each cluster has 2 VMs. We record and analyze the delta debugging process for each fault case and obtain the results as shown in Table 3. For each fault case, the table reports the number of deltas in the universal set (#Delta (U)), the number of deltas in the returned delta set (#Delta (R)), the number of tasks created during the process (#Task (C)), the number tasks scheduled to execute (#Task (S)), the number of tasks finished (#Task (F)), the time used (Time), and the indication of the returned deltas. It can be seen that these fault cases involve 36-63 deltas and the returned result includes 1-2 deltas. The whole delta debugging process needs 18-30 minutes to finish. During the process, 32-41 delta testing tasks are created, 20-26 of them are scheduled to execute, and 8-12 of them finish their executions.

To confirm the effectiveness of the approach, we analyze the returned deltas for each fault case. We first understand the indication of the returned deltas and then examine whether the root causes can be identified based on the deltas.

For F1, the returned delta indicates that the failure is induced by the multi-instance deployment of a microservice. The delta accurately reveals the circumstance delta that induces the failure. Based on the indication, the developers need to further check the states of the microservice to identify the root cause.

For F2, the returned delta indicates that the failure is induced by the memory limit of a microservice. Actually the fault involves the improper memory limits of multiple microservices and any of them can cause a failure. The delta reveals the problem of memory limit setting of one of the microservices. Based on the result, the developers can soon identify the root cause of one microservice, and subsequently identify the root causes of the other microservices, e.g., by iteratively performing the delta debugging process.

For F3, the returned 2 deltas indicate that the failure is induced by the orders of two pairs of asynchronous invocations, say (MS_1, MS_2) and (MS_1, MS_3). The real cause of this failure is only the order of the pair (MS_1, MS_3). In this case, the simplest circumstance for the sequence is $< MS_1, MS_2, MS_3 >$ and the given failure-inducing circumstance is $< MS_2, MS_3, MS_1 >$. The order between MS_1 and MS_2, and the order between MS_1 and MS_3 are included in the returned deltas as they are different in the two circumstances, but the failure is induced by only the order between MS_1 and MS_3. In this case, the right failure-inducing delta (i.e., the order between MS_1 and MS_3) is included in the returned deltas, and the developers need to eliminate the other returned delta (i.e., the order between MS_1 and MS_2) by further analyzing the data.

From the preceding analysis, we can see that our approach can effectively perform a delta debugging process, and can identify failure-inducing deltas of different dimensions for helping diagnose the root causes:

1. Instance deltas usually can accurately indicate the multi-instance-deployment problems of microservices. The developers need to further analyze the states of the microservices to identify the root causes.

2. Configuration deltas can identify the configuration problems of some microservices but may miss the same problems of other microservices. The developers need to iteratively perform the delta debugging process to identify the problems of more microservices.

3. Sequence deltas can identify the pairs of microservice invocations that induce the failure but may include irrelevant pairs of invocations in the same sequence. The developers need to further confirm the pairs involved in the deltas to identify the root causes.

5.3 Threats to Validity

The major threats to the external validity of our study lie in the representativeness of the benchmark system, failure cases, and testing environment used in our study. Although the benchmark system is the largest among evaluation subjects for microservice systems in the research literature, it is smaller and less complex than large industrial microservice systems. Although the used failure cases are derived from real industrial cases, these failure cases may be less complex than various failure cases in industrial systems. Therefore, the results of our experimental study may not be generalized to more complex systems, failure cases, or testing environments.

A major threat to the internal validity of our study lies in the uncertainties of the testing environment used in the study. The environment consists of virtual machines provided by a private cloud, and the performance and reliability of the virtual machines are uncertain, thus making the data (e.g., debugging time) collected from the environment likely inaccurate.

6 RELATED WORK

Delta Debugging. Our work is an extension of existing work on debugging, particularly, delta debugging. Delta debugging is proposed for traditional monolithic systems. Zeller *et al.* [16] propose delta debugging for simplifying and isolating failure-inducing inputs. Since then, there have been many extensions. For example, it is extended to isolate cause-effect chains from programs by contrasting program states between successful and failed executions in [17, 18]. Cleve *et al.* [4] extend delta debugging to identify the locations and times where the cause of a failure changes from one variable to another. Sumner *et al.* [12, 13] improve delta debugging in its precision and efficiency by combining it with more precise execution alignment techniques. A cause inference model [14] is

Table 2: Fault Cases Used in the Evaluation

Fault	Description	Dimension
F1	A microservice invocation chain involves two invocations of the same microservice, but the invocations are served by two microservice instances in different states.	Instance
F2	JVM's max memory configuration conflicts with Docker cluster's memory limitation configuration. As a result, Docker sometimes kills the JVM process.	Configuration
F3	A series of asynchronous microservice invocations are returned in an unexpected order.	Sequence

Table 3: Evaluation Results

Fault	#Delta (U)	#Delta (R)	#Task (C)	#Task (S)	#Task (F)	Time	Indication of Returned Deltas
F1	36	1	32	20	10	30 m	the multi-instance deployment of a microservice
F2	63	1	36	23	8	18 m	the memory limit of a microservice
F3	43	2	41	26	12	29 m	the orders of two pairs of asynchronous invocations

also proposed to provide a systematic way of explaining the difference between a failed execution and a successful execution. Burger *et al.* [3] propose an approach called JINSI that combines delta debugging and dynamic slicing for effective fault localization. JINSI can reduce the number of method calls and returns to the minimum number required to reproduce a failure. Misherghi *et al.* [10] propose hierarchical delta debugging to speed up delta debugging by considering hierarchical constraints in the system under debugging. Recently, it is further extended to coarse hierarchical delta debugging [7]. Multiple tools (e.g., [15]) have also been developed to support delta debugging. The preceding approaches are all designed for delta debugging traditional monolithic systems. As discussed earlier, these existing delta-debugging approaches are ineffective for microservices due to the unique characteristics of microservices (i.e., unique deltas and ways of constructing and executing delta testing tasks).

Microservice Analysis. Our work is also related to existing work on analyzing microservice systems. Francesco *et al.* [6] present a systematic study on the state of the art on microservice architectures from three perspectives: publication trends, focus of research, and potential for industrial adoption. One of their conclusions is that research on architecting microservices is still in its initial phase and the balanced involvement of industrial and academic authors is promising. Alshuqayran *et al.* [2] present a study on architectural challenges of microservice systems, the architectural diagrams used for representing them, and the involved quality requirements. Dragoni *et al.* [5] review the development history from objects, services, to microservices, present the current state of the art, and raise some open problems and future challenges. Carlos *et al.* [1] present an initial set of requirements for a candidate microservice benchmark system to be used in research on software architecture. Within the best of our knowledge, there exists no previous research on systematic debugging dedicated to microservices, as focused by our work.

7 CONCLUSION

In this paper, we have proposed a delta debugging approach for microservice systems with the objective of minimizing failure-inducing deltas of circumstances (e.g., deployment, environmental configurations, or interaction sequences) for effective debugging. Our approach includes novel techniques for defining, manipulating, and executing deltas during delta debugging. Our evaluation confirms that our approach can effectively identify failure-inducing deltas that help diagnose the root causes.

ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China under Grant No. 2018YFB1004803. Tao Xie's work was supported in part by National Science Foundation under grants no. CNS-1513939 and CNS-1564274.

REFERENCES

[1] Carlos M. Aderaldo, Nabor C. Mendonca, Claus Pahl, and Pooyan Jamshidi. 2017. Benchmark Requirements for Microservices Architecture Research. In *Proc. IEEE/ACM International Workshop on Establishing the Community-Wide Infrecaseructure for Architecture-Based Software Engineering (ECASE'17)*. 8–13.
[2] Nuha Alshuqayran, Nour Ali, and Roger Evans. 2016. A Systematic Mapping Study in Microservice Architecture. In *Proc. IEEE International Conference on Service-Oriented Computing and Applications (SOCA'16)*. 44–51.
[3] Martin Burger and Andreas Zeller. 2011. Minimizing Reproduction of Software Failures. In *Proc. International Symposium on Software Testing and Analysis (ISSTA'11)*. 221–231.
[4] Holger Cleve and Andreas Zeller. 2005. Locating Causes of Program Failures. In *Proc. International Conference on Software Engineering (ICSE'05)*. 342–351.
[5] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2016. Microservices: Yesterday, Today, and Tomorrow. *CoRR* abs/1606.04036 (2016).
[6] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. 2017. Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. In *Proc. IEEE International Conference on Software Architecture (ICSA'17)*. 21–30.
[7] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2017. Coarse Hierarchical Delta Debugging. In *Proc. IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*. 194–203.
[8] Istio. 2018. Istio. Retrieved February 21, 2018 from https://istio.io/
[9] Kubernetes.Com. 2018. Kubernetes. Retrieved February 21, 2018 from https://kubernetes.io/
[10] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *Proc. International Conference on Software Engineering (ICSE'06)*. 142–151.
[11] William Morgan. 2017. What's a Service Mesh? And Why Do I Need One? Retrieved February 21, 2018 from https://buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/
[12] William N. Sumner and Xiangyu Zhang. 2009. Algorithms for Automatically Computing the Causal Paths of Failures. In *Proc. International Conference on Fundamental Approaches to Software Engineering (FASE'09)*. 355–369.
[13] William N. Sumner and Xiangyu Zhang. 2010. Memory Indexing: Canonicalizing Addresses Across Executions. In *Proc. ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*. 217–226.
[14] William N. Sumner and Xiangyu Zhang. 2013. Comparative Causality: Explaining the Differences between Executions. In *Proc. International Conference on Software Engineering (ICSE'13)*. 272–281.
[15] Delta Tool. 2015. Delta Tool. Retrieved February 21, 2018 from http://delta.tigris.org/
[16] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *Proc. joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'99)*. 253–267.
[17] Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. In *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering 2002, Charleston, South Carolina, USA, November 18-22, 2002*. 1–10.
[18] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200.
[19] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. Poster: Benchmarking Microservice Systems for Software Engineering Research. In *Proc. International Conference on Software Engineering: Companion Proceeedings (ICSE'18)*. 323–324.