1-2022

Delta debugging microservice systems with parallel optimization

Xiang ZHOU
Fudan University

Xin PENG
Fudan University

Tao XIE
University of Illinois at Urbana-Champaign

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Chao JI
Fudan University

See next page for additional authors

Delta Debugging Microservice Systems with Parallel Optimization

Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding

Abstract—Microservice systems are complicated due to their runtime environments and service communications. Debugging a failure 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, we propose a debugging approach for microservice systems based on the delta debugging algorithm, which is to minimalize failure-inducing deltas of circumstances (e.g., deployment, environmental configurations). Our approach includes novel techniques for defining, deploying/manipulating, and executing deltas during delta debugging. In particular, to construct a (failing) circumstance space for delta debugging to minimalize, our approach defines a set of circumstance dimensions that can affect the execution of microservice systems. To automate the testing of deltas, our approach includes the design of an infrastructure layer for automating deployment and manipulation of microservice systems. To optimize the delta debugging process, our approach includes the design of parallel execution for delta testing tasks. Our evaluation shows that our approach is scalable and efficient with the provided infrastructure resources and the designed parallel execution for optimization. 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.

Index Terms—microservices, fault localization, delta debugging, parallel optimization

1 Introduction

A microservice system is composed of small independent microservices that are designed around business capability and owned by small self-contained teams. Those microservices run in their own processes and communicate with lightweight mechanisms such as HTTP resource APIs [1]. An industrial microservice system often includes hundreds to thousands of microservices and each microservice may have hundreds to thousands of instances. For example, Tencent's WeChat system [2] has more than 2,000 microservices running on more than 40,000 backend servers across multiple data centers [3]. Each of those microservice instances might be running inside a container (e.g., Docker [4]) and in a constantly-changing state as they are dynamically scheduled by an orchestrator such as Kubernetes [5].

Beyond the implementations of individual microservices, many failures of microservice systems are due to their runtime environments (e.g., containers), communications, or coordinations [6]. The numerous interactions between microservices are implemented using network communication; therefore, asynchronous invocations are pervasive, while synchronous invocations are considered harmful for microservices due to the multiplicative effect of downtime [1]. The nature of microservice systems has pushed their complexity from the component level (i.e., individual microservices) to the architecture level (i.e., deployment and network communication of microservices) [5], [7]. Moreover, microservices offer extensive deployment flexibility, while a poor deployment choice can increase cost, and compromise performance, scalability, and fault tolerance [8].

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. For example, the instances that serve the requests to a microservice may reside in different locations over the network, and thus an estimation of timeout may be inaccurate. Second, the *instances* of a microservice may be in inconsistent states and thus behave differently. For example, caches are widely used to reduce latency and chattiness [9], which may bring implicit states into microservice instances. An invocation chain may involve the same microservice's different instances that are in different states, thus causing failures that are hard to locate. Third, microservice systems involve complex environmental *configurations* such as memory/CPU limits of microservices and containers, and improper or inconsistent environmental configurations may incur runtime failures. For example, inconsistent memory-limit configurations of microservices and containers may cause the memory usage of a microservice instance to exceed the limit of the container, and in turn cause the instance to be killed by Kubernetes. Fourth, microservice invocations are executed or returned in an unpredictable *sequence* due to the use of asynchronous invocations (via REST invocations or message queues). Missing or improper coordination of the execution

- *X. Peng is the corresponding author (pengxin@fudan.edu.cn).*
- *X. Zhou, X. Peng, C. Ji, W. Li, and D. Ding are with the School of Computer Science and the Shanghai Key Laboratory of Data Science, Fudan University, China; and the Shanghai Institute of Intelligent Electronics & Systems, China.*
- *T. Xie is with the University of Illinois at Urbana-Champaign, USA.*
- *J. Sun is with the Singapore Management University, Singapore.*

or returning of microservice invocations may cause failures due to unmet assumptions.

Microservice developers in practice depend on log analysis and sometimes with visualization and trace analysis for fault analysis and debugging. Our recent industrial survey [10] reveals that they often need to manually examine a large number of logs, and the debugging depends heavily on the developers' experience on the system (e.g., overall architecture and error-prone microservices) and similar fault cases, as well as the technology stack being used. Due to the lack of tool support, they often spend days or even weeks analyzing and debugging microservice faults. Existing approaches of automated fault localization and debugging do not support the multi-dimensional nature of microservice faults. For example, slice-based fault localization [11], [12] reduces the search space for possible locations of a fault by program slicing; spectrum-based fault localization [13], [14] estimates possible fault locations based on program spectra (i.e., program entities) and their coverage status in failed and passed tests. These approaches are based on the analysis of program execution paths in testing, while microservice faults are relevant to not only execution paths but also factors in other dimensions such as microservice instance, interaction sequence, and environmental configuration.

To address the preceding challenges, in this article, 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 [15], a technique for simplifying or isolating failure causes (e.g., searching for minimal 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 earlier work [16] presented the basic concepts and approach of delta debugging microservice systems. However, the execution of delta testing tasks consumes numerous resources (e.g., virtual machines) and involves a complex setting of the deployment, environmental configurations, and interaction sequences of microservice instances. Our extension in this article provides two main techniques to address the challenging requirement for delta debugging microservice systems: high efficiency of executing delta testing tasks. First, to automate the testing of deltas, our approach includes the design of an infrastructure layer (with easy-to-use APIs) for automating deployment and manipulation of microservice systems. This infrastructure layer is based on the existing infrastructure of container orchestration and service mesh. Second, to optimize the delta debugging process, our approach includes the design of parallel execution for delta testing tasks.

Our evaluation shows that our approach is scalable with the provided infrastructure resources (virtual machines), and the optimization can substantially improve the efficiency of delta debugging. Our experimental study on a medium-size open microservice benchmark system [6] shows that the approach can effectively identify failure-inducing deltas that help identify the root causes.

In this work, we make the following main contributions:

- We define a set of dimensions of the circumstances that affect the execution of microservice systems. Based on the definition, we propose a representation of circumstances and deltas, and a delta debugging algorithm for microservice systems.
- We develop an infrastructure layer with easy-to-use APIs for automating the deployment and manipulation of microservice systems for delta debugging.
- We design an optimized parallel scheduling mechanism that supports highly efficient execution of delta testing tasks.
- We conduct an evaluation to demonstrate the scalability and efficiency of our approach and an experimental study to demonstrate the effectiveness of our approach.

The rest of the article is structured as follows. Section 2 presents background knowledge of delta debugging and microservice systems. Section 3 presents an overview of the proposed approach. Section 4 describes the delta debugging controller of the approach. Section 5 introduces the implementation of the infrastructure layer. Section 6 presents the evaluation of the proposed approach. Section 7 discusses related work. Section 8 concludes with future work.

2 BACKGROUND

Our work is based on delta debugging [15], which is an automated debugging technique. On the other hand, our work is enabled by the recent advances in the infrastructures and runtime management of microservices, allowing us to manipulate the runtime deployment, configuration, and interactions of microservice systems as required to test the target system with different settings.

2.1 Delta Debugging

Delta debugging [15] automates the debugging of programs by narrowing down the failure-inducing circumstances. A circumstance is a combination of the factors affecting program execution, including not only the program inputs but also other dimensions (e.g., deployment and environmental configuration) that may affect the program execution. The basic idea of delta debugging is that, by repeating a failed test over and over again under changed circumstances, we can identify what is relevant to the failure and what is not. The changes of circumstances are named deltas.

A delta debugging process starts with a failed test of a given program and the circumstances that may induce the failure. Delta debugging then iteratively tests the program under different circumstances and determines the relevance of the circumstances to the failure based on the test results, until a minimal failure-inducing circumstance is found. In each iteration, the circumstances are partitioned into subsets, and each subset and its complement are tested. If a subset or its complement makes the program fail, the potential failure-inducing circumstances are reduced and the delta debugging process proceeds to focus on the remaining circumstances and to reduce it further.

We refer the readers to the article on delta debugging [15] for a detailed introduction of delta debugging, including the concepts and processes.

2.2 Microservice System

Industrial microservice systems usually rely on runtime infrastructures for automating deployment, scaling, and management. Kubernetes [17] is the most widely used runtime infrastructure for microservice systems. Other microservice infrastructures include Docker Swarm [18], Spring Cloud [19], and Mesos [20]. Kubernetes supports the configuration management, service discovery, service registry, and load balancing of microservice systems. It groups containers that make up an application into logical units (called pods) for easy management and discovery [17]. A pod is the basic building block of Kubernetes and contains one or multiple containers that work together.

The rise of cloud native applications such as microservice-based ones promotes the introduction of service mesh [5] as a separate layer for handling service-to-service communication. The responsibility of the service mesh is to ensure end-to-end performance and reliability of service communication through the complex topology of services. For microservice systems, the service mesh typically includes an array of lightweight network proxies named sidecars, which are seamlessly deployed alongside microservice instances. The service mesh provides a uniform, application-wide facility for introducing visibility and control into the application runtime. For the purpose of delta debugging, the service mesh provides a means to monitor, manage, and control the communication between microservices.

Istio [21] 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. It includes an infrastructure layer (gray boxes) that automates the deployment and manipulation of microservice systems and a control layer (white boxes) that controls the whole scheduling and execution process.

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 the delta debugging approach and testing deltas on demand. The implementation of the infrastructure layer is described in Section 5.

Based on the infrastructure layer, the control layer takes as input a set of test cases (including a failing one and some passing ones) and a failure-inducing circumstance, and returns a minimal set of deltas that cause the failure. In particular, a circumstance is defined based on various

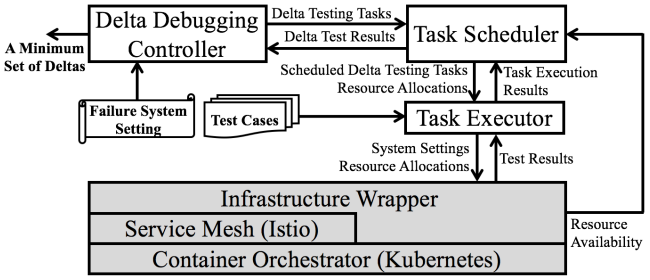


Fig. 1. Approach Overview

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 minimal 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 control layer 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, i.e., the one where the value of each dimension is the simplest setting. It then uses the delta debugging algorithm to iteratively search for a minimal 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 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 detail in Section 4.

4 Delta Debugging Controller

Our delta debugging approach for a microservice system is designed to address unique characteristics of microservices. First, 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. Second, the application and testing of deltas involve complex

deployment and setting of the microservice system in a containerized environment, and thus are time consuming. Therefore, our approach includes a novel optimization of the delta debugging process.

4.1 Dimensions

In general, delta debugging determines circumstances that are relevant for producing a failure [15]. 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 a local or remote cache (e.g., Redis [22]). 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, subsequently causing failures.
- **Input**. The input dimension determines the input of a microservice system and its influence on the 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 [15]. A *circumstance* is a specific combination of the four dimensions involved in a test execution. The differences between two circumstances are the *deltas*. The purpose of delta debugging is to isolate the minimal 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 number of instances 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 a microservice's instances: 0 for adopting the simplest setting (i.e., only one instance) and 1 for adopting the number of the microservice's instances in the given failure-inducing circumstance. For the configuration dimension, multiple bits are used, each indicating the value of a configuration item (i.e., a configuration parameter of a microservice or its runtime environment, e.g., the memory limit of a microservice or its residing container): 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.

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 first 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. This setting indicates that each pair of invocations are executed and returned according to the invocation order, e.g., MS_2 is executed and returned before MS_3. If the four microservices are invoked in the second order shown in Figure 2, the simplest setting is [1,

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

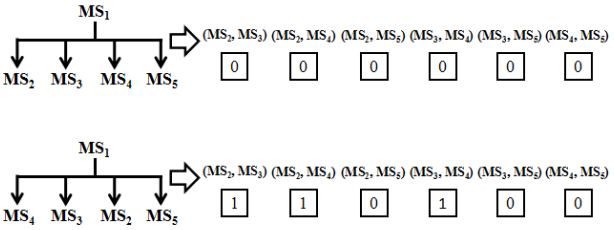


Fig. 2. Representation of Execution/Returning Sequence

1, 0, 1, 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: the second bit indicates that MS_4 is executed and returned before MS_2; while the first bit and the fourth bit indicate that MS_2 is executed and returned before MS_3, and MS_3 before MS_4; thus, MS_2 should be executed and returned before 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 minimal 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.

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 in the same order of requesting. If the failing test case still fails with the simplest circumstance, the failure is likely 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 minimal 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 minimal set contains 1 delta, which can help the developers identify the root cause of the failure. The minimizing delta debugging algorithm [15] is a variant of the original delta debugging algorithm [23], 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 fs and the simplest circumstance ss, let U' be a set of atomic deltas between circumstance fs and ss. In other words, applying all deltas in U' to ss results in fs. 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 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 result of the test cases with the circumstance obtained by applying K to ss. We have $test(\emptyset) = \checkmark$ where $\checkmark$ indicates all the test cases pass and $test(U) = \times$ where $\times$ indicates 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 $N' \subset N$ 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 changes the testing result.

The details of the algorithm, denoted as $ddmin(X, n)$, is shown in Algorithm 1. There are 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 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 a 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

Algorithm 1 DDMin Algorithm: $ddmin(X, n)$

```
partition X into n equal subsets △_1, ··· △_n;
for each subset △_i do
  if test(△_i) = × then
    return ddmin(△_i, 2);
  end if
end for
for each subset △_i do
  if test(X \ △_i) = × then
    return ddmin(X \ △_i, max(n − 1, 2));
  end if
end for
if n < |X| then
  return ddmin(X, min(|X|, 2n));
end if
return X;
```

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 still increase the granularity (i.e., $n < |X|$), we make a recursive 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 [23]). We refer the readers to [15] for a detailed discussion on the correctness and complexity of the algorithm. Note that the algorithm assumes that deltas are independent of each other, and it is known [15] that partitioning related deltas in the same partition improves the efficiency of the algorithm.

4.4 Optimization

Among the four dimensions of deltas, the application of node delta is the most expensive. A delta testing task involving node delta needs to allocate more nodes (e.g., virtual machines) to deploy the specified circumstance. Moreover, the initialization of multiple nodes is much more expensive than the initialization of a single node. Therefore, our optimization of the delta debugging process first considers to test the node delta at the beginning. If the failure-inducing circumstance involves only 1 node, then we can skip the testing of node delta. Otherwise, we run $test(U \setminus \triangle_{node})$ where $\triangle_{node}$ indicates the node delta. If $test(U \setminus \triangle_{node}) = \checkmark$, the failure is caused by the difference of the node number (i.e., multiple-node deployment). If $test(U \setminus \triangle_{node}) = \times$, the failure is irrelevant to node delta, and the rest of the delta debugging process does not need to consider node delta.

The original $ddmin$ algorithm is serial based on the assumption that each delta testing task can be executed efficiently. However, it is not true for a microservice system as the application of deltas involves the complex deployment and setting of the microservice system in a containerized environment. For example, the application of an instance delta involves not only the destroying and creation of Docker instances but also the initialization of microservice instances.

Following the idea of speculative execution, we propose an optimization of the $ddmin$ algorithm based on the parallel execution of delta testing tasks in a cloud environment. The algorithm, denoted as $ddminPar(X, n)$, partitions the set of deltas X into n equal-sized partitions $\triangle_1, \cdots, \triangle_n$ (see Line 1 in Algorithm 1). Instead of serially testing each subset $\triangle_i$ and $X \setminus \triangle_i$, the optimized algorithm creates a series of delta testing tasks of the following types.

- *Reduce to Subset Testing*. For each $\triangle_i$, create a delta testing task $test(\triangle_i)$.
- *Reduce to Complement Testing*. For each $\triangle_i$, create a delta testing task $test(X \setminus \triangle_i)$.
- *Increase Granularity Testing*. For each $\triangle_i$, partition it into two equal-sized partitions $\triangle_{i1}$ and $\triangle_{i2}$, and create two delta testing tasks $test(\triangle_{i1})$ and $test(\triangle_{i2})$.

These delta testing tasks together are added to the task queue of the task scheduler in the same preceding order. The tasks of the same type are ordered by the estimated failure probability and cost of their executions in the following way. We sort the set of deltas X to facilitate the ordering of delta testing tasks as follows. First, the deltas for different dimensions are sorted in the following order: instance, sequence, and configuration. Second, the deltas for each dimension are ranked according to the following rules.

- For the instance dimension, the deltas are ranked in the descending order by the number of instances implied by the delta. The assumption is that the more instances of a microservice are involved in the failure-inducing circumstance, the more likely the multi-instance problem of the microservice causes the failure.
- For the sequence dimension, the deltas are ranked in the descending order by the distance between the corresponding pair of invocations in the invocation sequence. For the example shown in Figure 2 (i.e., MS_1 asynchronously invokes a series of microservice MS_2, MS_3, MS_4, and MS_5), the distance between MS_2 and MS_3 is 1, and the distance between MS_2 and MS_4 is 2. The assumption is that the larger the change of the execution and returning order is, the more likely the sequence problem of the pair of microservices causes the failure.
- For the configuration dimension, the deltas are ranked in the descending order by the strictness implied by the delta. The assumption is that the stricter the environmental configuration (e.g., smaller memory limit) of a microservice is, the more likely the configuration problem of the microservice causes the failure.

Based on the ranked deltas, we then calculate the rank number of a delta testing task as the sum of the rank numbers of the deltas involved in the task. Note that the first delta in the ranked list of a dimension's deltas has rank number 1. Thus, a set of delta testing tasks of the same type are ordered by the rank numbers in the ascending

order. If the rank numbers of two tasks are equal, we further order them by the execution cost in the ascending order; the execution cost is the sum of the time required to apply all the deltas in the task. The time is estimated based on historical data. For example, when a delta indicating the number of a microservice's instances is applied in task execution, the time is recorded for subsequent cost estimation of the same delta.

These tasks are scheduled and executed according to the following rules.

- If a task has been executed, it is not executed again, and the recorded execution result is returned.
- If a task implies an invalid circumstance, e.g., having cycles in a sequence setting or unsatisfying predefined constraints, a success is returned without execution.
- If $test(\triangle_i)$ fails, all the tasks that are created together are canceled (if they are executing) or removed from the queue (if they are waiting for execution) except $test(\triangle_{i1})$ and $test(\triangle_{i2})$.
- If $test(X \setminus \triangle_i)$ fails, all the tasks that are created together are canceled (if they are executing) or removed from the queue (if they are waiting for execution) except those tasks $test(\triangle_j)$ ($\triangle_j \subset X \setminus \triangle_i$).
- If $test(\triangle_{i1})$ or $test(\triangle_{i2})$ fails, all the tasks that are created together are canceled (if they are executing) or removed from the queue (if they are waiting for execution).

Based on the preceding optimization, it is possible that multiple delta testing tasks are executed in parallel, thus improving the efficiency of the approach.

5 Infrastructure

Our current implementation of the infrastructure layer is based on Docker CE 17.03, Kubernetes 1.9, and Istio 0.6. We develop a wrapper to provide easy-to-use APIs for executing delta testing tasks. The wrapper leverages the capabilities provided by the container orchestrator (Kubernetes) and service mesh (Istio), and implements some optimization strategies for initializing delta testing tasks. We also customize Istio to implement the control of execution/returning sequence of asynchronous invocations. The infrastructure layer consists of multiple clusters, each of which includes one or multiple virtual machines. The resources provided for delta testing tasks are supplied and managed by a cluster. When a delta testing task is scheduled to execute, a cluster is allocated to it and initialized for its execution.

5.1 Infrastructure APIs

The wrapper provides the following four sets of APIs, each of which corresponds to a dimension of circumstance. Among these API sets, the node APIs, instance APIs, and configuration APIs are implemented based on Kubernetes REST APIs, and sequence APIs are implemented by customizing Istio.

- **Node**. The node APIs set the number of nodes used for the deployment of a microservice system. The APIs are implemented based on the Kubernetes APIs for expanding/shrinking virtual machines.
- **Instance**. The instance APIs set the number of a microservice's instances. The APIs are implemented based on the Kubernetes APIs for scaling pods.
- **Configuration**. The configuration APIs set the values of the environmental configurations of microservices. The APIs are implemented based on the Kubernetes APIs for configuring pods.
- **Sequence**. The sequence APIs set the execution/returning sequences of a series of asynchronous API invocations. The APIs are implemented based on our customization of Istio.

5.2 Task Initialization Optimization

The execution of a delta testing task includes two parts, i.e., the initialization of the task and the execution of the test cases. The initialization includes complex deployment and configuration of Docker containers and microservice instances, and thus is time consuming. To improve the efficiency of executing delta testing tasks, we make the following optimizations for task initialization in the implementation of the node, instance, and configuration APIs.

Delta initialization. When a cluster is allocated for a delta testing task, the cluster needs to be initialized according to the corresponding circumstance. Instead of restoring the cluster and then applying the specified circumstance, we adopt a strategy of delta initialization. It detects the differences (deltas) between the specified circumstance and the current circumstance used in the last task execution, and incrementally applies the detected deltas on the cluster.

Group initialization. The application of some deltas implies expensive infrastructure operations. For example, using Kubernetes APIs to apply a memory limit of a microservice may cause the restart or even destroying/recreation of corresponding microservice instances. To reduce such expensive operations, we group related deltas and apply the deltas in a group together.

Ordered initialization. When applying a group of deltas, the default execution strategy of Kubernetes may be inefficient. For example, when applying an instance delta of a microservice from 1 to 6 instances and a configuration delta that sets the memory limit of the microservice to 200 Mb, the default execution strategy of Kubernetes may be creating 5 instances of the microservice to apply the instance delta and then setting the memory limit of the 6 instances. As setting the memory limit causes the destroying and recreation of the 6 instances, the application of the group of deltas involves 6 destroyings and 11 creations of microservice instances. To optimize the application of a group of deltas, we define optimized orders for different kinds of deltas. For example, in the case mentioned earlier, the optimized order is to destroy 1 microservice instance (the existing one) and then create 6 microservice instances with the memory limit. In this way, the application of the group of deltas involves only 1 destroying and 6 creations of microservice instances.

5.3 Service Mesh Customization

The purpose of customizing the service mesh (Istio in our work) is to implement the control of execution/returning

sequence of asynchronous invocations. The implementation is based on the sidecar provided by Istio. A pod is the basic building block of Kubernetes and includes one or more microservice instances. Istio is integrated with Kubernetes by injecting a sidecar (a kind of proxy) instance into each pod. The communications between two microservices are then through the sidecars: a microservice request (response) is first routed to the requester (provider) sidecar, then sent to the provider (requester) sidecar, and finally forwarded to the provider (requester). In this way, Istio can monitor and manage the communications between microservices via its corresponding components (mixer and pilot).

For a series of asynchronous microservice invocations with a specified execution/returning sequence $\langle MS_1, \cdots, MS_n \rangle$, we implement the sequence control based on sidecar in the following steps:

1. block the provider sidecars of MS_1 to MS_n to hold the microservice requests and at the same time monitor all the requests;

2. after all the requests have been received by the corresponding provider sidecars, forward the request to MS_i (initially MS_1) for execution and returning;

3. after the requester receives the response of MS_i, forward the request to MS_{i+1} for execution and returning;

4. repeat Step 3 until all the microservice invocations are returned to the requester.

6 Evaluation

We implement our approach itself as a microservice system (including the delta debugging controller, task scheduler, and task executor) running on a containerized environment. To evaluate the effectiveness and efficiency of the approach, we conduct two experimental studies to answer the following two research questions:

RQ1 (Efficiency and Scalability). How efficient is our approach for debugging failures caused by different reasons? How well does it scale with the available resources (virtual machines)?

RQ2 (Effectiveness). How effective is our approach for debugging failures from industrial systems? How well does our approach identify failure-inducing deltas that help diagnose the root causes?

We conduct both studies based on a medium-size open benchmark microservice system named TrainTicket [6] (with 41 microservices reflecting real-world industrial practices) after adapting it to the implementation of our infrastructure layer. The environment used in the studies includes 13 virtual machines (VMs) provided by a private cloud at Fudan University. Each VM has an 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. The source code of the benchmark system, the fault cases, and corresponding test cases can be found in our replication package [24].

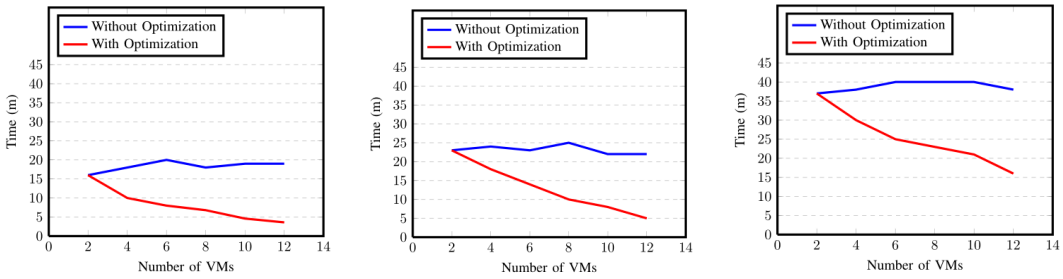
6.1 Efficiency and Scalability (RQ1)

To answer RQ1, we assess the efficiency and scalability of our approach for debugging failures caused by the circumstances of different dimensions. The debugging of failures caused by node circumstances is simple as we only need to test the delta between single-node deployment and multiple-node deployment of the system. Therefore, we design three failure cases, each of which is caused by the circumstances of one dimension (instance, configuration, or sequence). For each failure case, we inject a fault into the implementation or environmental configurations of the benchmark system: the fault of the instance dimension is caused by the lacking of coordination of different instances of a microservice that has an implicit state; the fault of the configuration dimension is caused by inconsistent configurations of a microservice (e.g., the memory limit of JVM is larger than that of Docker); the fault of the sequence dimension is caused by the lacking of coordination of a series of asynchronous invocations of a microservice. For each failure case, we prepare a set of test cases one of which triggers the failure.

We evaluate our approach with two different settings, i.e., with and without the optimization introduced in Section 4.4. For the setting with optimization, we use a multiple-cluster environment for the evaluation: the VMs are divided into multiple clusters each of which has two VMs (one used as the master node of Kubernetes and the other used for the deployment and execution of delta testing tasks). For the setting without optimization, we use a single-cluster environment for the evaluation: one VM is used as the master node of Kubernetes and the other VMs are used for the deployment and execution of delta testing tasks. We evaluate the approach with different supplies of resources from 2 VMs to 12 VMs, i.e., 1 to 6 clusters for the multiple-cluster environment. In the testing for each failure case, the delta debugging approach considers only the deltas of the corresponding dimension. The three failure cases involve different numbers of atomic deltas between the simplest circumstance and the failure-inducing circumstance: 15, 20, and 20 in the cases of instance delta, configuration delta, and sequence delta, respectively. For each execution of the delta debugging process, we collect the returned delta set and the execution time.

After the study, we manually check the returned results and confirm that all of them provide a valid set of deltas for diagnosing the root causes. We evaluate the efficiency and scalability of the approach by analyzing the delta debugging time with different numbers of VMs. The results presented in Figure 3 show that our approach uses 4-40 minutes to finish a delta debugging process. The time used for sequence deltas is much more than the other two kinds of deltas, since the sequence deltas involve many combinations of different orders of microservice pairs.

Without optimization, the approach cannot well utilize the provided VMs, as it needs to sequentially execute all the considered delta testing tasks. The time used for the delta debugging process fluctuates and in some cases even increases with the increase of the provided VMs. We suspect that such result may be caused by the increasing overhead of managing pods over a more distributed environment. In contrast, with optimization, the time used for the delta debugging process continuously decreases from 16-37 minutes to 4-16 minutes with the increase of the provided VMs. The limit of the optimization by parallelization is the time required for the execution of a single delta testing task,



(a) Instance Delta (b) Configuration Delta (c) Sequence Delta

Fig. 3. Efficiency and Scalability Evaluation Results

which is around 2 minutes in our environment according to our evaluation results.

The preceding analysis shows that our approach for delta debugging microservice systems can well scale with the provided VMs; it can complete a delta debugging process in minutes when fully optimized by parallelization.

6.2 Effectiveness (RQ2)

To answer RQ2, 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 collected in an industrial survey. Among the 11 fault cases, we choose 4 fault cases according to the following two criteria: being relevant to deployment, environmental configurations, or asynchronous interactions, along with being the most time consuming and complex according to the feedback from industrial developers. These fault cases are shown in Table 2. The first 3 fault cases correspond to a circumstance dimension, i.e., instance, configuration, or sequence, respectively. The last one involves circumstances of two dimensions (i.e., instance and sequence). According to our recent empirical study [10], the participants used 1.3-2.3 hours for debugging each of these fault cases, or even failed with an improved approach of trace visualization.

We incorporate the implementations of the four 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 then apply our approach to 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 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 1-4 deltas. The whole delta debugging process takes 18-46 minutes. During the process, 32-96 delta testing tasks are created, 20-70 of them are scheduled to execute, and 8-48 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 returned deltas and then examine whether the root causes can be identified based on the deltas.

For F1, the returned delta suggests 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 suggests 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 indication, the developers can soon identify the root cause of one microservice, but need to further identify the root causes of the other microservices, e.g., by iteratively performing the delta debugging process.

For F3, the returned 2 deltas suggest 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 failure 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. The failure however 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. The developer is thus required to eliminate the other returned delta (i.e., the order between MS_1 and MS_2) by further analyzing the data.

F4 involves circumstances of two dimensions, and thus it creates and executes the most delta testing tasks and consumes the most time. For F4, the returned 4 deltas suggest that the failure is induced by the multi-instance deployment of a microservice and the orders of three pairs of asynchronous invocations. Similar to F1, the deltas accurately reveal the microservice that has the problem of multi-instance deployment. Similar to F3, among the returned three pairs, only one is the right failure-inducing delta.

To understand how delta debugging is conducted, we record and analyze the delta debugging process for F4 with 8 VMs (divided into 4 clusters) as the resources. Figure 4 shows the process, which includes 9 rounds. In each round, a series of delta testing tasks are created, scheduled, and executed in parallel. Those tasks are shown as rectangles by the order of task creation. Each task is represented as $\triangle_{par}(dn)$, where par describes the way how the current delta subset is partitioned from the delta subset of the last parallel round and dn is the number of atomic deltas included in the

TABLE 2
Fault Cases Used in Effectiveness 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 limit 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
F4	Multiple asynchronous microservice invocations update a data structure in an unexpected order, and at the same time multiple instances of another microservice in the invocation chain access a storage without coordination.	Instance, Sequence

TABLE 3
Basic Results of Effectiveness Evaluation

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
F4	43	4	96	70	48	46 m	the multi-instance deployment of a microservice, the orders of three pairs of asynchronous invocations

current delta subset. par can be represented as i/n, denoting the ith subset of n equal subsets, or $1 \setminus i/n$, denoting the complement of the ith subset of n equal subsets. A delta testing task can have different results: ✓ indicates that the test passes; $\times$ indicates that the test fails; "Removed" indicates that the task is removed before it is scheduled to execute; "Canceled" indicates that the task is scheduled to execute but canceled before it finishes; "Executed" indicates that the task has been executed in previous rounds.

In each round, the created tasks are scheduled to execute in parallel, and once a task returns a failure, some of the other tasks are removed or canceled. For example, in Round 1, the first 6 tasks are scheduled to execute in parallel, each with a cluster. When some tasks finish, some other tasks (e.g., $\triangle_{1\setminus 4/4}$) are scheduled to execute. When the task $\triangle_{1/2}$ returns failure, Round 1 ends with a subset of 21 atomic deltas. At the same time, some executing tasks (e.g., $\triangle_{1\setminus 4/4}$) are canceled and some tasks in the queue (e.g., $\triangle_{1\setminus 1/4}$) are removed. The tasks that test the subsets of $\triangle_{1/2}$ (i.e., $\triangle_{1/4}$ and $\triangle_{2/4}$) are kept. Round 2 creates a series of tasks to test the subsets of the result of Round 1, and ends with a subset of 16 atomic deltas. The delta debugging process continues till Round 8 finds a subset of 4 deltas. Finally Round 9 has all the created tasks returning unknown, thus confirming that the subset found in Round 8 is the minimal delta subset.

The task statistics of the delta debugging process are shown in Table 4, including the numbers of delta testing tasks that are created, finished, canceled, and removed, respectively. Among the 84 tasks that are created, 51 (60%) are executed and finished, 21 (25%) are executed and canceled, and 12 (15%) are removed without execution. The extra resource consumption (i.e., canceled tasks) of the parallel execution is reasonably low. Moreover, these tasks are canceled before they are finished, and thus the actual overhead is even lower. It can also be seen that 19 (23%) tasks are reused, i.e., their results are used without execution in subsequent rounds.

The preceding analysis shows that the approach can well utilize parallelization and optimized scheduling to efficiently perform a delta debugging process. The approach 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 indicate 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 indicate 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.

TABLE 4
Task Statistics in the Delta Debugging Process of F4

Round	Created	Finished	Canceled	Removed	Reused
Round 1	11	6	3	2	0
Round 2	13	8	3	2	0
Round 3	8	2	4	2	3
Round 4	10	5	3	2	1
Round 5	20	15	3	2	0
Round 6	5	5	0	0	5
Round 7	4	2	2	0	4
Round 8	9	4	3	2	0
Round 9	4	4	0	0	6
Total	84	51 (60%)	21 (25%)	12 (15%)	19 (23%)

6.3 Threats to Validity

The major threats to the external validity of our studies lie in the representativeness of the benchmark system, failure cases, and testing environment used in our studies. 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. The testing environment used in our studies may not represent more complex cloud environments with higher overhead for parallelization. Therefore, the results of our experimental studies may not be generalized to larger or more complex systems, failure cases, or testing environments.

A major threat to the internal validity of our studies lies in the uncertainties of the testing environment used in the studies. 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.

7 Related Work

Delta Debugging. Our work is an extension of existing work on debugging, particularly, delta debugging. Delta

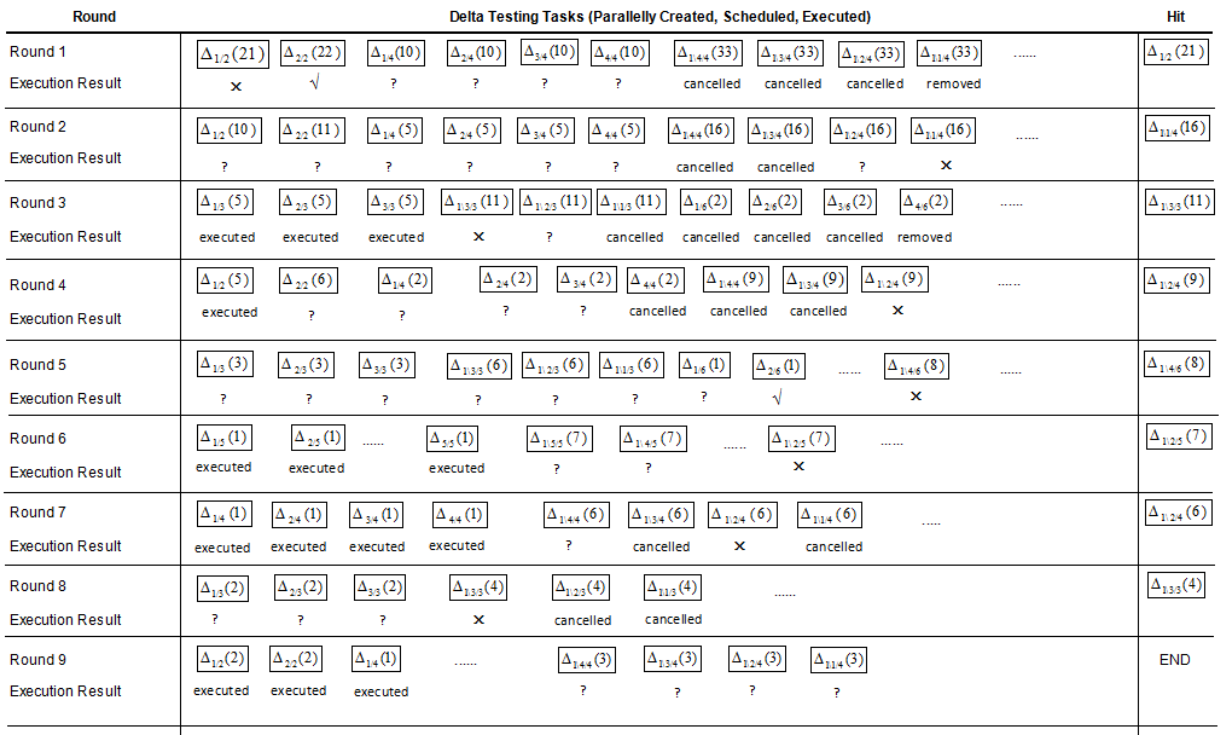


Fig. 4. Delta Debugging Process of F4

debugging is proposed for traditional monolithic systems. Zeller *et al.* [23] 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 [15], [25]. Cleve *et al.* [26] extend delta debugging to identify the locations and times where the cause of a failure changes from one variable to another. Sumner *et al.* [27], [28] improve delta debugging in its precision and efficiency by combining it with more precise techniques of execution alignment. A cause inference model [29] is also proposed to provide a systematic way of explaining the difference between a failed execution and a successful execution. Burger *et al.* [30] 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.* [31] 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 [32]. Multiple tools (e.g., [33]) 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 microservice systems due to the unique characteristics of microservice systems (i.e., unique deltas and ways of constructing and executing delta testing tasks).

Debugging Concurrent/Distributed Programs. Our work is related to existing work on debugging concurrent programs [34], [35], [36] and distributed systems [37], [38], [39], [40]. In view of the difficulty in debugging concurrent programs and distributed systems, a variety of different approaches have been proposed. We next discuss some samples of these approaches. Choi *et al.* [41] apply delta debugging to multi-threaded failures, defining the differences between a failing test execution and passing test execution in terms of the scheduling. Asadollah *et al.* [42] present a systematic study on debugging concurrent and multicore software in the decade between 2005 and 2014. Bailis *et al.* [37] present a survey on recent approaches for debugging distributed systems. Their conclusion is that the state of the art of debugging distributed systems is still in its infancy. Giraldeau *et al.* [38] propose an approach to visualize the execution of distributed systems using scheduling, network, and interrupt events. Aguerre *et al.* [39] present a simulation and visualization platform that incorporates a distributed debugger. Beschastnikh *et al.* [40] discuss the key features and debugging challenges of distributed systems and present a visualization tool named ShiViz. Leonardo *et al.* [43] introduce a lightweight fault localization approach for cloud systems; it can localize faults with high precision, by relying on only lightweight positive training. In contrast to the preceding previous work, our work is the first to conduct delta debugging for microservice systems. Compared to debugging traditional concurrency systems, debugging microservice systems is considerably more challenging due to the high complexity and dynamism of microservices. A microservice system includes numerous microservice instances running on a large number of nodes. These instances are created and destroyed dynamically, and involve complex environmental configurations in different layers (e.g., JVM, container, VM). Moreover, the invocation chains of microservices can be very long and most of the invocations

are asynchronous. A basic way for debugging distributed systems is tracing and visualizing system executions over system nodes. However, for microservice systems, there lacks a natural correspondence between microservices and system nodes in distributed systems, as microservice instances can be dynamically created and destroyed [10].

Debugging Service-Oriented Architecture (SOA) Applications. Our work is closely related to existing work on debugging SOA applications. Arora *et al.* [44] present an approach for automatically reproducing production failures to provide a sandboxed debugging environments for SOA applications. This approach requires developers to attach debuggers and analysis tools in the debugging environments. Chen [45] applies the methodology of spectrum-based fault localization to SOA applications. Alodib and Bordbar [46] present a model-based approach to fault diagnosis in SOA systems. It extends techniques of Discrete Event Systems to monitor service interactions and identify possible failures. These approaches do not consider the multi-dimensional nature of microservice faults, e.g., nodes, microservice instances, interaction sequences, and environmental configurations.

Microservice Analysis. Our work is also related to existing work on analyzing microservice systems. Francesco *et al.* [47] 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.* [48] present a study on architectural challenges of microservice systems, the architectural diagrams used for representing them, and the involved quality requirements. Dragoni *et al.* [49] 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.* [7] 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.

8 Conclusion

In this article, 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 more effective debugging. Our approach includes novel techniques for defining, manipulating, and executing deltas during delta debugging. Our evaluation shows that our approach is scalable and efficient with the provided infrastructure resources. It also confirms that our approach can effectively identify failure-inducing deltas for helping diagnose the root causes.

Our current approach is limited in the granularity of the supported atomic deltas. For example, we consider only the difference between the default value (e.g., unlimited memory) and the value in a failure setting (e.g., a memory limit of 200 Mb) of a configuration item in the configuration dimension. Moreover, the circumstance dimensions that our current approach considers are also limited. In our future work, we plan to further improve the approach by refining the granularity of deltas and at the same time considering additional circumstance dimensions (e.g., invocation chains).

Acknowledgments

This work was supported by the National Key Research and Development Program of China under Grant No. 2018YFB1004803, NSF under grants no. CNS-1513939, CNS-1564274, CCF-1816615, and a grant from Huawei.

References

[1] J. Lewis and M. Fowler, "Microservices: a definition of this new architectural term," 2014. [Online]. Available: http://martinfowler.com/articles/microservices.html

[2] Tencent, "WeChat," 2018. [Online]. Available: https://www.wechat.com

[3] H. Zheng, D. Li, B. Liang, X. Zeng, W. Zheng, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for android: Towards getting there in an industrial case," in *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 253–262.

[4] Docker.Com, "Docker," 2018. [Online]. Available: https://docker.com/

[5] W. Morgan, "What's a service mesh? and why do i need one?" 2017. [Online]. Available: https://buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/

[6] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking microservice systems for software engineering research," in *40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 323–324.

[7] C. M. Aderaldo, N. C. Mendonca, C. Pahl, and P. Jamshidi, "Benchmark requirements for microservices architecture research," in *1st IEEE/ACM International Workshop on Establishing the Community-Wide Infrecaseructure for Architecture-Based Software Engineering, ECASE@ICSE 2017, Buenos Aires, Argentina, May 22, 2017*, 2017, pp. 8–13.

[8] A. R. Sampaio, H. Kadiyala, B. Hu, J. Steinbacher, T. Erwin, N. Rosa, I. Beschastnikh, and J. Rubin, "Supporting microservice evolution," in *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, 2017, pp. 539–543.

[9] A. W. S. Whitepaper, "Microservices on AWS," 2017. [Online]. Available: https://d0.awsstatic.com/whitepapers/microservices-on-aws.pdf

[10] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[11] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang, "Slice-based statistical fault localization," *Journal of Systems and Software*, vol. 89, pp. 51–62, 2014.

[12] E. Alves, M. Gligoric, V. Jagannath, and M. d'Amorim, "Fault-localization using dynamic slicing and change impact analysis," in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, 2011, pp. 520–523.

[13] W. Wen, "Software fault localization based on program slicing spectrum," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 1511–1514.

[14] A. Perez and R. Abreu, "A qualitative reasoning approach to spectrum-based fault localization," in *40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 372–373.

[15] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 183–200, 2002.

[16] X. Zhou, X. Peng, T. Xie, J. Sun, W. Li, C. Ji, and D. Ding, "Delta debugging microservice systems," in *33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 802–807.

[17] Kubernetes.Com, "Kubernetes," 2018. [Online]. Available: https://kubernetes.io/

[18] DockerSwarm.Com, "Docker swarm," 2018. [Online]. Available: https://docs.docker.com/swarm/

[19] SpringCloud.Com, "Spring cloud," 2018. [Online]. Available: http://projects.spring.io/spring-cloud/

[20] Mesos.Com, "Mesos," 2018. [Online]. Available: http://mesos.apache.org/

[21] Istio, "Istio," 2018. [Online]. Available: https://istio.io/

[22] Redis.Io, "redis.io," 2016. [Online]. Available: https://redis.io/

[23] A. Zeller, "Yesterday, my program worked. today, it does not. why?" in *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999*, 1999, pp. 253–267.

[24] Replication-Package, "Delta debugging microservice systems," 2018. [Online]. Available: http://45.77.211.219/msDeltaDebugging/

[25] A. Zeller, "Isolating cause-effect chains from computer programs," in *Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering 2002, Charleston, South Carolina, USA, November 18-22, 2002*, 2002, pp. 1–10.

[26] H. Cleve and A. Zeller, "Locating causes of program failures," in *27th International Conference on Software Engineering, ICSE 2005, 15-21 May 2005, St. Louis, Missouri, USA*, 2005, pp. 342–351.

[27] W. N. Sumner and X. Zhang, "Memory indexing: canonicalizing addresses across executions," in *18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, 2010, pp. 217–226.

[28] ——, "Algorithms for automatically computing the causal paths of failures," in *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009.*, 2009, pp. 355–369.

[29] ——, "Comparative causality: explaining the differences between executions," in *35th International Conference on Software Engineering, ICSE 2013, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 272–281.

[30] M. Burger and A. Zeller, "Minimizing reproduction of software failures," in *20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, 2011, pp. 221–231.

[31] G. Misherghi and Z. Su, "HDD: hierarchical delta debugging," in *28th International Conference on Software Engineering, ICSE 2006, Shanghai, China, May 20-28, 2006*, 2006, pp. 142–151.

[32] R. Hodován, Á. Kiss, and T. Gyimóthy, "Coarse hierarchical delta debugging," in *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, 2017, pp. 194–203.

[33] D. Tool, "Delta tool," 2015. [Online]. Available: http://delta.tigris.org/

[34] S. Park, R. W. Vuduc, and M. J. Harrold, "UNICORN: a unified approach for localizing non-deadlock concurrency bugs," *Softw. Test., Verif. Reliab.*, vol. 25, no. 3, pp. 167–190, 2015.

[35] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "Spectrum-based multiple fault localization," in *24th IEEE/ACM International Conference on Automated Software Engineering, ASE 2009, Auckland, New Zealand, November 16-20, 2009*, 2009, pp. 88–99.

[36] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: fault localization in concurrent programs," in *32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, 2010, pp. 245–254.

[37] P. Bailis, P. Alvaro, and S. Gulwani, "Research for practice: tracing and debugging distributed systems; programming by examples," *Commun. ACM*, vol. 60, no. 7, pp. 46–49, 2017.

[38] F. Giraldeau and M. Dagenais, "Wait analysis of distributed systems using kernel tracing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 8, pp. 2450–2461, 2016.

[39] C. Aguerre, T. Morsellino, and M. Mosbah, "Fully-distributed debugging and visualization of distributed systems in anonymous networks," in *International Conference on Computer Graphics Theory and Applications and International Conference on Information Visualization Theory and Applications, GRAPP & IVAPP 2012, Rome, Italy, 24-26 February, 2012*, 2012, pp. 764–767.

[40] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst, "Debugging distributed systems," *Commun. ACM*, vol. 59, no. 8, pp. 32–37, 2016.

[41] J. Choi and A. Zeller, "Isolating failure-inducing thread schedules," in *International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, 2002, pp. 210–220.

[42] S. A. Asadollah, D. Sundmark, S. Eldh, H. Hansson, and W. Afzal, "10 years of research on debugging concurrent and multicore software: a systematic mapping study," *Software Quality Journal*, vol. 25, no. 1, pp. 49–82, 2017.

[43] L. Mariani, C. Monni, M. Pezzè, O. Riganelli, and R. Xin, "Localizing faults in cloud systems," *CoRR*, vol. abs/1803.00356, 2018.

[44] N. Arora, J. Bell, F. Ivancic, G. E. Kaiser, and B. Ray, "Replay without recording of production bugs for service oriented applications," in *33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 452–463.

[45] C. Chen, "Automated fault localization for service-oriented software systems," Ph.D. dissertation, Delft University of Technology, Netherlands, 2015.

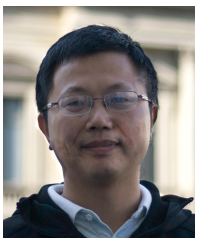
[46] M. I. Alodib and B. Bordbar, "A model-based approach to fault diagnosis in service oriented architectures," in *7th IEEE European Conference on Web Services, ECOWS 2009, 9-11 November 2009, Eindhoven, The Netherlands*, 2009, pp. 129–138.

[47] P. D. Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in *2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3-7, 2017*, 2017, pp. 21–30.

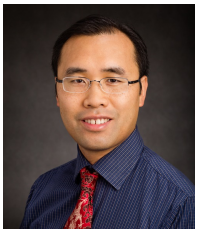
[48] N. Alshuqayran, N. Ali, and R. Evans, "A systematic mapping study in microservice architecture," in *9th IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2016, Macau, China, November 4-6, 2016*, 2016, pp. 44–51.

[49] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *CoRR*, vol. abs/1606.04036, 2016.

Xiang Zhou is a PhD student of the School of Computer Science at Fudan University, China. He received his master degree from Tongji University in 2009. His PhD work focuses on the development and operation of microservice systems.



Xin Peng is a professor of the School of Computer Science at Fudan University, China. He received Bachelor and PhD degrees in computer science from Fudan University in 2001 and 2006. His research interests include data-driven intelligent software development, software maintenance and evolution, mobile and cloud computing. His work won the Best Paper Award at the 27th International Conference on Software Maintenance (ICSM 2011), the ACM SIGSOFT Distinguished Paper Award at the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2018), the IEEE TCSE Distinguished Paper Award at the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME 2018).



Tao Xie is a professor and Willett Faculty Scholar in the Department of Computer Science at the University of Illinois at UrbanaChampaign, USA. He received his PhD degree in Computer Science from the University of Washington at Seattle in 2005. His research interests are software testing, program analysis, software analytics, software security, intelligent software engineeirng, and educational software engineering. He is a Fellow of the IEEE.

Jun Sun is currently an associate professor at Singapore Management University (SMU). He received Bachelor and PhD degrees in computing science from National University of Singapore (NUS) in 2002 and 2006. In 2007, he received the prestigious LEE KUAN YEW postdoctoral fellowship. He has been a faculty member since 2010. He was a visiting scholar at MIT from 2011-2012. Jun's research interests include software engineering, formal methods, program analysis and cyber-security.

Chao Ji is a Master student of the School of Computer Science at Fudan University, China. He received his Bachelor degree from Fudan University in 2017. His work focuses on the development and operation of microservice systems.

Wenhai Li is a Master student of the School of Computer Science at Fudan University, China. He received his Bachelor degree from Fudan University in 2016. His work focuses on the development and operation of microservice systems.

Dan Ding is a Master student of the School of Computer Science at Fudan University, China. She received her Bachelor degree from Fudan University in 2017. Her work focuses on on the development and operation of microservice systems.