Trace-based Multi-Dimensional Root Cause Localization of Performance Issues in Microservice Systems

Chenxi Zhang[*]
Fudan University
China

Zhen Dong[*] [†]
Fudan University
China

Xin Peng[*]
Fudan University
China

Bicheng Zhang[*]
Fudan University
China

Miao Chen[*]
Fudan University
China

ABSTRACT

Modern microservice systems have become increasingly complicated due to the dynamic and complex interactions and runtime environment. It leads to the system vulnerable to performance issues caused by a variety of reasons, such as the runtime environments, communications, coordinations, or implementations of services. Traces record the detailed execution process of a request through the system and have been widely used in performance issues diagnosis in microservice systems. By identifying the execution processes and attribute value combinations that are common in anomalous traces but rare in normal traces, engineers may localize the root cause of a performance issue into a smaller scope. However, due to the complex structure of traces and the large number of attribute combinations, it is challenging to find the root cause from the huge search space. In this paper, we propose TraceContrast, a trace-based multi-dimensional root cause localization approach. TraceContrast uses a sequence representation to describe the complex structure of a trace with attributes of each span. Based on the representation, it combines contrast sequential pattern mining and spectrum analysis to localize multi-dimensional root causes efficiently. Experimental studies on a widely used microservice benchmark show that TraceContrast outperforms existing approaches in both multi-dimensional and instance-dimensional root cause localization with significant accuracy advantages. Moreover, TraceContrast is efficient and its efficiency can be further improved by parallel execution.

CCS CONCEPTS

• **Software and its engineering → Cloud computing**; **Software performance**; **Software reliability**.

[*]C. Zhang, Z. Dong, X. Peng, B. Zhang, and M. Chen are with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China and the Shanghai Collaborative Innovation Center of Intelligent Visual Computing, China
[†]Z. Dong is the corresponding author (zhendong@fudan.edu.cn).

KEYWORDS

Microservice, Root Cause Analysis, Tracing

ACM Reference Format:
Chenxi Zhang, Zhen Dong, Xin Peng, Bicheng Zhang, and Miao Chen. 2024. Trace-based Multi-Dimensional Root Cause Localization of Performance Issues in Microservice Systems. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639088

1 INTRODUCTION

Microservice systems have gained popularity in recent years for their ability to enhance scalability, flexibility, and agile iteration capabilities in large-scale systems [48]. Industrial microservice systems typically consist of hundreds to thousands of services, with each service having tens to thousands of independently deployed instances that communicate with each other through lightweight mechanisms [18]. Meanwhile, the fine-grained decomposition of microservices makes the system vulnerable to performance issues. To operate microservice systems reliably and with high uptime, performance issues must be detected quickly and the root causes pinpointed.

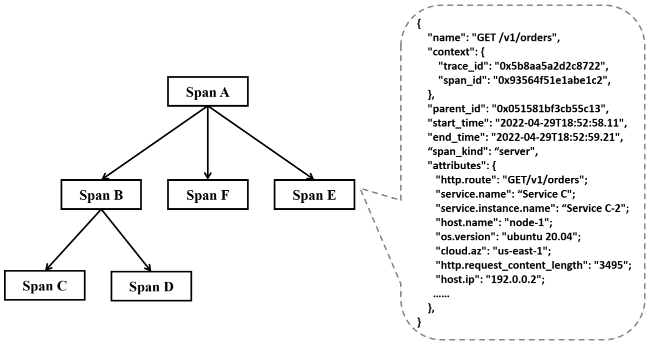
Performance issues in microservice systems are complicated due to the dynamic and complex interactions and runtime environments [48]. Microservice systems usually have complex invocation chains, each invocation chain may involve several components (e.g., service, service instance, host) in the microservice system and the interaction between these components (e.g., service invocation, database invocation). Moreover, the systems are highly heterogeneous, such as deploying multiple versions of services at the same time and running with different infrastructure configurations [48, 50]. It leads to the multi-dimensional nature of the root causes of microservice performance issues, which could be the runtime environments, communications, coordinations, or implementations of services [48]. Therefore, it is challenging to diagnose performance issues in microservice systems. Nowadays, some monitoring infrastructures (e.g., Loki [7], Prometheus [10]) can observe what happens in a microservice instance (e.g., the error rate per minute), but they tell us little about the fine-grained context of the interactions between microservices.

To support fine-grained fault diagnosis in microservice systems, distributed tracing has been widely adopted in industrial microservice systems and becomes a part of their infrastructures [9, 13, 31]. Each trace describes the detailed execution process of a request through the system, and the details of each operation in it are described by a structured log called a span. As the example shown in the Figure 1, the trace describes the execution of a request, and the attributes in span E record the detailed information about the operation, such as the URL of the API, service instance name, host name, and length of the request content. Operation engineers and developers usually analyze the structure and attributes of the traces to diagnose performance issues. By identifying the execution process and attribute value combination that are common in anomalous traces but rare in normal traces, engineers may localize the root cause of a performance issue into a smaller scope. For example, if a performance issue occurs due to a bug between a specific service version (i.e., version $V1$ of $ServiceB$) and a certain client service (i.e., $ServiceA$). Then, engineers may find that a large number of traces containing this invocation are anomalous. And the root cause of this example can be represented as a sequence consisting of attribute values $< ServiceA \rightarrow (ServiceB, V1) >$, which we call a multi-dimensional root cause in this paper. However, due to the scale and complexity of microservice systems, manual investigation of trace data is tedious and inefficient for localizing the multi-dimensional root cause.

The main challenge of multi-dimensional root cause localization in microservice systems is the huge search space due to the complex structure of traces and the large number of attribute combinations. Existing multi-dimensional root cause localization approaches [5, 20, 22–24, 35, 38] represent telemetry data (e.g., logs, issue reports, metrics) as multi-dimensional tabular data to search the root cause. However, if we represent traces as multi-dimensional tabular data, we will ignore the complex structures of traces, which leads to an inability to locate performance issues related to the execution path. Existing trace analysis approaches [3, 21, 26, 40, 47, 49] only localize the root causes at the service/instance level, which ignores the fact the multi-dimensional nature of microservice performance issues.

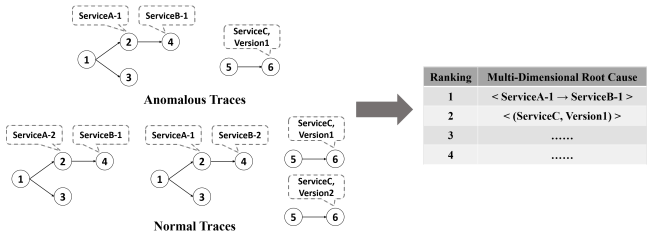
To address the preceding challenges, we propose TraceContrast, a multi-dimensional root cause localization approach for performance issues in microservice systems. TraceContrast aims to find the specific execution process and attribute value combination that are common in anomalous traces but rare in normal traces which are more like the multi-dimensional root cause. TraceContrast uses a sequence representation to describe the complex structure of a trace with the attributes of each span. Such representation enables us to combine contrast sequential pattern mining and spectrum analysis to achieve high-efficient multi-dimensional root cause localization. Specifically, TraceContrast first extracts the critical path from each trace and represents it as an event sequence. Then it detects which critical paths are affected by the performance issue. Based on the anomaly and normal critical paths, TraceContrast mine candidate multi-dimensional root causes based on a parallel contrast sequential pattern mining algorithm. Finally, TraceContrast ranks candidate root causes based on a spectrum formula and removes the redundant root causes by combining the domain knowledge of microservice systems.

To evaluate the accuracy and efficiency of TraceContrast, we conduct a series of experimental studies on a medium-scale microservice benchmark system. The result shows that TraceContrast outperforms existing trace-based root cause localization approaches



```
{
  "name": "GET /v1/orders",
  "context": {
    "trace_id": "0x5b8aa5a2d2c8722",
    "span_id": "0x93564f51e1abe1c2",
  },
  "parent_id": "0x051581bf3cb55c13",
  "start_time": "2022-04-29T18:52:58.11",
  "end_time": "2022-04-29T18:52:59.21",
  “span_kind": “server",
  "attributes": {
   "http.route": "GET/v1/orders";
   "service.name": “Service C";
   "service.instance.name": “Service C-2";
   "host.name": "node-1";
   "os.version": "ubuntu 20.04";
   "cloud.az": "us-east-1";
   "http.request_content_length": "3495";
   "host.ip": "192.0.0.2";
  ......
  },
}
```

Figure 1: An Example of Trace and Span Log



Ranking	Multi-Dimensional Root Cause
1	< ServiceA-1 → ServiceB-1 >
2	< (ServiceC, Version1) >
3	......
4	......

Figure 2: An Example of Multi-Dimensional Root Cause in Microservice Systems

by 195.3% and 101.2% on average in terms of top-5 hit ratio in multi-dimensional and instance dimensional root cause localization respectively. Moreover, the experimental result confirms the efficiency of TraceContrast and shows that our approach is scalable with the provided computing resources.

2 BACKGROUND AND MOTIVATION

2.1 Background

When a microservice system receives a request, the execution process can involve multiple components in the system. In order to understand and analyze the execution process, it is important to trace and record the end-to-end execution process of each request. To achieve this, distributed tracing has become a widely adopted technique in microservice systems [19]. Distributed tracing is an approach used to profile and monitor distributed systems, which record the end-to-end execution process of a request as a set of structured logs. There are a growing number of industry and open-source distributed tracing frameworks, such as Dapper [31], Zipkin [34], OpenTelemetry [9], and SkyWalking [32]. Distributed tracing frameworks typically support manual or automatic instrumentation for each service instance, and when the instrumentation point is triggered, a log is generated to record information about this invocation. These logs are then collected by the distributed tracing framework to reconstruct the complete traces for further analysis.

According to the OpenTelemetry specification, a trace consists of one or more spans and can be represented as a directed acyclic graph [9]. Each span is a structured log representing a unit of work or operation [9] with a variety of information. A span log consists of basic information, span context, span attributes, etc. The basic

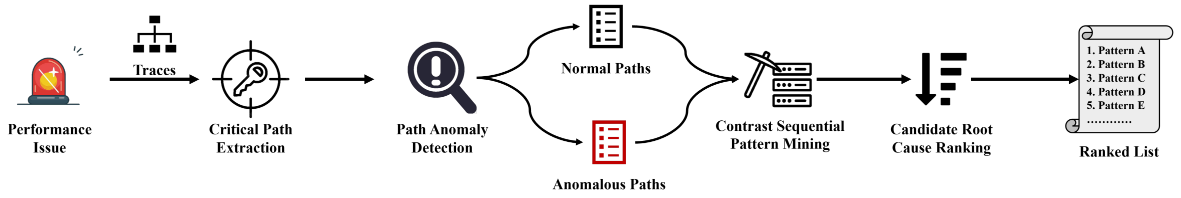


Figure 3: Overview of TraceContrast

information records the name, start time, end time, and type (e.g., Client, Server, Producer, and Consumer) of the span. Span context records trace id, span id, and other information which are needed to propagate to the downstream services. Span attributes record the information about the current operation, such as service name, database statement, request content length, and any other data required for trace analysis. OpenTelemetry have defined a number of attributes for common operations (e.g., database invocation, HTTP invocation) and resources (e.g., service, container, cloud infrastructure) in microservice systems. And the SDKs provided by OpenTelemetry will automatically attach the predefined attributes to the span log when it is generated. These SDKs also support users to customize the attributes which are needed. An example of the trace and span log according to the OpenTelemetry specification is shown in Figure 1. For easy understanding, we removed all client spans in that trace, after which the trace contains a total of 6 spans, where Span A is the root span. As shown in Figure 1, the span log of Span E records the span name, the trace ID, the span ID, the ID of the parent span, the start/end timestamp, and the kind of the span. The attributes in the span log record resource information (e.g., OS version, service name, availability zone) and operation information (e.g., request content length, URL) corresponding to the invocation that generated the span.

2.2 Motivation

In practice, operation engineers and developers usually locate the root cause of a performance issue by comparing the structure and attributes of anomalous traces and normal traces. Figure 2 shows an example of multi-dimensional root cause localization of microservice systems. The figure shows the normal traces and anomalous traces when a performance failure occurs. For the sake of simplicity, we represent the different operations with different numbers and only show the attributes that differ in the normal and anomalous traces. By comparing anomalous and normal traces, we can find the potential root causes. Moreover, each potential root cause can be represented as a sequence consisting of attribute values, which is called a multi-dimensional root cause. In this example, we can find that the two most suspicious root causes are $< ServiceA\text{-}1 \longrightarrow ServiceB\text{-}1 >$ and $< (ServiceC, Version1) >$. It can be found that $< ServiceA\text{-}1 \longrightarrow ServiceB\text{-}1 >$ only appears in anomalous traces, which makes it more likely to be the real root cause. Based on the located multi-dimensional root causes, operations engineers and developers can further analyze and mitigate the performance issue.

Existing trace-based root cause localization approaches [3, 21, 26, 40, 49] uses spectrum analysis, heuristic methods, or machine learning techniques to localize the root cause of performance issues. However, these approaches only localize the root causes at the service/instance level, which does not support the multi-dimensional nature of microservice performance issues. Moreover, these approaches do not make good use of the attribute information in the trace, they just represent traces as invocation pairs or execution paths at the service/instance level for root cause localization. As shown in the example in Figure 2, if we do not use the attribute information (e.g., service version), we cannot locate the suspicious root cause $< (ServiceC, Version1) >$.

Existing multi-dimensional root cause localization approaches [5, 20, 22–24, 35, 38] represent telemetry data (e.g., logs, issue reports, metrics) as a multi-dimensional table and then search the attribute value combination where the anomalies are mostly concentrated as the multidimensional root cause. However, different from other telemetry data (e.g., logs, issue reports, metrics), traces usually have a complex structure that is not suitable to be represented as a multi-dimensional table. If we represent trace as tabular data by converting each span log as a row in a multi-dimensional table, we will lose the structural information of trace and thus fail to locate the root cause of some performance issues. For example, if we represent the traces in Figure 2 into tabular data, we will lose the execution process information. Therefore, we cannot locate the suspicious root cause $< ServiceA\text{-}1 \longrightarrow ServiceB\text{-}1 >$.

Based on the analysis, we can see that in order to locate the multi-dimensional root cause in microservice systems, we need to combine the trace structure and the attributes in each span log. Therefore, we propose a sequence representation to describe the complex structure of a trace with the attributes in each span log. Then, we combine contrast sequential pattern mining and spectrum analysis to locate multi-dimensional root causes in microservice systems.

3 APPROACH

TraceContrast is a multi-dimensional root cause localization approach for performance issues in microservice systems. It takes traces as input and outputs a ranked list consist of candidate multi-dimensional root causes.

An overview of TraceContrast is shown in Figure 3, the whole process includes four steps. When a performance issue occurs, TraceContrast is triggered to localize the root cause based on the input traces. Critical Path Extraction extracts the critical path from each trace and represents it as an event sequence. Path Anomaly Detection detects anomalous traces and then identifies which critical paths are affected by the performance issue. Contrast Sequential Pattern Mining uses a parallel contrast sequential pattern mining algorithm to find candidate root causes. Candidate Root Cause Ranking ranks the candidate root causes based on their contrast score and removes redundant candidate root causes from the result list.

3.1 Critical Path Extraction

Traces may have complex structures consisting of invocation hierarchy and asynchronous/parallel invocations. As a result of this complexity, it is very hard to mine potential multi-dimensional root causes. The critical path is widely used for performance analysis in parallel and distributed computing [39] and has been applied to distributed tracing in recent studies [2, 30, 45]. A critical path describes the ordered list of steps that directly contribute to the slowest path of a request moving through a distributed system [2]. The impact of performance issues tends to reflect in the critical path. Therefore, we extract the critical path of each trace and represent it as an event sequence that incorporates both the structure of the trace and the attributes of each span.

3.1.1 Critical Path Algorithm. TraceContrast extracts the critical path of each trace based on the span kind and the causal relationship between spans. Existing approaches [30, 45] extract critical paths only based on the start/end time of the span. However, traces are often affected by the clock drift problem, which can lead to inaccurately extracting critical paths. We take the start and end of each span as different steps in the execution of a request, and the critical path is extracted as a sequence of steps. The proposed algorithm for critical path extraction is presented in Algorithm 1.

Algorithm 1 takes the root span of a trace as input and computes the critical path starting from its end. We sort all the child spans of the root span in descending order according to their end times. Subsequently, we select the last-ended child span and use it as input to invoke Algorithm 1. Then, we look for the next child span of the root span which is the last-ended before the start of the current span, and use it as input to invoke Algorithm 1. We perform the same process until there are no spans left. The process is recursive and stops until the input span has no child spans, or the input span type is Producer. We keep only the server span for a pair of client and server spans. Moreover, for a span with no child span, we merge the start and end into one step to make the critical path concise.

Figure 4 shows an example of the critical path extract from the trace depicted in Figure 1. Similarly, we exclude all client spans in the figure. Span A is the root span of this trace. This trace includes an asynchronous invocation (Span D) and a parallel invocation (Span E and Span F). The red segments in Figure 4 represent the execution process of the critical path of the trace. By applying Algorithm 1 to the trace, the critical path is represented as the step sequence $< A.start \to B.start \to C \to B.end \to E \to A.end >$. It can be seen that both asynchronous invocation (Span D) and one invocation in the parallel invocation (Span F) are not included in the critical path because they do not contribute to the end-to-end latency.

3.1.2 Critical Path Representation. To combine the attributes of a trace with the corresponding critical path for multi-dimensional root cause localization, we convert the step sequence of a critical path into a sequence of event sets. Each event set corresponds to

Algorithm 1 Critical Path Extraction

```
Require: Root Span $root$
Ensure: Critical Path $path$
 1: procedure CRITICALPATH($root$)
 2:     if $root.children$ is None or $root.kind$ is Producer then
 3:         return [$root$]
 4:     end if
 5:     if $root.kind$ is Client then
 6:         return CriticalPath($root.child$)
 7:     end if
 8:     $path \leftarrow [root.end]$
 9:     $children \leftarrow$ Descending sort $root.children$ by end time
10:     $lastChild \leftarrow children[0]$
11:     $path \leftarrow$ CriticalPath($lastChild$).extend($path$)
12:     for $span$ in $children[1:]$ do
13:         if $span.endTime < lastChild.startTime$ then
14:             $path \leftarrow$ CriticalPath($span$).extend($path$)
15:             $lastChild \leftarrow span$
16:         end if
17:     end for
18:     $path \leftarrow [root.start]$.extend($path$)
19:     return $path$
20: end procedure
```

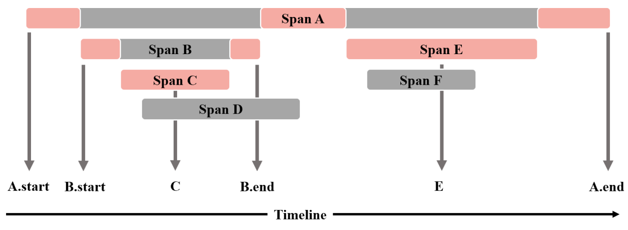


Figure 4: An Example of Critical Path

a step in the critical path and contains multiple events, with each event corresponding to an attribute value in the corresponding span.

Given a critical path, we convert each step to an event set in the following way. For each step, we obtain all the attributes of the corresponding span. Then, convert each attribute into discrete events based on their value type (nominal or numerical). For each nominal attribute whose value is given by a finite set of categories, e.g., service name, instance id, we treat each occurrence of a nominal value as a unique event and insert it into the event set. For each numerical attribute, such as request payload length, we obtain the attribute's values in all traces and discretize it into different intervals. Then, given a specific numerical attribute value, we replace it with its interval and treat it as a unique event. Specifically, we use Jenks' natural breaks [15] to find intervals in the values, which is a clustering method designed to determine the best arrangement of values into different classes. For example, the step E in the critical path in Figure 4 can be represent as the event set $(GET/v1/orders, ServiceC, ServiceC\text{-}2, node\text{-}1, 2000 < requestlength < 4000, \cdots)$. We perform this procedure for each step until the sequence has been processed. In particular, for a more concise representation of the critical path, for attributes that have the same meaning we keep only one of them, such as service id and service name.

3.2 Path Anomaly Detection

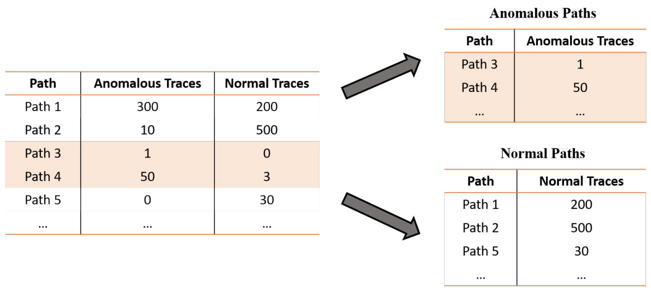
To perform spectrum analysis, we need to first identify which critical paths pass through the root causes. Existing approaches [21, 40] assume that all anomalous traces pass through the root causes, but this is inaccurate as a fault may propagate across the microservice system. For example, if a performance issue occurs in an API of a service, an anomaly may occur when other APIs in that service are invoked because of concurrent invocations. Our insight is that if the critical path of a trace passes through the root cause, it has a higher probability of being anomalous than other traces. Moreover, a microservice system has limited scales and provides limited functionality, the critical path for some traces is the same. Therefore, we first perform anomaly detection on each trace based on its critical path. Then the traces with the same critical path are aggregated into a group, and the proportion of anomalous traces in the group is calculated. If the group has a high percentage of anomalous traces, its corresponding critical path will be considered as anomalous.

First, We use $k - \sigma$ to detect whether a trace is anomalous based on its critical path. The end-to-end latency of traces with the same critical path in the operation level usually has the same distribution. It is because the end-to-end latency is usually determined by the operations invoked, independent of other attributes. Thus, we keep only the operation names in the critical path representations and aggregate traces with the same critical path into a group. Then, we calculate the mean μ and standard deviation σ of end-to-end latency of each group from a set of traces collected from a period time of normal execution. For each trace, we obtain the mean and standard deviation of the end-to-end latency of its critical path. The trace is detected as anomalous if its latency is higher than $\mu + k \times \sigma$, where k is the parameter used to adjust the threshold. For example, if the mean latency and the standard deviation of the normal execution of the critical path in Figure 4 is $20ms$ and $2ms$, then a new trace that has the same critical path and the latency higher than $20 + k \times 2$ ms will be detected as anomalous.

When a performance issue occurs, there can be critical paths that are not present in the normal execution of the system. To detect anomalous traces in these scenarios, we calculate the expected latency of each trace based on its critical path. As shown in Figure 4, the latency of each trace is the sum of the local execution time of each span in the critical path. Local execution time is the execution time of the current span with the waiting time of its child spans excluded [12, 43]. Thus, we calculate the expected local execution time for each span in the critical path and treat the sum of the expected local time as the expected latency of the trace. Inspired by previous work [14, 40], the excepted latency of each trace is calculated using Equation 1.

$$L_{path} = \sum_{span \in path} \mu_{span.op} + n \times \sigma_{span.op} \tag{1}$$

where $\mu_{span.op}$ is the mean of the local execution time of the corresponding operation in normal execution; $\sigma_{span.op}$ is the standard deviation of the local execution time of the corresponding operation in normal execution; n is used to adjust upper bound values.



Path	Anomalous Traces	Normal Traces
Path 1	300	200
Path 2	10	500
Path 3	1	0
Path 4	50	3
Path 5	0	30
...	...	...

Anomalous Paths

Path	Anomalous Traces
Path 3	1
Path 4	50
...	...

Normal Paths

Path	Normal Traces
Path 1	200
Path 2	500
Path 5	30
...	...

Figure 5: Example of Path Anomaly Detection

Based on the detected anomalous traces, we then detect the anomalous critical paths. We first aggregate traces with the same critical path into a group. Unlike trace anomaly detection, we use all possible attributes for root cause location in this aggregation. Only critical paths with the same attributes and order will be aggregated into a group. Then a critical path is determined as anomalous when there are at least α percent of traces (α is 70% in this paper) in its group are anomaly traces. As shown in Figure 5, there are only 60% traces that are anomalous in the group of Path 1, thus it will be determined as a normal path. Finally, we will output all anomalous and normal paths, and the number of traces corresponding to each path, as shown in the example in Figure 5. In this way, the number of paths that need to be processed is greatly decreased when performing contrast sequential pattern mining in the next step.

3.3 Contrast Sequential Pattern Mining

We frame the task of trace-based multi-dimensional root cause localization as a contrast sequential pattern (CSP) mining problem. Contrast sequential pattern mining problem aims to find patterns that occur frequently in one sequence dataset but not in others [16, 46]. It can discover the characteristics of different classes in sequence datasets and has been widely used in sequential data analysis, such as anomaly detection and customer behavior analysis [16, 46]. For the trace-based multi-dimensional root cause localization task, each pattern is a sequence consisting of attribute values, and the root cause is included in the patterns that occur frequently in anomaly paths but not in normal paths.

TraceContrast treats the anomaly paths and normal paths as two sequential databases that consist of critical paths, then uses eCSP algorithm [46] to mine contrast patterns. As a critical path is represented as a sequence of event sets, thus a pattern is a subsequence of this sequence. For example, $< ServiceA \longrightarrow ServiceB >$ and $< (ServiceA, API_1) >$ are some sub-sequences of the sequence $< (ServiceA, API_1) \longrightarrow (ServiceB, API_2) >$. It can be seen that the sub-sequences can represent entities in the system and their interactions at different levels of dimension which enables multi-dimensional root cause localization.

eCSP [46] is a tree-based algorithm similar to PrefixSpan, which implements a downward and depth-first search strategy on the tree to find all patterns. As shown in Figure 6, each node in the tree represents a pattern, and the parent node represents its prefix.

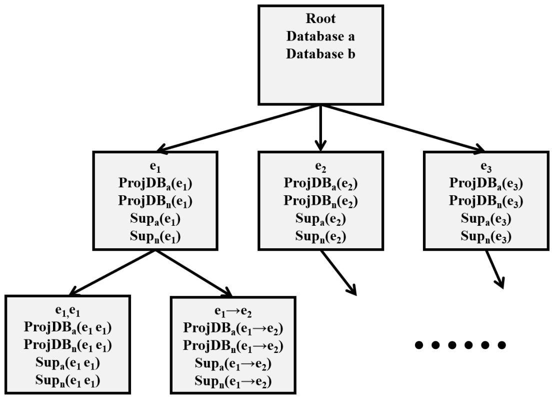


Figure 6: The Structure of the Tree in eCSP

And each node stores the information of a pattern, including the projected databases and the support of each database. The support of a pattern p in one sequential database, denoted as $Sup(p)$, is the number of traces whose critical path contains p divided by the number of traces in the sequential database. As we record the number of traces corresponding to each critical path, we can efficiently calculate the supports of each sub-sequence. eCSP will search the tree until the patterns reach the pre-defined maximum length. Then all the patterns of the nodes that have been searched are output as results. Readers can refer to [46] for more details of eCSP algorithm.

The number of critical paths and events in a large-scale microservice system is often enormous, resulting in the mining process can be inefficient. We improve the efficiency of eCSP in the following two aspects. First, we extend the eCSP algorithm to a parallel version by utilizing the tree structure. Specifically, We execute the eCSP algorithm until the projected database size of each node to be searched is less than a threshold. We then start multiple threads, and each thread independently searches different subtrees. We implemented the algorithm using Spark [11] to enable it to be used in any distributed computing clusters. On the other hand, we adopt several pruning methods to reduce the search space. Following [46], we employ the following pruning methods.

- **Events pruning**: Its removes events that appear only in the normal paths and not in the anomaly paths. This is because, in the root cause localization task, the root cause must include in the anomaly paths but not necessarily in the normal paths.
- **Chi-square pruning**: It prunes a pattern when its sample distribution similar to its prefix's. Specifically, TraceContrast calculates the chi-square statistic value between a pattern and its prefix. If the value is less than a predefined threshold δ, then the node will be pruned.
- **Minimum support pruning**: TraceContrast pruning a pattern when its support in the anomaly paths is lower than a threshold θ.

Readers can refer to [46] for more details of the last two pruning methods.

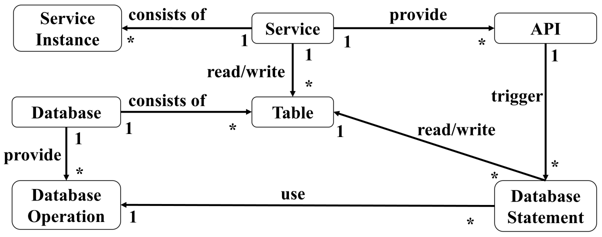


Figure 7: Relationship of Some Concepts in Microservice Systems

3.4 Candidate Root Cause Ranking

TraceContrast ranks patterns found by the eCSP algorithm based on the discriminative ability of each pattern. Previous works [16, 46] use some statistical metrics to calculate the contrast score of each pattern to measure its discriminative ability, such as growth rate, and support difference. However, they are not designed for the root cause localization task. Spectrum analysis-based fault localization has been widely used in program debugging [36], and several formulas have been proposed. TraceContrast uses a spectrum analysis formula to calculate the contrast score of each pattern. The redundant patterns are then removed based on microservice system domain knowledge and a ranked list of candidate root causes is returned.

3.4.1 *Contrast Score.*

TraceContrast uses the spectrum analysis formula to calculate the contrast score of each pattern. The intuition of the spectrum analysis formula is measuring which program element is covered by more failed test cases and less passed test cases. For multi-dimensional root cause localization, we consider patterns that are covered by more anomalous paths and less normal paths are more likely to be the root cause, which is similar to the intuition of the spectrum analysis. Specifically, TraceContrast uses the Ochiai ranking formula [1], which is defined as follows:

$$Score_s = \frac{e_f(s)}{\sqrt{(e_f(s) + n_f(s)) * (e_f(s) + e_p(s))}} \tag{2}$$

where s is a pattern in the result; $n_f(s)$ is the number of traces whose critical path doesn't contain pattern s in the anomaly database; $e_f(s)$ is the number of traces whose critical path contains pattern s in the anomaly database; $e_p(s)$ is the number of traces whose critical path contains pattern s in the normal database. Our approach also supports using other formulas to calculate contrast scores.

3.4.2 *Redundant Patterns Mitigation.*

Although TraceContrast finds and ranks patterns that possibly contain the root cause, it can sometimes contain redundant patterns that correspond to the same potential root cause. Operations engineers and developers want the tool to output different potential root causes as much as possible to speed up fault diagnosis. Thus we remove redundant patterns from the result to output a concise result.

We define the redundant pattern as the pattern whose discriminative ability comes from another pattern in the result list. Then, we identify redundant patterns based on the relationships between different patterns and the relationships between concepts in the microservice system. As shown in Figure 7, the concepts in the microservice system have a hierarchical structure. This leads to the fact that when a concept is affected by a fault, all of its underlying concepts are also affected. For example, when there is a bug in a service, all its instances are affected by that bug. In this paper, we use the conceptual hierarchy shown in Figure 7, which is applicable to most microservice systems. Our approach also supports the user to customize the concept hierarchy as required.

TraceContrast first identifies redundant patterns whose discriminative ability comes from its sub-pattern. Pattern p_1 is the sub-pattern of pattern p_2 when the corresponding sequence of p_1 is a sub-sequence of the corresponding sequence of p_2. If a pattern's score is less than or equal to the score of one of its sub-patterns, it will be identified as a redundant pattern. For example, if pattern $< ServiceA \rightarrow ServiceB >$ has a score of 0.9 and its sub-sequence $< ServiceA >$ has a score of 0.9, then $< ServiceA \rightarrow ServiceB >$ is identified as a redundant pattern.

Then, TraceContrast identifies redundant patterns whose discriminative ability comes from the high level concepts in the system. Based on the concept hierarchy, we define the concept sub-patterns of a pattern as the patterns that can be obtained by converting one or more events in the pattern or its sub-patterns to their high level concept events. For example, pattern $< ServiceA >$ is a concept sub-pattern of pattern $< API_2 \rightarrow ServiceB >$, where API_2 belongs to $ServiceA$. If a pattern's score is less than or equal to the score of one of its concept sub-patterns, it will be identified as a redundant pattern. For example, if pattern $< ServiceA >$ has a contrast score of 0.9, and pattern $< API_2 \rightarrow ServiceB >$ has a contrast score of 0.8, then $< API_2 \rightarrow ServiceB >$ is identified as a redundant pattern.

4 EVALUATION

To evaluate TraceContrast we conduct a series of experimental studies to investigate the following research questions:

- **RQ1**: How accurate is TraceContrast in root cause localization compared with baseline approaches?
- **RQ2**: How efficient is TraceContrast in root cause localization compared with baseline approaches? How well can it scale with the available computing resources?
- **RQ3**: How much do the different modules of TraceContrast contribute to the accuracy?
- **RQ4**: How sensitive is TraceContrast to the minimum support in contrast sequential pattern mining?

4.1 Experiment Setup

4.1.1 Dataset. Our experiments are conducted on a medium-scale open-source microservice benchmark system: TrainTicket [48, 51]. It provides typical train ticket booking functionalities and has been widely used in researches on microservice architecture, resource management, and fault diagnosis [17, 29, 30, 37, 40, 42, 43, 49]. It contains 47 services implemented in different languages and communicating with synchronous REST invocations and asynchronous messaging.

We deploy TrainTicket on a Kubernetes cluster with six virtual machines. Each virtual machine is equipped with an 8-core Intel Xeon 3.0GHz CPU, and 24GB RAM, and runs with CentOS 7.7. Some services are deployed as multiple instances, with a total of 90 service instances. We use OpenTelemetry [9] as the distributed tracing framework to collect traces and use Grafana Tempo [8] as the distributed tracing backend.

Because the datasets from existing work [21, 26, 40] only contain faults at the instance/service dimension, and cannot well evaluate multi-dimensional root cause localization. We use fault injection to simulate different dimensions performance issues for our experimental studies following previous works [17, 37, 40, 42, 43]. We use the widely used chaos engineering tool ChaosBlade [6] to inject different faults. In particular, we adopt 5 fault types and inject faults at different dimensions to simulate performance issues on multi-dimensional. Table 1 shows the details of the fault injection strategies in our dataset. We label the root cause of each performance issue according to the fault types and injection targets. For example, we inject the slow SQL fault at the service (all service instances belong to this service) and service instance levels, and label the root cause of the fault as the combination of the slow SQL statement/command and the service/service instance.

We use the load generator provided by TrainTicket [48, 51] to execute automated test cases to simulate different user requests to generate traces. For each performance issue, we inject one fault at a time and make it last for six minutes. In total, we collected a total of 175 performance issues along with 4,331,882 traces. The longest trace in our dataset contains more than 400 spans; each span contains at least 5 attributes and at most 18 attributes. Compared to existing work, we constructed the dataset using a higher concurrency request number, which better reflects the impact of fault propagation and results in more traces in each performance issue. In addition, we create an initialization dataset, which includes 322,828 traces collected from a normal execution of TrainTicket in 2 hours. The initialization dataset is used for the initialization of TraceContrast and also the initialization or training of all the baseline approaches.

4.1.2 Baseline Approaches. We compared TraceContrast with the following four state-of-the-art trace-based unsupervised root cause localization approaches.

- **CRISP** [45]: It represents traces as critical paths. Then it uses the same model and lightweight heuristic method as TraceAnomaly [26] to locate root cause instances.
- **MicroRank** [40]: It uses a PageRank-based spectrum analysis method to locate the root cause instances.
- **TraceRCA** [21]: It uses frequent itemset mining and spectrum analysis to locate the root cause instances.
- **Minesweeper** [28]: It combines sequential pattern mining and statistical isolation measures to locate the root causes of crash reports. It represents user behavior as sequences of events to locate root causes, but it does not support multi-dimensional root cause localization. We extend it to support multi-dimensional root cause localization and feed it the same anomalous paths and normal paths as TraceContrast to locate multi-dimensional root causes.

Table 1: Fault Injection Strategies in the TrainTicket Dataset

Fault Type	Description	Injection Targets
CPU Exhausted	The CPU of a container is exhausted, resulting in an increase in the processing time of invocations.	Service, Service Instance
Network Delay	A network jam occurs, resulting in an increase in the time spent on network communication.	Service, Service Instance, Service Invocation Pair, Service Instance Invocation Pair
API Delay	Implementation bugs of an API or traffic scheduling issues, resulting in an increase in processing time when invoking that API.	Service, Service Instance
Slow SQL	A bug in table structure design or SQL statement implementation, resulting in an increase in processing time for a type of SQL command or a SQL statement.	Service, Service Instance
Producer Delay	Excessive concurrent requests to the message queue, resulting in an increase in the time to produce a message.	Service, Service Instance

4.1.3 Evaluation Metrics. We use the top-k hit ratio (HR@k) and mean reciprocal rank (MRR) to evaluate the accuracy of multi-dimensional root cause localization following existing works [25, 37, 40].

- **HR@k** represents the probability that the root cause is included in the top-k result list ($k = 1, 3, 5$ in this paper).
- **MRR** is the multiplicative inverse of the rank of the root cause in the result list. If the root cause is not included in the top-10 result list, the rank can be regarded as positive infinity [25]. Given a set of fault instances A, $Rank_i$ is the rank of the root cause in the returned list of the ith fault instance, MRR is calculated by the following equation:

$$MRR = \frac{1}{|A|}\sum_{i=1}^{|A|}\frac{1}{Rank_i} \tag{3}$$

For a fair comparison with the baseline approaches, we also evaluate the accuracy of instance dimensional root cause localization of all approaches. For each performance issue, we regard all service instances which are injected with fault as the root cause. When evaluating the accuracy of instance dimensional root cause localization, we regard it as a hit only when all service instances are included in the result list. For TraceContrast and Minesweeper, which are multi-dimensional root cause location approaches, we regard it as hitting all instances of a service when the result list contains the service.

4.1.4 Implementation and Settings. We implement TraceContrast in Python 3.8, Scala 2.12, and Spark 3.2.3 [11] (for contrast sequential pattern mining). With Spark, TraceContrast can be easily deployed in a large-scale computing cluster, supporting usage in industrial-scale systems. The settings of TraceContrast are the following: the threshold ϵ in chi-square pruning is 3.84; the minimum support θ is 0.5. We set k in $k - \sigma$ as 3 and n in Equation 1 is 3 following previous studies. In practice, these two parameters can be decided based on the validation set. The conceptual hierarchy used for redundant pattern mitigation is shown in Figure 7. All the experimental studies are conducted on a Linux server with two AMD EPYC 7T83 64-Core Processor CPU, 512GB RAM, RTX 3090 with 24GB GPU memory and running Ubuntu 20.04.5.

4.2 Accuracy

Table 2 and Table 3 show the results of TraceContrast and the baseline approaches for multi-dimensional and instance dimensional root cause localization respectively. It can be seen that TraceContrast outperforms all baseline approaches in both multi-dimensional and instance-dimensional root cause localization and achieves 0.629 and 0.829 in term of HR@5 respectively.

Table 2: Accuracy of Different Approaches at Multi Dimensional

Approach	HR@1	HR@3	HR@5	MRR
CRISP	0.217	0.240	0.257	0.229
MicroRank	0.154	0.183	0.229	0.178
TraceRCA	0.223	0.234	0.234	0.227
Minesweeper	0	0.120	0.160	0.063
TraceContrast	**0.497**	**0.589**	**0.629**	**0.554**

Table 3: Accuracy of Different Approaches at Instance Dimension

Approach	HR@1	HR@3	HR@5	MRR
CRISP	0.326	0.429	0.469	0.385
MicroRank	0.274	0.429	0.589	0.394
TraceRCA	0.417	0.463	0.463	0.442
Minesweeper	0.011	0.200	0.269	0.115
TraceContrast	**0.629**	**0.777**	**0.829**	**0.706**

For multi-dimensional root cause localization, TraceContrast outperforms the baseline approaches by 144.7%-293.1% in term of HR@5. MicroRank, TaceRCA, and CRISP show poor performance in multi-dimensional root cause localization as they can only localize root causes at the instance dimension. While Minesweeper supports localizing multi-dimensional root causes, its accuracy is much lower than the other methods. It is because Minesweeper performs frequent sequence pattern mining on normal and abnormal traces respectively, which can make it difficult to find patterns with high discriminative ability. On the other hand, it aggregates candidate root causes with similar scores into a group and only the longest root cause is retained in each group, resulting in many candidate root causes being incorrectly identified as redundant. TraceContrast performs much better than the baseline approaches, as it can find contrast patterns from the traces and combine domain knowledge to identify redundant patterns.

Table 4: Average Execution Time of Different Approaches

Approach	Time Consuming
CRISP	8.7s
MicroRank	250.9s
TraceRCA	43.2s
Minesweeper	1,088.9s
TraceContrast	**209.6s**

For instance dimensional root cause localization, TraceContrast outperforms the baseline approaches by 40.7%-208.1% in term of HR@5. This is because MicroRank and TaceRCA perform spectrum analysis based on the original traces rather than the critical paths, which can lead to the mislocalization of instances involving parallel and asynchronous invocations. Although CRISP uses the critical path, it simply treats the root cause as the instance corresponding to the last anomaly span in the critical path. This makes CRISP difficult to diagnose performance issues where the root cause includes multiple instances.

In conclusion, TraceContrast is effective in multi-dimensional and instance-dimensional root cause localization. And TraceContrast outperforms baseline approaches by 319.1% and 184.1% on average in term of MRR in multi-dimensional and instance-dimensional root cause localization respectively.

4.3 Efficiency and Scalability

To evaluate the efficiency of TraceContrast and the baseline approaches, we compare the average execution time of root cause localization of TraceContrast and the baseline approaches. We did not compare the execution time of trace anomaly detection because it is less time-consuming and most approaches can use any trace anomaly detection method.

Table 4 shows the average execution time of TraceContrast and baseline approaches. On average, these approaches take 8.7-1,088.9 seconds to locate the root cause of a performance issue. In general, all approaches are efficient, except the Minesweeper which average cost 1,088.9 seconds to locate the root cause of a performance issue. TraceContrast is slower than CRISP and TraceRCA but faster than Microrank and Minesweeper.

CRISP is much faster than other approaches because it uses a lightweight heuristic method. TraceRCA is slightly slower than CRISP because it mines frequent invocation pairs. TraceContrast and Minesweeper are slower than TraceRCA as sequential pattern mining is more time-consuming. Microrank is slower than TraceContrast because the PageRank algorithm takes much time when the graph size is large. Although we have optimized Minesweeper with parallel execution, it still consumes more time than TraceContrast. It is because it mines potential root causes in anomaly and normal traces respectively, without using any effective pruning strategies.

To evaluate the scalability of TraceContrast, we calculate the average execution time of TraceContrast with different computational resources from 8 cores to 128 cores. The results in Figure 8 show that the average execution time of TraceContrast decreases from 940.5 to 209.6 seconds with increasing computational resources. It can be seen that parallel execution can effectively improve the efficiency of contrast sequential pattern mining. However, the reduction in execution time decreases as the computational resources increase, because the bottleneck in execution time may exist in a small number of subtrees. We implement TraceContrast based on Spark, which can effectively utilize big data computing clusters and can be effectively applied to large-scale microservices systems.

In conclusion, TraceContrast achieves an average execution time of 209.6s, which is acceptable for root cause localization. Furthermore, the efficiency of TraceContrast can be further improved by parallel execution with more computing resources.

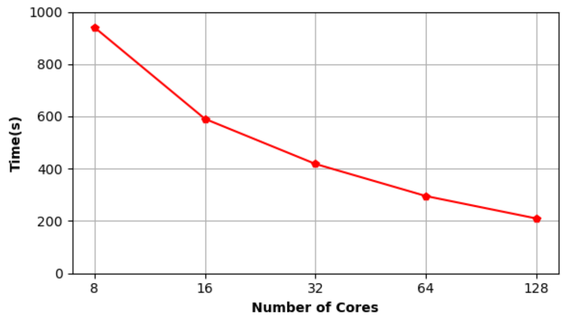


Figure 8: Changes of Execution Time with the Increase of CPU Cores

Table 5: Evaluation of Contribution of Anomaly Path Detection and Redundant Patterns Mitigation at Multi Dimension

Approach	HR@1	HR@3	HR@5	MRR
TraceContrast w/oPAD	0.429	0.480	0.491	0.458
TraceContrast w/oRPM	0.463	0.549	0.594	0.523
TraceContrast w/oALL	0.383	0.469	0.497	0.433
TraceContrast	**0.497**	**0.589**	**0.629**	**0.554**

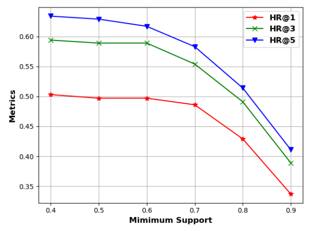
4.4 Ablation Study

We perform an ablation study to evaluate how different modules of TraceContrast contribute to the accuracy of root cause localization. Path anomaly detection and redundant pattern mitigation are two important modules that affect the accuracy of TraceContrast. We derive the following three variants of TraceContrast: **TraceContrast w/oPAD** removes the path anomaly detection module and uses the original anomaly and normal traces as the input of contrast sequential pattern mining directly. **TraceContrast w/oRPM** removes the redundant pattern mitigation module and outputs the original rank list. **TraceContrast w/oALL** removes both of the above modules.

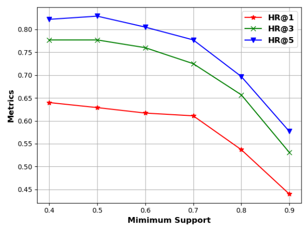
Table 5 and Table 6 show the evaluation results of the contribution of the two steps at multi-dimensional and instance-dimensional root cause localization respectively. In terms of MRR, TraceContrast outperforms the three variants by 54.8% and 68.9% on average in multi-dimensional and instance-dimensional root cause localization. This is because the path anomaly detection module removes

Table 6: Evaluation of Contribution of Anomaly Path Detection and Redundant Patterns Mitigation at Instance Dimension

Approach	HR@1	HR@3	HR@5	MRR
TraceContrast w/oPAD	0.509	0.640	0.680	0.570
TraceContrast w/oRPM	0.600	0.680	0.691	0.646
TraceContrast w/oALL	0.480	0.554	0.571	0.520
TraceContrast	**0.629**	**0.777**	**0.829**	**0.706**



(a) Multi Dimension



(b) Instance Dimension

Figure 9: Impact of Minimum Support on Hit Ratio of Root Cause Localization

critical paths which are not pass through the root cause from the anomalous paths, thus preventing some patterns that affect by fault propagation from getting into the final rank list. The redundant pattern mitigation module effectively removes redundant patterns and makes the rank list more informative.

In conclusion, path anomaly detection and redundant pattern mitigation can effectively improve the accuracy of TraceContrast for both multi-dimensional and instance-dimensional root cause localization. It also illustrates the rationality of the design of TraceContrast.

4.5 Sensitivity to Minimum Support

We investigate the impact of the minimum support in TraceContrast. Minimum support influences the number of candidate root causes found by TraceContrast and the execution time. We test the hit ratio and execution time of TraceContrast with different values of minimum support from 0.4 to 0.9 and keep all the other configurations unchanged.

Figure 9 and Figure 10 show the impact of the minimum support on the hit ratio and execution time respectively. It shows that with the increase of the minimum support the hit ratio declines and the execution time increases. It is because that smaller support makes TraceContrast search more nodes in the tree in eCSP, thus finding more candidate root causes. However, when the minimum support is less than 0.6, with the decrease of the minimum support the execution time increases significantly, but the hit rate increase slowly. Therefore, the trade-off between accuracy and execution time should be considered when choosing the minimum support.

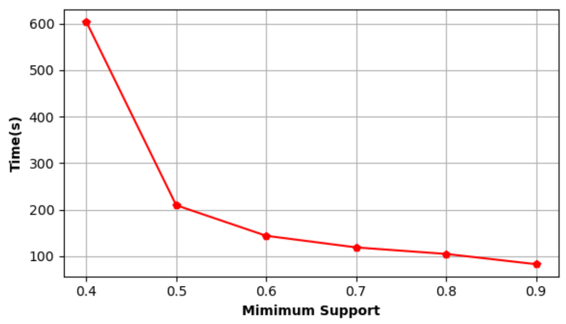


Figure 10: Impact of Minimum Support on Execution Time of Root Cause Localization

4.6 Threats to Validity

The internal threats in our studies are mainly from the implementation and configuration of baseline approaches. MicroRank and TraceRCA have provided publicly available implementations, thus we use them directly. CRISP is based on TraceAnomaly [26], thus we implement CRISP based on TraceAnomaly's publicly available implementation [33]. Although Minesweeper has no publicly available implementations, it is based on the standard sequential pattern mining algorithm PrefixSpan. We implement it following its paper and use the standard PrefixSpan algorithm provided by Spark to reduce the threat. In terms of the impact of configuration, we chosse the best configuration for all baseline approaches through experimentation.

The external threats in our studies are mainly from the representativeness of benchmark system and performance issues. We have only conducted experiments on TrainTicket as there are no publicly available datasets from industrial systems. Although it is one of the largest open-source microservices systems and is widely used in existing studies [21, 26, 42, 44, 45, 49], there is still a gap between its scale and industrial large-scale microservices systems. Moreover, although we use more fault types than existing studies [21, 26, 40], these faults still can hardly cover all fault types in industrial large-scale microservices systems. Therefore, the results of our experimental studies may not be generalized to larger or more complex systems or performance issues.

5 RELATED WORK

With the development of distributed tracing and its infrastructure, researchers have investigated distributed tracing-based fault diagnosis techniques. Some existing studies [19, 27, 48] have analyzed the role of distributed tracing in the fault diagnosis of large-scale microservices systems, which shows that automated trace analysis is critical for fault diagnosis in large-scale microservices systems.

Recently, researchers have proposed some trace-based root cause localization approaches based on machine learning techniques. Zhou et al. [49] propose MEPFL, a supervised learning approach that uses a set of predefined features to represent each trace, It trains three models to predict anomalous traces, anomalous services, and anomaly types respectively. Gan et al. [4] propose Seer, which uses supervised learning to train a CNN and LSTM based model to locate root cause service. These supervised approaches use fault injection to generate labeled traces, but fault injection is difficult to cover all types of faults. This makes it difficult to apply these approaches in industrial systems. Liu et al. [26] propose an unsupervised approach called TraceAnomaly, which trains deep Bayesian networks model to detect anomalous traces and uses a lightweight heuristic method to locate anomalous service instances. Zhang et al. [45] further extend TraceAnomaly by representing traces as critical paths. Gan et al. [3] propose Sage, which is based on graph neural networks and counterfactual inference for unsupervised root cause service instance localization. However, machine learning-based approaches face the problem of gradual performance degradation due to concept drift, resulting in the requirement to periodically retrain the model to alleviate this problem.

Some researchers extend spectrum analysis techniques to achieve more practical approaches for trace-based microservice root cause localization. Yu et al. [40] propose MicroRank, which uses a PageRank-based spectrum analysis method to locate root cause service instances. Then they propose TraceRank [41], which extends MicroRank by combining spectrum analysis with random walk-based method. Li et al. [21] combine spectrum analysis with frequent pattern mining to locate root cause service instances. These approaches don't rely on training data and are easy to implement, which makes them more acceptable to industry systems. However, they all focus on root cause localization at the service/instance level, ignoring the fact that multi-dimensional root causes are common in microservice systems.

Multi-dimensional root cause localization is important in the cloud and large-scale online systems, which has attracted the attention of many researchers [5, 20, 22–24, 35, 38]. These approaches represent telemetry data as multi-dimensional tabular data and search for the attribute value combination where the anomalies are most concentrated as the root cause. CMMD [38], Squeeze [22], and ImAPTr [35] transform metrics data into multi-dimensional tabular data based on the attributes in raw data. MID [5] and iDice [24] represent issue reports as multi-dimensional tabular data by extracting attributes from issue reports. Lin et al. [23] represent logs as multi-dimensional tabular data by extracting pre-defined features from structured logs, then use FP-Growth to find frequent combinations of feature values as the root cause. These approaches are designed for multidimensional tabular data and cannot well handle data with complex structures, such as traces. Therefore, these approaches cannot well support the multi-dimensional root cause localization of microservice systems.

6 CONCLUSION

In this paper, we proposed a trace-based multi-dimensional root cause localization approach for performance issues of microservice systems, called TraceContrast. It represents traces as event sequences which combine the complex structure of traces and attributes of each span. Based on the representation, TraceContrast combines contrast sequential pattern mining and spectrum analysis to localize multi-dimensional root causes. Experiments on a medium-scale microservice benchmark system show that TraceContrast outperforms existing approaches in both multi-dimensional and instance-dimensional root cause localization. And TraceContrast is efficient and its efficiency can be further improved by parallel execution.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006)*. IEEE Computer Society, 39–46. https://doi.org/10.1109/PRDC.2006.18

[2] Brian Eaton, Jeff Sterart, Jon Tedesco, and N. Cihan Tas. 2022. Distributed Latency Profiling through Critical Path Tracing: CPT can provide actionable and precise latency analysis. *ACM Queue* 20, 1 (2022), 40–79. https://doi.org/10.1145/3526967

[3] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: practical and scalable ML-driven performance debugging in microservices. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 135–151. https://doi.org/10.1145/3445814.3446700

[4] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019*. ACM, 19–33. https://doi.org/10.1145/3297858.3304004

[5] Jiazhen Gu, Chuan Luo, Si Qin, Bo Qiao, Qingwei Lin, Hongyu Zhang, Ze Li, Yingnong Dang, Shaowei Cai, Wei Wu, Yangfan Zhou, Murali Chintalapati, and Dongmei Zhang. 2020. Efficient incident identification from multi-dimensional issue reports via meta-heuristic search. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020*. ACM, 292–303. https://doi.org/10.1145/3368089.3409741

[6] https://chaosblade.io/. 2023. ChaosBlade. Retrieved July 15, 2023 from https://chaosblade.io/

[7] https://grafana.com/. 2023. Grafana Loki. Retrieved July 15, 2023 from https://grafana.com/oss/loki/

[8] https://grafana.com/. 2023. Grafana Tempo. Retrieved July 15, 2023 from https://grafana.com/oss/tempo/

[9] https://opentelemetry.io/. 2023. OpenTelemetry. Retrieved July 15, 2023 from https://opentelemetry.io/

[10] https://prometheus.io/. 2023. Prometheus. Retrieved July 15, 2023 from https://prometheus.io/

[11] https://spark.apache.org/. 2023. Apache Spark. Retrieved July 15, 2023 from https://spark.apache.org/

[12] Lexiang Huang and Timothy Zhu. 2021. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*. ACM, 76–91. https://doi.org/10.1145/3472883.3486994

[13] Jaegertracing.Io. 2023. Jaeger. Retrieved July 15, 2023 from https://www.jaegertracing.io/

[14] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. 2017. Performance Monitoring and Root Cause Analysis for Cloud-hosted Web Applications. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017*. ACM, 469–478. https://doi.org/10.1145/3038912.3052649

[15] George F Jenks. 1967. The data model concept in statistical mapping. *International yearbook of cartography* 7 (1967), 186–190.

[16] Xiaonan Ji, James Bailey, and Guozhu Dong. 2005. Mining Minimal Distinguishing Subsequence Patterns with Gap Constraints. In *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM 2005)*. IEEE Computer Society, 194–201. https://doi.org/10.1109/ICDM.2005.96

[17] Cheryl Lee, Tianyi Yang, Zhuangbin Chen, Yuxin Su, and Michael R. Lyu. 2023. Eadro: An End-to-End Troubleshooting Framework for Microservices on Multi-source Data. In *45th IEEE/ACM 45th International Conference on Software Engineering, ICSE 2023*. https://doi.org/10.48550/arXiv.2302.05092

[18] James Lewis and Martin Fowler. 2014. Microservices a definition of this new architectural term. Retrieved July 25, 2023 from https://www.martinfowler.com/articles/microservices.html

[19] Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. 2022. Enjoy your observability: an industrial survey of microservice tracing and analysis. *Empir. Softw. Eng.* 27, 1 (2022), 25. https://doi.org/10.1007/s10664-021-10063-9

[20] Liqun Li, Xu Zhang, Shilin He, Yu Kang, Hongyu Zhang, Minghua Ma, Yingnong Dang, Zhangwei Xu, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. 2023. CONAN: Diagnosing Batch Failures for Cloud Systems. In *45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP@ICSE 2023*. IEEE, 138–149. https://doi.org/10.1109/ICSE-SEIP58684.2023.00018

[21] Zeyan Li, Junjie Chen, Rui Jiao, Nengwen Zhao, Zhijun Wang, Shuwei Zhang, Yanjun Wu, Long Jiang, Leiqin Yan, Zikai Wang, Zhekang Chen, Wenchi Zhang,

Xiaohui Nie, Kaixin Sui, and Dan Pei. 2021. Practical Root Cause Localization for Microservice Systems via Trace Analysis. In *29th IEEE/ACM International Symposium on Quality of Service, IWQOS 2021*. IEEE, 1–10. https://doi.org/10.1109/IWQOS52092.2021.9521340

[22] Zeyan Li, Dan Pei, Chengyang Luo, Yiwei Zhao, Yongqian Sun, Kaixin Sui, Xiping Wang, Dapeng Liu, Xing Jin, and Qi Wang. 2019. Generic and Robust Localization of Multi-dimensional Root Causes. In *30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019*. IEEE, 47–57. https://doi.org/10.1109/ISSRE.2019.00015

[23] Fan Fred Lin, Keyur Muzumdar, Nikolay Pavlovich Laptev, Mihai-Valentin Curelea, Seunghak Lee, and Sriram Sankar. 2020. Fast Dimensional Analysis for Root Cause Investigation in a Large-Scale Service Environment. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 2 (2020), 31:1–31:23. https://doi.org/10.1145/3392149

[24] Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, and Dongmei Zhang. 2016. iDice: problem identification for emerging issues. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016,*. ACM, 214–224. https://doi.org/10.1145/2884781.2884795

[25] Dewei Liu, Chuan He, Xin Peng, Fan Lin, Chenxi Zhang, Shengfang Gong, Ziang Li, Jiayu Ou, and Zheshun Wu. 2021. MicroHECL: High-Efficient Root Cause Localization in Large-Scale Microservice Systems. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021*. IEEE, 338–347. https://doi.org/10.1109/ICSE-SEIP52600.2021.00043

[26] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, and Dan Pei. 2020. Unsupervised Detection of Microservice Trace Anomalies through Service-Level Deep Bayesian Networks. In *31st IEEE International Symposium on Software Reliability Engineering, ISSRE 2020*. IEEE, 48–58. https://doi.org/10.1109/ISSRE5003.2020.00014

[27] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *SoCC '21: ACM Symposium on Cloud Computing*. ACM, 412–426. https://doi.org/10.1145/3472883.3487003

[28] Vijayaraghavan Murali, Edward Yao, Umang Mathur, and Satish Chandra. 2021. Scalable Statistical Root Cause Analysis on App Telemetry. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021*. IEEE, 288–297. https://doi.org/10.1109/ICSE-SEIP52600.2021.00038

[29] Xin Peng, Chenxi Zhang, Zhongyuan Zhao, Akasaka Isami, Xiaofeng Guo, and Yunna Cui. 2022. Trace analysis based microservice architecture measurement. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022*. ACM, 1589–1599. https://doi.org/10.1145/3540250.3558951

[30] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020*. USENIX Association, 805–825. https://www.usenix.org/conference/osdi20/presentation/qiu

[31] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure.

[32] skywalking.apache.org. 2023. Apache SkyWalking. Retrieved July 15, 2023 from http://skywalking.apache.org/

[33] TraceAnomaly. 2023. TraceAnomaly. Retrieved July 15, 2023 from https://github.com/NetManAIOps/TraceAnomaly

[34] Twitter. 2023. Zipkin. Retrieved July 15, 2023 from https://zipkin.io/

[35] Hao Wang, Guoping Rong, Yangchen Xu, and Yong You. 2020. ImpAPTr: A Tool For Identifying The Clues To Online Service Anomalies. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*. IEEE, 1307–1311. https://doi.org/10.1145/3324884.3415301

[36] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Trans. Software Eng.* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[37] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2020. MicroRCA: Root Cause Localization of Performance Issues in Microservices. In *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium*. IEEE, 1–9. https://doi.org/10.1109/NOMS47738.2020.9110353

[38] Shifu Yan, Caihua Shan, Wenyi Yang, Bixiong Xu, Dongsheng Li, Lili Qiu, Jie Tong, and Qi Zhang. 2022. CMMD: Cross-Metric Multi-Dimensional Root Cause Analysis. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, 2022*. ACM, 4310–4320. https://doi.org/10.1145/3534678.3539109

[39] Cui-Qing Yang and Barton P. Miller. 1988. Critical Path Analysis for the Execution of Parallel and Distributed Programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems, 1988*. IEEE Computer Society, 366–373. https://doi.org/10.1109/DCS.1988.12538

[40] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. 2021. MicroRank: End-to-End Latency Issue Localization with Extended Spectrum Analysis in Microservice Environments. In *WWW '21: The Web Conference 2021*. ACM / IW3C2, 3087–3098. https://doi.org/10.1145/3442381.3449905

[41] Guangba Yu, Zicheng Huang, and Pengfei Chen. 2021. TraceRank: Abnormal service localization with dis-aggregated end-to-end tracing data in cloud native systems. *Journal of Software: Evolution and Process* (2021), e2413. https://doi.org/10.1002/smr.2413

[42] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. 2022. DeepTraLog: Trace-Log Combined Microservice Anomaly Detection through Graph-based Deep Learning. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022*. ACM, 623–634. https://doi.org/10.1145/3510003.3510180

[43] Chenxi Zhang, Xin Peng, Tong Zhou, Chaofeng Sha, Zhenghui Yan, Yiru Chen, and Hong Yang. 2022. TraceCRL: contrastive representation learning for microservice trace analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022*. ACM, 1221–1232. https://doi.org/10.1145/3540250.3549146

[44] Ke Zhang, Chenxi Zhang, Xin Peng, and Chaofeng Sha. 2022. PUTraceAD: Trace Anomaly Detection with Partial Labels based on GNN and PU Learning. In *IEEE 33rd International Symposium on Software Reliability Engineering, ISSRE 2022*. IEEE, 239–250. https://doi.org/10.1109/ISSRE55969.2022.00032

[45] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022*. USENIX Association, 655–672. https://www.usenix.org/conference/atc22/presentation/zhang-zhizhou

[46] Zhigang Zheng, Wei Wei, Chunming Liu, Wei Cao, Longbing Cao, and Maninder Bhatia. 2016. An effective contrast sequential pattern mining approach to taxpayer behavior analysis. *World Wide Web* 19, 4 (2016), 633–651. https://doi.org/10.1007/s11280-015-0350-4

[47] Tong Zhou, Chenxi Zhang, Xin Peng, Zhenghui Yan, Pairui Li, Jianming Liang, Haibing Zheng, Wujie Zheng, and Yuetang Deng. 2023. TraceStream: Anomalous Service Localization based on Trace Stream Clustering with Online Feedback. In *34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023*. IEEE, 601–611. https://doi.org/10.1109/ISSRE59848.2023.00033

[48] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2021. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Trans. Software Eng.* 47, 2 (2021), 243–260. https://doi.org/10.1109/TSE.2018.2887384

[49] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *2019 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019*. 683–694. https://doi.org/10.1145/3338906.3338961

[50] 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 2018*. ACM, 802–807. https://doi.org/10.1145/3238147.3240730

[51] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. Benchmarking microservice systems for software engineering research. In *40th International Conference on Software Engineering, ICSE 2018*. ACM, 323–324. https://doi.org/10.1145/3183440.3194991