STIGs: Spatio-Temporal Graphs for Expressing Microservice Interference

Iqra Zafar [1] · Christian M. Adriano [1] · Finn Kaiser [2] · Holger Giese [1]

Hasso-Plattner Institute,University of Potsdam, Potsdam, Germany

✉ [1] firstname.lastname@hpi.de, [2] finn.kaiser@student.hpi.uni-potsdam.de

Abstract. Microservices have gained significant popularity as a solution for developing large-scale applications in cloud environments, offering agility for evolving services and scalability for satisfying surges in user requests. However, in multi-tenant platforms, services often compete for resources, leading to a phenomenon called interference, which can affect multiple services and systems. Because interference propagates in complex anomaly patterns, it can bias (confounds) the diagnostics produced by methods like anomaly detection (*AD*) and root-cause analysis (*RCA*). Current interference mitigation (*IM*) methods are effective to measure performance loss between competing services, but they are limited to pairs of services. Hence, these methods do not contemplate the propagation effects through multiple services and across systems. As a first step to address this limitation, we formalize a new model (Spatio-Temporal Interference Graph - STIG) and a corresponding algorithm to instantiate this model from execution trace data. We demonstrate our approach on two publicly available microservice application benchmarks.

1 Introduction

Microservice architectures enable agility in evolving services to contemplate new features and scalability to satisfy surges in user requests [5]. However, stability of these systems can be a challenge, because of their interconnectivity, local events can propagate as complex anomalies patterns across and beyond a single system. For that, several methods exist for anomaly detection (*AD*) and root-cause analysis (*RCA*)[8]. The *AD* methods primarily concentrate on distinguishing normal behavior from abnormal behavior, while *RCA* methods aim to identify the affected services as well as the root cause of the anomaly. This enables the identification and resolution of the anomalies, effectively preventing their recurrence. It is important to note that addressing an anomaly does not always require making code changes and ensuring the passing of tests. Instead, fixes often involve changes to the configuration and deployment settings [2], such as modifying resource allocation (compute, memory, I/O) or adjusting the placement of services (clustering, containerization). Because *AD* is not concerned with root-causes, the corresponding methods often rely on coarse-grained metrics and require less information about the interdependencies among services [3]. On the other hand, *RCA* methods rely more heavily on dependency models to uncover the source of the anomaly [10, 4]. Despite the similarities and differences between *AD* and *RCA* methods, their outcomes can be biased (confounded) by the phenomenon of microservice interference (definition 1). The *AD* and *RCA*

methods typically focus on scenarios within a single application, where the stable call-graph dependencies and the determinism of their sequential executions limit the risk of interference emerging from resource competition. However, in multi-tenant platforms, this assumption no longer holds [9]. To address this, various interference mitigation (*IM*) methods have been developed - originally, for virtualized cloud environments [7] and, later, for microservices [6, 1].

Definition 1 ***Interference*** *happens when two services that have no logical dependency (caller-callee relation) compete for the same resource (compute, memory, I/O) to the extent that they affect each other's performance (e.g., throughput, latency)[7].*

Nonetheless, to the best of our knowledge, current *IM* methods face two main limitations that prevent them from effectively mitigating confounding in *AD* and *RCA* methods: (1) *IM* methods cover few number of services (four as in [1, 11]) and (2) they rely solely on metrics while not considering the interdependencies between services (call-graphs). The current interference metrics consist of *contention* (service usage pressure on a resource, e.g., CPU) and *sensitivity* (susceptibility of a service being impacted by other services) [7]. While *contention* is scale independent, *sensitivity* is measured for pairs of services [11, 1]. This limited focus on pairs hinders the measurement of the multiple sources and targets of interference. To capture interference across numerous microservices from various applications, we have proposed an new approach that formulates the interference phenomenon as a graph problem, more specifically as a Spatio-Temporal Graph (definition 2).

Definition 2 ***Spatio-Temporal Interference Graph (STIG)*** *is denoted as* $\mathcal{G} = (V, E, X_{v(t)}, X_{e(t)})$*, where* V *are nodes representing individual services,* E *are the directed edges representing interference between any two services from distinct applications,* $X_{v(t)}$ *are the time-varying features of nodes (e.g., resource per service), and* $X_{e(t)}$ *the features of the edges (e.g., probability of interference).*

As an example of **interference scenario**, assume two e-commerce applications having 14 microservices (shown in fig. 2) with shared common services (labeled *multi-service*) deployed on the same server (either *host1*, *host2*) each with a CPU of 4 cores and 10 GB of memory. The occurrence of a sudden surge of 200% in users during a flash sales event could subsequently cause an increase of the demand for these applications, e.g., from 40% to 90% CPU and memory usage from 4GB to 8GB. As the services compete for shared resources, the load increase could induce a low response time, e.g., 800ms from originally 200ms among the resource-sharing services. This, in turn, could evolve to more severe problems like intermittent or permanent failures. Because anomalies cross the applications' borders, one cannot rely on the individual call-graphs and performance metrics. This limitation is addressed by the STIG model, which captures dependencies originated both from the call-graph and the deployment-graph (e.g., service placement configuration).

Our contributions encompass (1) an **algorithm** for building STIGs from execution trace data and (2) a practical **demonstration** based on two microservice benchmarks (SockShop[1] and TeaStore[2]), which we combined into a single multi-tenant system.

2 Approach

In the following section, we explain our 4-step approach to generating STIGs (Figure 1). Foremost, we take the call-graphs of the two e-commerce shops as input and label the services based on their

[1] SockShop: https://microservices-demo.github.io/
[2] Tea-Store: https://github.com/DaGeRe/TeaStore

generic functionality to produce a typed-graph (**step 1**). If a service is present in both shops with the same type, we consider it as a replica service (**step 2**), i.e., both shops supposedly sharing that service. This allows us to simulate the complex interdependencies typical of multi-tenant systems. Next, to generate a hypothetical deployment configuration in a clustered environment (**step 3**), we apply three different service placement criteria: (1) *frequent communication* - services that have a high number of calls (dependency) between them can be deployed on the same server to reduce network latency; (2) *similar resource requirements* - services with similar usage of CPU and memory can be grouped together to ensure efficient server utilization; (3) *same scaling requirements* - services whose demand grows at similar proportion can be deployed together. Following these criteria, we deployed our services on two host nodes, hence allowing certain services to share resources across shops. In order to generate interference sub-graphs, we traverse the typed-graph in a way to extract distinct paths and determine whether a given path meets the conditions to be classified as an "interference path", which is a subgraph of the deployment graph. In fig. 2, we show a deployment graph with services being placed (dashed lines) on *host1* and *host2*, where the node's label indicates both the service name abbreviation and a suffix, which either stands for the shop instance (e.g., *service name:shop name*) or that the service is shared (e.g., suffix *Multi*). This implies that there are multiple instances of a single service in different executing paths. In this deployment graph, we consider all services that belong to both applications. To investigate the interference phenomenon (**step 4**), we specify its cause and

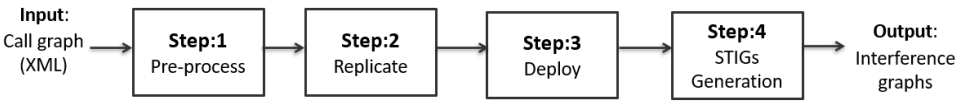

Figure 1: Approach

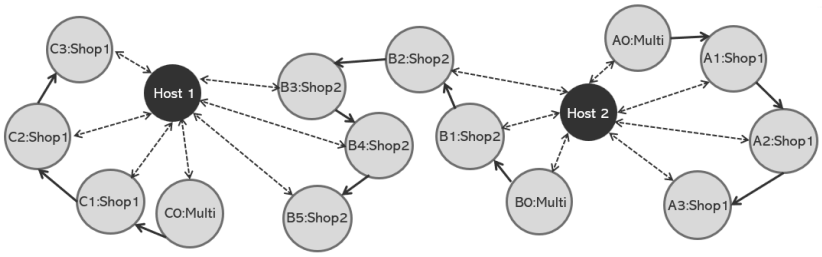

Figure 2: Service deployment in a hypothetical cluster. The solid arrows stand for the caller-callee relationship, whereas the dashed arrows represent the hosting relationship between service and host. Original service names are abbreviated for readability.

corresponding effect as follows. The cause of an interference refers to services within an execution trace that compete for resources with services from a *distinct* execution trace. The effect of an interference consists of services whose executions are hindered (i.e., sensitivity metric) by services from a *distinct* execution trace. In this context, we define a time interval of interference, that represents the maximum time overlap between the start and ending of the execution of any pair of services in the source and target traces. This time overlap can correspond to any number of nodes executed within these source and target traces. The probability of interference is the likelihood that a source service interfered with one or more target services, which is determined by both the time overlap and the amount of resources shared between source and target services. Therefore, a greater chance of

interference is brought on by a longer time interval and more resource usage (i.e., contention metric of the source service). Using this information, we can construct the STIGs (definition 2), whose probabilities are computed by the algorithm 1 and algorithm 2. In algorithm 1, we generate a list of the impacted pairs that work on two sets of nodes - referred to as the *sourceStack* and the *targetStack* based on the overlapping of execution times. The algorithm starts by sorting both source and target Stacks at the start time of their execution (**line 2**) and find out the current source node and the target nodes list based on the execution time conditions (**lines 3-10**). Based on that, we can estimate the interference probabilities for the STIG (**line 11** by calling algorithm 2). This involves computing for for each source node (*curSource*) the list of target nodes (*curTargetList*) and their corresponding the execution time overlap, as well as the magnitude of the resource usage shared with each source and target nodes (**lines 2-6**). The resulting list of impacted pairs is then returned by algorithm 1 (**line 12**).

Concerning scalability, the algorithm 1 has linear complexity on the number of m target nodes, i.e, $O(m)$. The complexity of the algorithm 2 is given by sorting both lists, $n \times log(n)$, and the worst case of comparing each node in the n-size *sourceStk* stack with every node in the m-size *targetStk* stack, $n \times m$. Because m and n are related (i.e., cannot grow independently), we assume linear complexity for the algorithm 2 as well, $O(n \times log(n) + n \times m)$.

Algorithm 1 Compute List of Impacted Pairs

```
1:  procedure IMPACTEDPAIRSLIST(sourceStk, targetStk)
2:      Sort both input sets by start of execution
3:      while sourceStk is not empty do
4:          Pop curSource from sourceStk
5:          while endTime of curSource > starting.time.target at head of targetStk do
6:              Pop curTarget from targetStk
7:              Put curTarget into curTargetList
8:              if endingTime of curSource is < ending time of curTarget then
9:                  Set starting time of curTarget to endingTime of curSource
10:                 Push it back to stack
11:             Append ComputeSTIGProb (curSource, curTargetList) to resultList
12:     return resultList
```

Algorithm 2 Compute STIG Interference Probability Edges

```
1: procedure COMPUTESTIGPROB(curSource, curTargetList)
2:     totalSourceT := curSource.endTime - curSource.startTime
3:     for each node in curTargetList do
4:         totalTargetT := min(curTarget.endTime, curSource.endTime) - curTarget.startTime
5:         curMag := curSource.resUsage + curTarget.resUsage
6:         return {source, target, probability, magnitude} := curSource, curTarget, totalTargetT /
   totalSourceT, curMag
```

3 Demonstration

We study of the interference phenomenon by selecting a query on two services belonging to different execution traces. For instance, we assume that the shared service instances $A0 : Multi$ and

$B0 : Multi$ (2) interfere with each other and also with neighboring services from distinct execution traces. We compute a STIG that contains interference impact from source to target service. Figure 3 (a) shows source(*front-end-M2:shop2*) and target(*front-end-M1:shop1*) interference impact of a single STIG. These source and target services are the instances of front-end service that are being shared by multiple shops and deployed on the same host. The goal is to show how the *front-end-M1:shop1* service instance execution is affected by *front-end-M2:shop2* service instance and other downstream services in the execution trace (i.e., sensitivity of the *front-end-M1:shop1*). The final probability of interference is calculated by summing up the probabilities of the edges directly connecting the source and the target nodes. The probability of interference between *front-end-M2:shop2* and *front-end-M1:shop1* is relatively higher (Value:1) than between other services due to this interference (start execution simultaneously). This also has an impact on other services because of the waiting time of these services in a queue to start their execution. In this demonstration, the generated STIGs allowed us to simulate various interference paths, which can, in turn, induce the propagation of anomalies between *shop1* and *shop2*. In this sense, we suggest that these STIG models can convey valuable knowledge for root-cause analysis. To visualize that, we extracted only the source and target pairs of the front-end service based on the maximum interference effect and obtained all associated source and target pairs within the same STIG. As a result, we got a dependency structure between services from both single and multiple simulated STIGs. As a reference, Figure 3 (b) shows a *structural dependency matrix (SDM)* representing the interference probabilities (STIG edges) between source and target services (STIG nodes) of two shops, where the dark color represents high probability. In the SDM on the right, the *front-end-M1:shop1* shows the highest probability (1.0) of being interfered by *front-end-M2:shop2*, which stems from the assumed determinism of these services starting simultaneously. Conversely, as the effect of interference propagates, there is a lower interference probability, which reflects less execution overlap among downstream services.

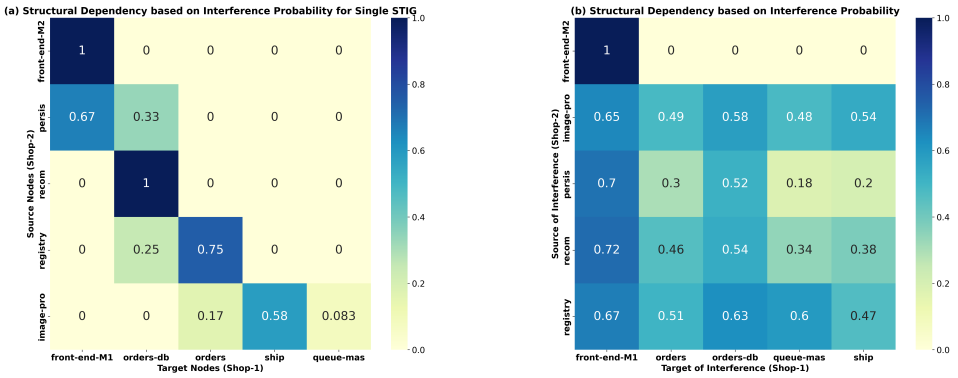

Figure 3: Structural Dependency Matrix. Table-A shows the probability of interference for a single STIG, where rows represent the target of the interference and each column is the source. Table-B consolidates the averages of interference across a set of STIGs.

4 Conclusion

We presented a novel approach to the problem of service interference in multi-tenant microservice architectures, where concurrency over shared resources induces the propagation of complex anomaly

patterns. Our approach relies on a Spatio-Temporal Interference Graph (STIG) and an a corresponding algorithm, which demonstrated with a hypothetical service placement cluster comprised of two popular benchmark applications. In future work, we plan to evaluate the interference phenomenon on a large-scale multi-tenant deployment and measure the confounding impact on the outcomes of the state-of-the-art methods for anomaly detection and root-cause analysis.

References

[1] Madhura Adeppady, Paolo Giaccone, Holger Karl, and Carla Fabiana Chiasserini. Reducing microservices interference and deployment time in resource-constrained cloud systems. *IEEE Transactions on Network and Service Management*, 2023.

[2] Vincent Bushong, Amr S. Abdelfattah, Abdullah A. Maruf, Dipta Das, Austin Lehman, Eric Jaroszewski, Michael Coffey, Tomas Cerny, Karel Frajtak, Pavel Tisnovsky, and Miroslav Bures. On Microservice Analysis and Architecture Evolution: A Systematic Mapping Study. *Applied Sciences*, 11(17), 2021.

[3] Marcello Cinque, Raffaele Della Corte, and Antonio Pecchia. Micro2vec: Anomaly detection in microservices systems by mining numeric representations of computer logs. *J. of Network and Computer Applications*, 208:103515, 2022.

[4] Shenghui Gu, Guoping Rong, Tian Ren, He Zhang, Haifeng Shen, Yongda Yu, Xian Li, Jian Ouyang, and Chunan Chen. Trinityrcl: Multi-granular and code-level root cause localization using multiple types of telemetry data in microservice systems. *IEEE Transactions on Software Engineering*, 2023.

[5] Claus Pahl, Pooyan Jamshidi, and Olaf Zimmermann. Architectural principles for cloud software. *ACM Transactions on Internet Technology (TOIT)*, 18(2):1–23, 2018.

[6] Yicheng Pan, Meng Ma, Xinrui Jiang, and Ping Wang. Faster, deeper, easier: crowdsourcing diagnosis of microservice kernel failure from user space. In *30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 646–657, 2021.

[7] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, and Calton Pu. Understanding performance interference of I/O workload in virtualized cloud environments. In *2010 IEEE 3rd international conference on cloud computing*, pages 51–58, 2010.

[8] Jacopo Soldani and Antonio Brogi. Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. *ACM Computing Surveys*, 55(3), 2022.

[9] Miguel G Xavier, Kassiano J Matteussi, Fabian Lorenzo, and Cesar AF De Rose. Understanding performance interference in multi-tenant cloud databases and web applications. In *2016 IEEE international conference on big data (big data)*, pages 2847–2852. IEEE, 2016.

[10] Ruyue Xin, Peng Chen, and Zhiming Zhao. Causalrca: Causal inference based precise fine-grained root cause localization for microservice applications. *J. of Systems and Software*, 2023.

[11] Chaobing Zeng, Fangming Liu, Shutong Chen, Weixiang Jiang, and Miao Li. Demystifying the Performance Interference of Co-Located Virtual Network Functions. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 765–773, 2018.