A BlackBox Approach to Profile Runtime Execution Dependencies in Microservices

Xuhang Gu
Louisiana State University
Baton Rouge, USA
xgu5@lsu.edu

Jianshu Liu
Louisiana State University
Baton Rouge, USA
jliu96@lsu.edu

Qingyang Wang
Louisiana State University
Baton Rouge, USA
qwang26@lsu.edu

***Abstract*—Loosely-coupled and lightweight microservices running in containers are likely to form complex execution dependencies inside the system. The execution dependency arises when two execution paths partially share component microservices, resulting in potential runtime performance interference. In this paper, we present a blackbox approach that utilizes legitimate HTTP requests to accurately profile the internal pairwise dependencies of all supported execution paths in the target microservices application. Concretely, we profile the pairwise dependency of two execution paths through performance interference analysis by sending bursts of two types of requests simultaneously. By characterizing and grouping all the execution paths based on their pairwise dependencies, the blackbox approach can derive a clear dependency graph(s) of the entire backend of the microservices application. We validate the effectiveness of the blackbox approach through experiments of open-source microservices benchmark applications running on real clouds (e.g., EC2, Azure).**

***Index Terms*—Microservices, performance analysis, dependency**

I. Introduction

Web application architecture is gradually evolving from the traditional monolithic multi-tier-based to loosely-coupled and lightweight microservices [14]. This trend is due to the special advantages of the microservice architecture in many aspects, such as fine-grained scalability, cross-team development, friendly deployment, etc. However, decomposing the originally monolithic architecture into hundreds to thousands of fine-grained microservices creates complex internal communication dependencies among different microservice components, causing significant challenges for performance prediction and management [14], [19]. For example, to manage performance and reason about system behavior, Google's recent work [10] discussed how to explicitly track microservice dependencies through proactive control or passive measurements.

In this paper, we present a blackbox approach that utilizes legitimate and public HTTP requests to profile the dependencies of the runtime execution paths inside the system. A typical execution path is triggered by an incoming HTTP request, which traverses through a series of microservice components to accomplish a transaction. For example in an e-commerce application, the order execution path may involve inventory, pricing, and credit card processing microservices. The dependency between two execution paths arises when they share some microservice components. A recently released Alibaba trace [14] shows that 5% microservices (called "hotspot" microservices) are shared by 90% execution paths in their application, showing that execution dependency widely exists in a production system.

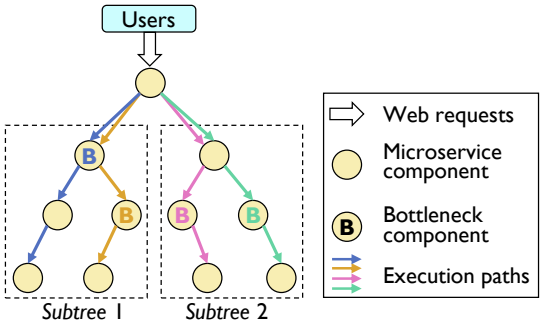


Fig. 1: Call graph is topologically similar to a tree. Execution paths that share upstream components have performance interference and can be grouped together.

The pairwise dependency can be profiled through performance interference analysis by sending bursts of two types of requests simultaneously. Specifically, we conclude that dependency exists between two execution paths when the normal latency of one type of requests are significantly prolonged by the burst of the other type of requests. By applying performance interference analysis to all the execution paths, we can divide them into multiple performance dependency groups. Execution paths within the same group have dependency mutually while execution paths across dependency groups have no dependency. In the later sections, we will demonstrate how we conduct performance interference analysis through experiments of an open-source microservices benchmark application running on real clouds (e.g., EC2). Our hypothesis is that precise profiling of runtime dependency of microservices applications will help system administrators to better reason about system abnormal behaviors and manage system performance under real-world bursty workload in production.

II. Dependency in Microservice Call Graph

A microservice call graph is *topologically similar to a tree* [14]. The root component of the tree is the gateway microservice that dispatches user requests to mid-tier/backend components. The mid-tier/backend components then further

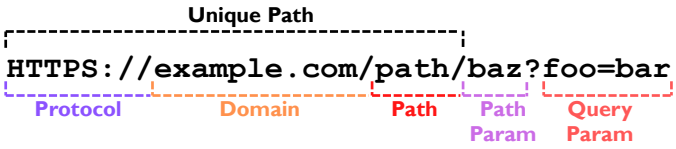


Fig. 2: Structure of a URL for example.com. Typically, parameters can be sent both in the URL and in the body of the HTTP request. We ignore the parameters to find the unique paths.

dispatch requests to multiple downstream components, which are also organized in a subtree structure. Each user request will trigger an execution path that traverses from the gateway to some downstream components as shown in Fig. 1. Different user requests accessing the same subtree must share microservice components, which creates potential performance interference between them. For example, when a shared bottleneck microservice component is overloaded, the performance of all execution paths sharing the same component will be degraded.

Depending on the relative location of bottleneck components among different paths, performance interference creates different execution dependencies. Fig. 1 illustrates two representative execution dependencies among different paths that have shared components. In subtree 1, the bottleneck component of the blue execution path is upstream of the bottleneck component of the orange execution path (called *sequential dependency*). Once the blue path is overloaded, the bottleneck component of the blue execution path will block the execution of both the blue and the orange execution paths. On the other hand, if the orange path is overloaded, the bottleneck component of the orange path could start to have *cross-tier queue overflow* to the upstream component that is shared by both blue and orange paths [22], which blocks the execution of the two paths. In subtree 2, the bottleneck components of the pink and the green execution path are located in different branches of the subtree (called *parallel dependency*), while they have a shared upstream component. Performance interference occurs when either of the execution paths is overloaded and the bottleneck component causes queue overflow to the shared upstream component.

By employing systematic profiling, our approach can deduce the execution dependencies among various paths and categorize them into distinct dependency groups.

III. Overview of the Dependency Profiler

Our approach behaves as an external user that accesses the target microservice application through public HTTP requests without having any prior knowledge about the backend structure. We profile the target microservice application with the proposed *Pependency Profiler* by sending different types of HTTP requests and recording the end-to-end response time of each request. By analyzing the existence of performance interference between any pair of requests under different conditions, the Profiler can identify the pairwise execution dependency (i.e., parallel and sequential) and then construct the dependency groups based on the pairwise dependencies with the following four steps.

TABLE I: Website pages crawled in our experiments.

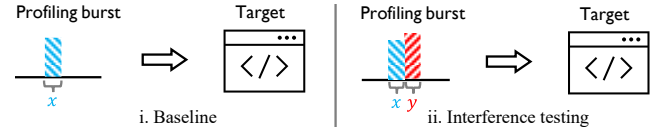
Name	# of crawled URLs	# of unique paths	Total time
Amazon	5,000,000+	96	15 hours
BestBuy	4,000,000+	83	14 hours
NYTimes	1,000,000+	116	8 hours
SocialNetwork	1,000,000+	32	6 hours
SockShop	500,0000+	18	4 hours

Extracting supported execution paths via public URLs. In a microservice-based web application, each type of user request would traverse among multiple components and trigger one execution path. Hence, we can profile the public URLs of the target application to retrieve all supported user requests, and we label the execution path with the request name. Website scraping tools [1], [2] can profile all supported HTTP requests of a target system. In our implementation, we use similar approaches introduced in the work [21] by using the script-based web browser PhantomJS to retrieve and distinguish dynamic requests from static ones automatically. For some dynamic requests that require input (e.g., POST request), we may need to provide some initial values for associate input forms (e.g., user name and password).

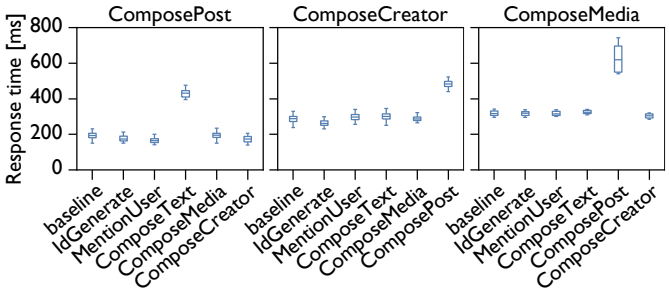
Typically, modern large-scale websites may contain more than millions of public URLs due to mixed combinations of different URL parameters. However, the number of unique execution paths without considering input parameters is small. As illustrated in Fig. 2, a URL usually consists of four parts: protocol, domain, path, and parameters [16]. We consider URLs with the same protocol, domain, and path but with different input parameters as sharing the same execution path in the system and use a *unique path* to represent them. While it is true that a single path with varying parameter combinations can lead to different execution paths in the microservices backend (e.g., due to cache effects), we opt for a broader approach. Specifically, we identify parameter combinations that consistently yield stable response times, considering them as representative instances of a "unique path. Table I shows representative websites crawled in our experiments[1]. For each unique path, we sample one valid URL parameter combination for the further steps.

Testing existence of performance interference. Based on all extracted valid unique paths, we need to test whether any two execution paths have performance interference before we identify the dependencies. Fig. 3 shows our pairwise performance interference testing. We first profile each execution path by sending a burst of single type requests as baseline (see. Fig. 3a(i)). Next, we send a pair of execution paths successively in one burst for interference testing (see. Fig. 3a(ii)). Finally, we analyze the response time of each execution path under different cases. Fig 3b shows example

[1]We compliant the "robots.txt" of the target commercial websites (e.g., Amazon, BestBuy) during our crawling to avoid any ethical problem, thus the number may vary in different scenarios.

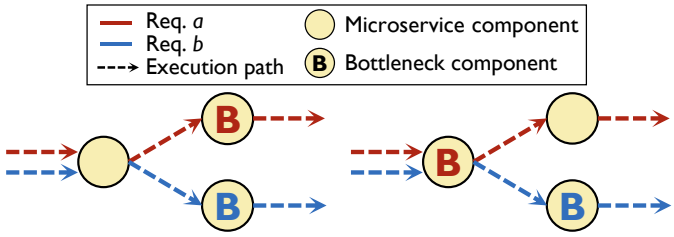


(a) Performance interference testing setup.



(b) Boxplot of the target path response time when it conducts pair-wise dependency testing with other paths. We use SVM to determine the cases with significantly larger response time than the baseline as the existence of performance interference (e.g., `ComposePost` and `ComposeText` have performance interference).

Fig. 3: Testing for performance interference



(a) Parallel dependency. The requests a and b have different bottleneck components while they share an upstream component.

(b) Sequential dependency. The bottleneck component of a is an upstream component of the bottleneck component of b.

Fig. 4: Illustration of different execution dependencies

results of response time analysis. If pair-wise performance interference exists, the response time of the sample execution path should be significantly higher than the baseline. In our implementation, we adopt a one-class Support Vector Machine (SVM) classifier [4], [20] to systematically observe the existence of pair-wise performance interference.

Identifying dependency type between two execution paths. Given any two execution paths, we identify the dependency type by analyzing the triggering conditions of performance interference. Similar to testing the performance interference, we send bursts consisting of two types (namely req. a and b) of requests with different volumes and sending orders. Then we check whether the changes in volumes or sending orders affect the existence of performance interference.

Parallel Dependency exists when the two execution paths have different bottleneck components while sharing an upstream (see Fig 4a), which indicates that type a requests can affect type b requests only when cross-tier queue overflow to the shared upstream component. For example, if the two types of requests are sent successively in an order of $a \rightarrow b$, the performance of the type b requests will be degraded if the bottleneck in path a causes its local queue to fill up and further the queue overflows to the shared upstream component. Then the next incoming requests (type b) will be blocked at the shared upstream component due to the cross-tier queue overflow (thus performance interference occurs).

However, if the bottleneck in the path a does not cause cross-tier queue overflow due to low volume, the incoming type b requests would directly reach its downstream components without blocking. Then we could not observe performance interference between paths a and b. Hence, we send a series of bursts of requests a and b and gradually increase the volume of requests (type a) from low to high until we reach a predefined maximum volume limitation (to avoid a shutdown of the target system). During the process, (1) if no performance interference is observed at any volume, we consider that the two execution paths have no dependency; (2) if the existence of performance interference varies with the volume (e.g., no interference at low volume and interference arises when volume increased), we consider that the two paths have a parallel dependency relationship; (3) if the performance interference exists persistently (does not vary with the volume), we move to the next step *Sequential Dependency* testing.

Sequential Dependency exists when the bottleneck component of one execution path is an upstream component of the other path (see Fig 4b). In contrast to the parallel dependency, the upstream path (e.g., a) would always have performance interference with the downstream path (e.g., b) no matter how much volume of the first type of request was sent. This is because the bottleneck triggered by the upstream path happens on the shared upstream component, which can block the incoming requests (type b) directly without queue overflow.

However, if we change the sending order of profiling requests (e.g., $b \rightarrow a$), the first arrived requests (type b) would reach its bottleneck component, and we can not observe the performance interference on the next incoming requests (type a) unless cross-tier queue overflow occurs to the shared upstream component. Hence, we send a series of bursts consisting of the two types of requests with small volumes that cannot cause cross-tier queue overflow in path b with different sending orders. If we observe the performance interference between path a and b vanishes after switching the sending order, it suggests a sequential dependency between paths a and b. We notice that if the volume of requests b can cause queue overflow to the shared upstream, we may wrongly consider a parallel dependency as a sequential dependency. To prevent such a case, we still need to send a series of bursts by gradually decreasing the volumes to void queue overflow. If the reversed order of requests always leads to different performance interference, we can label the paths a and b have a sequential dependency.

Constructing the dependency groups. After identifying the pairwise dependency, we can construct the dependency group based on the pairwise dependencies. We use an arrow to connect execution paths with sequential dependency (see Fig. 5a) while using connected arrows to illustrate execution paths with

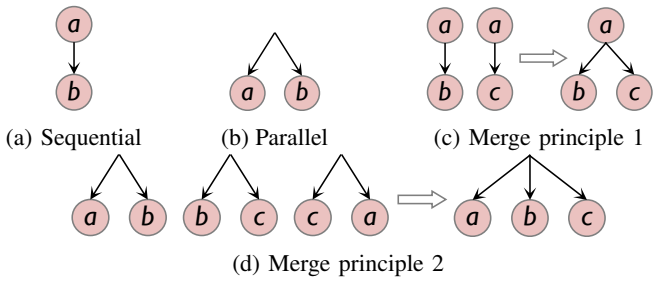


Fig. 5: Construction of dependency groups. (a) Sequential dependency. (b) Parallel dependency. (c) Shared sequential dependencies. (d) Shared parallel dependencies.

parallel dependency (see Fig. 5b). In addition, the execution paths that have a connection mutually form a dependency group. We construct the dependency group based on the pairwise dependencies with the following two principles.

Merge principle 1, *shared sequential dependencies*. If two sequential dependency pairs have a shared upstream path, the two downstream paths have a parallel dependency. Then the three paths can be merged with the upstream path connecting the two downstream paths, as shown in Fig. 5c.

Merge principle 2, *shared parallel dependencies*. If three paths have a mutual parallel dependency on any pair of them, the three paths can be merged with connected arrows, as shown in Fig. 5d.

IV. Results

Experimental setup. To evaluate the feasibility of our Profiler in real cloud production environments, we deploy an open-source microservice application SocialNetwork [11] on two popular commercial cloud platforms (Amazon EC2 [5], Microsoft Azure [6]). SocialNetwork implements a broadcast-style social network website with uni-directional follow relationships, where users can create, view, and comment on posts. We use separate containers to deploy microservice components in Docker Swarm Mode, where each is hosted by one container with at least a dedicated vCPU. To simulate normal users accessing the application as baseline workloads, we adopt RUBBoS workload generator [3]. Each user follows a Markov chain model to navigate among web pages, with an average 7-second Poisson distributed thinking time between every two consecutive requests from the same user. By controlling the number of concurrent users who access the target system, we can simulate different workloads.

Results. Fig 6 shows the topology and profiling result of dependency groups in the SocialNetwork application. The top part illustrates the topology of the SocialNetwork microservice application. A user request to the application will be dispatched by the frontend gateway and trigger a series of microservice components, which form a call graph. The internal communication between microservices and call graph is invisible to external users and our Profiler. The bottom part shows the dependency groups constructed by the Profiler,

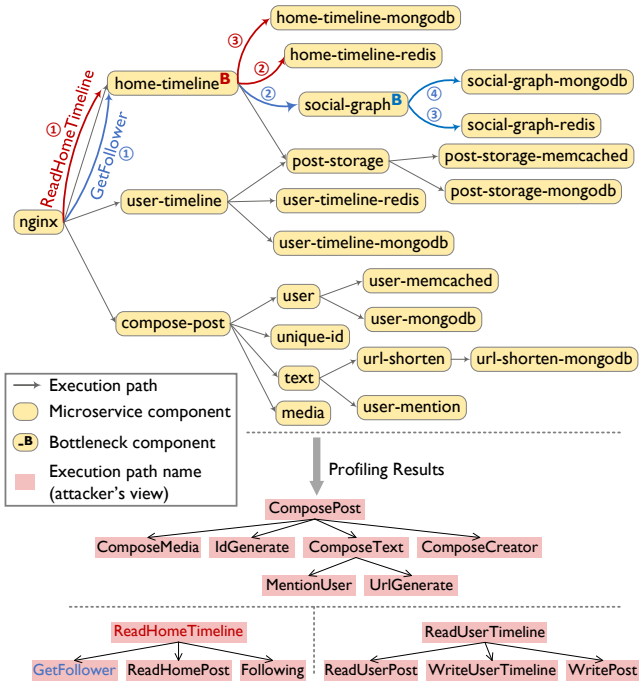


Fig. 6: Dependency groups constructed by our Profiler for the SocialNetwork benchmark application.

where each element represents one execution path and the arrows connect paths with sequential dependency. For instance, execution paths `ReadHomeTimeline` and `GetFollower` have a sequential dependency while `GetFollower` and `ReadHomePost` have a parallel dependency.

V. Related Work

Distributed tracing has garnered significant attention within the open-source community, with numerous active projects dedicated to its implementation, such as Pinpoint [18], Zipkin [24], and Jaeger [12]. As the adoption of distributed tracing grows in complex distributed systems, it has given rise to specifications like OpenTracing [8] and OpenCensus [17]. Distributed tracing finds widespread utility in tasks like anomaly detection (as demonstrated in CRISP [23], Canopy [13], and Pivot tracing [15]), resource management (e.g., FIRM [19]), and evaluating microservice architecture (as seen in the work of Engel et al. [9] and Bogner et al. [7]).

Despite the precision of distributed tracing in uncovering execution dependencies within microservices, it fundamentally relies on intrusive white-box approaches. These methods necessitate deep involvement from service developers and require the instrumentation of each individual microservice component, as well as a complex infrastructure for collecting and processing trace data. Our research demonstrates that even a black-box approach, utilizing publicly accessible HTTP requests, can yield a significant amount of valuable runtime execution dependency information from the backend microservices system. This approach empowers system administrators and potential adversaries alike to gain profound insights into

the system's architecture and vulnerabilities in a non-intrusive manner.

VI. Conclusion

In this paper, we introduce a black-box methodology for accurately profiling the internal pairwise dependencies within a target microservices application using legitimate external HTTP requests. Our approach involves analyzing the performance interference between execution paths by simultaneously sending bursts of two distinct request types and monitoring latency variations. By categorizing and clustering execution paths based on their pairwise dependencies, our blackbox approach constructs comprehensive dependency groups for the entire backend of the microservices application. To validate its effectiveness, we conducted experiments on open-source microservices benchmark applications deployed on real cloud platforms such as EC2 and Azure. The runtime profiling of microservices dependencies enabled by our approach empowers system administrators to gain deeper insights into system anomalies and enhance performance management, especially when dealing with real-world bursty workloads in production environments.

VII. Acknowledgements

This research has been partially funded by the National Science Foundation under Grant CNS-2000681 and from contracts with Fujitsu Limited. Any opinions, findings, and conclusions are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] "20 best open source automation testing tools in 2022," https://www.softwaretestinghelp.com/open-source-testing-tools/, accessed, 2022.

[2] "Automate software testing — testim," https://go.testim.io/testim-automate-enterprises, accessed, 2022.

[3] "Rubbos: Bulletin board benchmark," https://projects.ow2.org/view/rubbos/, accessed: 2021.

[4] "Scikit learn one class svm," https://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html, 2023.

[5] "Amazon ec2," https://aws.amazon.com/ec2/, Accessed: 2021.

[6] "Microsoft azure," https://azure.microsoft.com/en-us/, Accessed: 2021.

[7] J. Bogner, S. Schlinger, S. Wagner, and A. Zimmermann, "A modular approach to calculate service-based maintainability metrics from runtime data of microservices," in *International Conference on Product-Focused Software Process Improvement*. Springer, 2019, pp. 489–496.

[8] CNCF, "The opentracing project," https://opentracing.io/, 2022.

[9] T. Engel, M. Langermeier, B. Bauer, and A. Hofmann, "Evaluation of microservice architectures: A metric and tool-based approach," in *Information Systems in the Big Data Era: CAiSE Forum 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings 30*. Springer, 2018, pp. 74–89.

[10] S. Esparrachiari, T. Reilly, and A. Rentz, "Tracking and controlling microservice dependencies: Dependency management is a crucial part of system and software design." *Queue*, vol. 16, no. 4, pp. 44–65, 2018.

[11] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *ASPLOS*, 2019, pp. 3–18.

[12] Jaeger, "Jaeger: open source, end-to-end distributed tracing," https://www.jaegertracing.io/, 2022.

[13] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi *et al.*, "Canopy: An end-to-end performance tracing and analysis system," in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 34–50.

[14] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM SoCC*, 2021, pp. 412–426.

[15] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 35, no. 4, pp. 1–28, 2018.

[16] S. A. Mirheidari, S. Arshad, K. Onarlioglu, B. Crispo, E. Kirda, and W. Robertson, "Cached and confused: Web cache deception in the wild," in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 665–682.

[17] OpenCensus, "Opencensus," https://opencensus.io/, 2023.

[18] Pinpoint, "Pinpoint — leading open-source apm," https://pinpoint-apm.github.io/pinpoint/, 2023.

[19] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "{FIRM}: An intelligent fine-grained resource management framework for slo-oriented microservices," in *14th ({OSDI} 20)*, 2020, pp. 805–825.

[20] B. Schölkopf, R. C. Williamson, A. Smola, J. Shawe-Taylor, and J. Platt, "Support vector method for novelty detection," *Advances in neural information processing systems*, vol. 12, 1999.

[21] H. Shan, Q. Wang, and Q. Yan, "Very short intermittent ddos attacks in an unsaturated system," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2017, pp. 45–66.

[22] Q. Wang, C.-A. Lai, Y. Kanemasa, S. Zhang, and C. Pu, "A study of long-tail latency in n-tier systems: Rpc vs. asynchronous invocations," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 207–217.

[23] Z. Zhang, M. K. Ramanathan, P. Raj, A. Parwal, T. Sherwood, and M. Chabbi, "{CRISP}: Critical path analysis of {Large-Scale} microservice architectures," in *USENIX ATC 22*, 2022, pp. 655–672.

[24] Zipkin, "Openzipkin - a distributed tracing system," https://zipkin.io/, 2023.