MicroProf: Code-level Attribution of Unnecessary Data Transfer in Microservice Applications

SYED SALAUDDIN MOHAMMAD TARIQ, University of Michigan - Dearborn,
LANCE MENARD, University of Michigan - Dearborn,
PENGFEI SU, University of California - Merced,
PROBIR ROY, University of Michigan - Dearborn,

The microservice architecture style has gained popularity due to its ability to fault isolation, ease of scaling applications, and developer's agility. However, writing applications in the microservice design style has its challenges. Due to the loosely coupled nature, services communicate with others through standard communication APIs. This incurs significant overhead in the application due to communication protocol and data transformation. An inefficient service communication at the microservice application logic can further overwhelm the application. We perform a grey literature review showing that unnecessary data transfer is a real challenge in the industry. To the best of our knowledge, no effective tool is currently available to accurately identify the origins of unnecessary microservice communications that lead to significant performance overhead and provide guidance for optimization.

To bridge the knowledge gap, we propose MicroProf, a dynamic program analysis tool to detect unnecessary data transfer in Java-based microservice applications. At the implementation level, MicroProf proposes novel techniques such as remote object sampling and hardware debug registers to monitor remote object usage. MicroProf reports the unnecessary data transfer at the application source code level. Furthermore, MicroProf pinpoints the opportunities for communication API optimization. MicroProf is evaluated on four well-known applications involving two real-world applications and two benchmarks, identifying five inefficient remote invocations. Guided by MicroProf, API optimization achieves an 87.5% reduction in the number of fields within REST API responses. The empirical evaluation further reveals that the optimized services experience a speedup of up to 4.59×.

CCS Concepts: • **Software and its engineering → Software performance**; **Cloud computing**.

Additional Key Words and Phrases: Microservice, Unnecessary communication, Dynamic program analysis

1 INTRODUCTION

Recent years have seen increasing adoption of microservice architecture in cloud application design. Big tech companies such as Amazon, Twitter, Facebook, Netflix, Uber, and eBay have adopted various microservice-based design patterns while designing their services [62] [44] [15] [61]. The microservice architectural design leverages service-oriented architecture (SOA) principles at a finer granularity. Each service in microservice applications is loosely coupled and communicates with others through standard communication APIs. Such modularity

This is a new article, not an extension of a conference paper.
This work was supported by the National Science Foundation under Grant No. CNS-2006373.
Author's address: Syed Salauddin Mohammad Tariq, University of Michigan - Dearborn, 4901 Evergreen Road, Dearborn, MI 48128, USA; email: ssmtariq@umich.edu. Lance Menard, University of Michigan - Dearborn, 4901 Evergreen Road, Dearborn, MI 48128, USA; email: lmenard@umich.edu. Pengfei Su, University of California - Merced, 5200 Lake Rd, Merced, CA 95343, USA; email: psu9@ucmerced.edu. Probir Roy (corresponding author), University of Michigan - Dearborn, 4901 Evergreen Road, Dearborn, MI 48128, USA; email: probirr@umich.edu.

1544-3566/2023/9-ART $15.00
https://doi.org/10.1145/3622787

```
%------- Employee.java ----------------%
public class Employee {
    public String name;
    public String address;
    public String email;
}
%------- PayrollController.java ------%
@RestController
public class PayrollController {
  @Autowired
  private HRService hrService;
  @GetMapping("/getEmployee/{id}")
  public String getPaidEmployeeNameById(int id) {
      Employee e = hrService.getEmployee(id); // unnecessary data transfer
      return e.getName();
  }
  @GetMapping("/getEmployee/validate/{id}")
  public String validatePaidEmployeeInfo(int id) {
      Employee e = hrService.getEmployee(id); // not an unnecessary
          ↪ transfer
      return validate(id, e.getName(), e.getAddress(), e.getEmail());
  }
}
```

Listing 1. A motivating example: unnecessary data transfer

enables the developer's agility, fault isolation, and scalability. However, on the downside, due to disaggregation, microservice applications pose new challenges, such as efficient CPU and memory resource utilization [6] [69], reducing communication overhead [41] [18] [30] [63], and meeting the quality of service (QoS) [20] [47].

Previous research [19] [32] [58] has indicated that microservice applications possess a higher communication-to-computation ratio compared to monolithic applications. Microservice applications spend significant time processing network requests across services, contributing significantly to the end-to-end tail latency. Existing studies have addressed the communication overhead by communication-aware microservice scheduling [18], designing hardware acceleration for network packet processing [32], and coupling the dependent services [41]. However, these techniques fail to address the inherent communication overhead originating from unnecessary data transfer due to inefficient service API design. Listing 1 demonstrates the problem of unnecessary data transfer with an example.

In this example, the `payroll-service` microservice implements a function `getPaidEmployeeNa- meById(id)` inside the `PayrollController`, which returns the requested employee name. Within `getPaidEmployeeNameById(id)`, another remote microservice `HR-service`'s `getEmployee(id)` method is invoked to search its database for the requested employee. The API returns an object of the `Employee` class over the communication channel. Since `payroll-service` receives the `Employee` object from a remote service, we call it a `remote object`. While the `Employee` class has three fields, the caller function, `getEmployeeNameById(id)` only accesses a single field, `name`. As a result, transferring the other two object fields becomes unnecessary.

Recent research [5] has shown that optimizing microservice APIs can significantly reduce the number of object fields and transferred bytes up to 94% and 99%, respectively. To motivate the significance of the problem,

we further conducted a comprehensive grey literature review in Section 2. We identified 27 tech companies highlighting the prevalence of this category of code inefficiency in their microservice applications. Through optimization, developers reported improved resource efficiency.

One source of unnecessary data transfer is the inflexibility of traditional REST API. New query language and runtime engines such as GraphQL [23] were proposed to address this inflexibility. GraphQL enables clients to specify exactly what data they need, rather than retrieving a pre-determined set of data, which can help reduce the amount of unnecessary data transfer and improve performance.

However, the major challenge lies in the inability to determine the data usage in the caller microservices. Real-world microservices are vast and complex, which makes manual analysis to identify object field usage impractical. For example, Uber's microservice architecture has over 4,000 microservices and 40,000 distinct remote procedure call (RPC) endpoints [68]. Due to code complexity, scale, and data flow, identifying object field access manually is challenging for developers. Additionally, developers must prioritize optimizing unnecessary data transfer that causes significant performance constraints [27]. Due to the lack of observability tools, applications using GraphQL still encounter unnecessary data transfer due to developers making inessential field requests [53].

Previous research [8] [7] has explored the potential of static program analysis tools as a solution; however, they have certain limitations. One of the key limitations is the inability to accurately measure program performance inefficiencies. Performance measurement is crucial to prioritize code optimization based on the significance of performance issues. Additionally, static analysis tools have limited context; they can only examine code in isolation. However, it cannot monitor the calling context (aka call path), the context in which a function is invoked. While RPC APIs can be invoked from multiple caller functions, not all of them result in unnecessary data transfer and cause significant performance challenges. Therefore, knowing the calling context of an inefficient RPC API invocation through dynamic analysis is crucial. Moreover, static analysis tools often fall short of handling complex code analysis such as alias analysis. Hence, identifying such optimization opportunities without dynamic program analysis techniques is difficult. To the best of our knowledge, there are currently no dynamic program analysis tools capable of identifying performance inefficiencies in microservice applications resulting from unnecessary data communication and guiding developers on optimizing them.

Building such dynamic program analysis tools has its challenge. The tool requires monitoring remote objects received through RPC and their usage in the service. One way to monitor these remote object usage is to instrument the microservices and trace every memory access. However, tracing every memory access causes a significant overhead for the applications. For instance, HOTL [64] shows that trace-based memory analysis can incur 153× runtime overhead. As a result, a tracing-based approach to monitor every memory access becomes impractical. Especially, microservice applications are latency sensitive and have strict QoS requirements.

In this paper, we propose MicroProf, a lightweight dynamic program analysis tool, to tackle the aforementioned challenges. MicroProf uses a principled approach to identify unnecessary data communication that causes significant performance overhead. For this purpose, it evaluates the microservice applications in two phases. Firstly, it utilizes Jaeger [29] RPC tracing to pinpoint communication choke points. In the second phase, MicroProf leverages statistical profiling [1] to determine if the root cause of the overhead is unnecessary data transfer. To further refine the analysis, MicroProf performs code-centric attribution [22] [13] to identify the source code instructions, functions, and function call paths responsible for making inefficient requests for remote objects. Additionally, using data-centric attribution [38] [39], MicroProf identifies unnecessary object fields and guides developers in optimizing them.

The statistical memory profiling technique used by MicroProf helps reduce the measurement overhead compared to all memory access tracing. Similar to other distributed profiling tools such as CRISP [68], and Jaeger [52], MicroProf also allows for dynamic adjustment of the sampling rate, providing practitioners with more control over the runtime overhead. Furthermore, MicroProf implements memory profiling using hardware

debug registers. With the help of hardware debug registers, MicroProf can intercept remote object usage in a non-intrusive manner, further reducing the monitoring overhead.

To demonstrate MicroProf's effectiveness, we run MicroProf on four well-known microservice applications written in Java Spring Boot [56]: TrainTicket [70], Daytrader [45], ThingsBoard [60], and Eclipse-Kapua [14]. Among these applications, ThingsBoard and Eclipse-Kapua are widely-deployed IoT frameworks. On the other hand, to our knowledge, TrainTicket is the largest open-source microservice application benchmark written in Java that comprises 41 services. Initially developed by IBM as a monolithic application benchmark, Daytrader has recently been implemented in microservice architecture [27]. Guided by MicroProf, we identified five unnecessary data transfers in these applications. Upon optimizing the communication APIs, we observe up to 87.5% reduction in the number of fields within REST API response. The empirical evaluation reveals the optimization can achieve a speedup of 4.59×.

To summarize, this paper makes the following contributions:

- The paper proposes MicroProf, a novel dynamic program analysis tool to detect unnecessary data transfer that causes significant performance overhead in Java-based microservice applications. At the implementation level, MicroProf leverages techniques such as statistical profiling and hardware debug registers to monitor remote object usage.
- MicroProf performs code-centric attribution to identify the source code instructions, functions, and function call paths responsible for invoking inefficient remote communication APIs. Furthermore, it performs data-centric attribution to identify unnecessary remote object fields for further optimization.
- The paper evaluates MicroProf's effectiveness on two microservice application benchmarks (TrainTicket, DayTrader) and two widely-deployed IoT frameworks (ThingsBoard and Eclipse-Kapua). Guided by MicroProf, we optimized the inefficient remote API invocations and observed significant speedups.

The rest of the paper is organized as follows. In Section 2, we present a grey literature review on the industry experience of unnecessary data transfer. Section 3 examines the existing related work and distinguishes MicroProf. Section 4 introduces the relevant background knowledge. Section 5 introduces the proposed mechanism. Section 6 describes the implementation details of MicroProf. Section 7 discusses the empirical evaluation of MicroProf. Section 7.1 offers a discussion of applying MicroProf to four microservice applications. Finally, Section 9 concludes the paper.

Replication: We have shared the replication package[1] for further study.

2 GREY LITERATURE REVIEW ON UNNECESSARY DATA TRANSFER

Academic research on microservices is still relatively young, and tech companies adopting microservices are generating a substantial amount of grey literature [55]. There is a gap between academic research and industry practices, particularly in understanding the challenge of unnecessary data transfer. To bridge this gap, we conduct a concise yet insightful grey literature review to shed light on this important topic. Our study aims to answer the research question, "Is unnecessary data transfer a significant challenge in the tech industry?". We begin by identifying tech companies that adopted GraphQL as part of their microservice architecture. Since many of these companies publish engineering blogs to share their knowledge with the wider tech community, we rigorously search these blogs to identify posts related to GraphQL adaptation. Specifically, we select articles discussing the mitigation of over-fetching issues through GraphQL implementation. Our efforts resulted in identifying engineering blog posts from 27 different companies. The comprehensive list of these engineering blog posts is included in the MicroProf replication package [1], labeled from A1 to A27.

Meta's engineering blog (A17) laments the challenges of dealing with the disparities between the data their apps require and the corresponding server queries. Meanwhile, PayPal acknowledges (A19) that their REST APIs

[1]Replication package: https://figshare.com/s/5156839e442f03d97747

were sending clients more data than necessary, resulting in extraneous data transfer. On the other hand, Atlassian reports (A5) that as they began converting frequently used REST requests to equivalent GraphQL queries in Trello, they discovered that they were over-fetching vast amounts of data. According to 1stDibs (A1), by precisely specifying the required fields, they were able to decrease data response from over 1MB to approximately 90KB for a buyer viewing a full page of orders. These insights from the literature review collectively confirm that unnecessary data transfer poses a notable challenge for user-facing applications. Optimizing this inefficiency can result in significant enhancements in QoS and better utilization of cloud resources.

3 RELATED WORK

In this section, we summarize related works in three following directions:

Performance monitoring tools for distributed applications. A large body of work contributes to developing monitoring and tracing frameworks for distributed system applications. Magpie [4] provides a tracing system that records fine-grained events generated by the kernel and application components. Magpie further constructs workload models to predict system performance. Similar to Magpie, Google's Dapper [54] offers a distributed tracing platform that enables tracing difficult system issues that are impossible to reproduce. X-Trace [17] represents a tracing framework that generates a comprehensive view of distributed system traces by enabling relevant information logging for connected devices. Pivot tracing [42] identifies application bugs and configuration issues in Java-based distributed systems utilizing dynamic instrumentation with minimal execution overhead. More recently, CRISP [68] performs critical path analysis on large-scale microservice applications in order to identify performance issues. Guided by CRISP, the authors performed fine-grained analysis to identify the root cause of the performance overhead. Similarly, MicroProf also performs critical path analysis to identify significant performance overhead in microservices. However, once a critical path is identified, MicroProf goes further by performing fine-grained analysis to determine whether unnecessary data transfer is the root cause of the issue. To guide developers in optimizing their code, MicroProf attributes inefficiencies to specific source code instructions and functions.

Unnecessary data retrieval in cloud applications. Previous research [8] [7] has investigated unnecessary data retrieval from databases. Yang et al. performed an empirical study on database-backed web applications and highlighted several inefficiency patterns, including unnecessary data retrieval [67]. The authors further proposed static analysis tools to identify the inefficiency pattern in the application code. Recently, another static analysis tool is proposed [21] to identify potential data leaks to make microservice applications more secure. However, static analysis tools have certain limitations which restrict their abilities. In contrast, MicroProf uses dynamic program analysis to identify unnecessary data transfer that causes significant performance inefficiency.

Dynamic analysis for data utilization. Several dynamic program analysis tools have been proposed to identify data structure-related application inefficiencies. Xu et al. propose a runtime analysis tool to detect low-utilization data structures in Java applications [65]. However, to implement the tool, the authors modified the underlying JVM. Chilimbi et al. proposed structure splitting for Java applications to reduce the unnecessary data on the CPU cache [9]. More recently, JXPerf [57] and its successors OJXPerf [36] and DJXPerf [35] propose lightweight techniques to identify inefficient computation, memory allocations, and memory accesses in Java applications. However, none of these dynamic program analysis tools identify inefficient communication across microservices.

Unnecessary data movement optimization. GraphQL [23] is a query language for APIs and runtime for executing queries against existing data. The GraphQL clients can request the exact data needed by making a GraphQL query. As a result, GraphQL APIs enjoy significantly less overhead compared to REST APIs. While

GraphQL reduces the over-fetching, it does not detect an unnecessary request for data in the microservice application source code. Often, the runtime behavior is unknown to the developer and may result in unnecessary data queries. One such example is in Listing 3, where object fields are dependent upon calling context. In contrast to GraphQL, MicroProf detects unnecessary data transfer in the microservice applications.

4 BACKGROUND

In this section, we briefly discuss the technologies used by MicroProf.

Code-Centric and data-centric attribution. Code-centric attribution is a dynamic program analysis technique where runtime events/metrics are attributed to application source code instructions, code blocks such as loops, functions, and calling contexts. Tools such as VTune [28], Oprofile [34], CodeAnalyst [13], and gprof [22] are examples of code-centric tools that link performance metrics to source code. However, code-centric attribution falls short of attributing events/metrics to data objects. In contrast, data-centric attribution attributes events/metrics to dynamically allocated objects and variables. Tools such as HPCToolKit [39], ArrayTool [40], and StructSlim [49] perform both code- and data-centric attribution to identify the root causes of various performance inefficiencies.

Java Virtual Machine Tool Interface (JVMTI). JVMTI [11] is a native programming interface for developing tools to monitor Java applications running on Java Virtual Machine (JVM). A tool using the JVMTI, also known as a JVMTI agent, can be statically linked or dynamically loaded during JVM initialization. Once attached to the JVM, the agent can use the JVMTI to monitor JVM states, including but not limited to profiling, debugging, monitoring, thread analysis, and coverage analysis tools.

Debug registers. On an x86 processor, a debug register [12] is a hardware component that allows developers to debug applications at the source code level. Debug registers enable trapping a target application's memory reference and instruction execution. A debug register is configured to set a watchpoint at the target memory address to trap a memory reference. When the application refers to the monitored memory address, it causes an interrupt and sends a signal to a registered signal handler. The signal handler can observe the application state, such as the instruction pointer responsible for memory reference. Finally, the signal handler collects the information to process the event further.

5 METHODOLOGY

Real-world microservices are complex, with many remote services interacting with each other in intricate ways. Fig 1 presents a simplified scenario where two services, `payroll-service` and `admin-service` invoke remote endpoint, `getEmployee()` of the `hr-service`. However, only remote invocations from `payroll-service` cause significant performance overhead, warranting further investigation. Nevertheless, as shown in Listing 1, only the caller function `getPaidEmployeeNa- meById()` demonstrates unnecessary data transfer. Pinpointing such inefficiency will require measuring the communication overhead across services and monitoring the remote data usage of the caller services. However, the situation can quickly escalate as the monitoring effort grows significantly with the increase in the number of API endpoints and their invocations.

To minimize the effort, we adopt a selective and incremental approach for monitoring. Our incremental monitoring is based on a principled approach: first, we identify critical service-to-service call paths that cause significant overhead in serving end-user requests by critical path analysis (CPA) [66]. CPA allows us to identify the specific service chains that create performance challenges in the microservice application, ensuring we target our monitoring efforts where they are most needed. Once we identify the critical path and the involved services, we perform fine-grain program analysis on the caller services of these critical paths to monitor remote object usage.

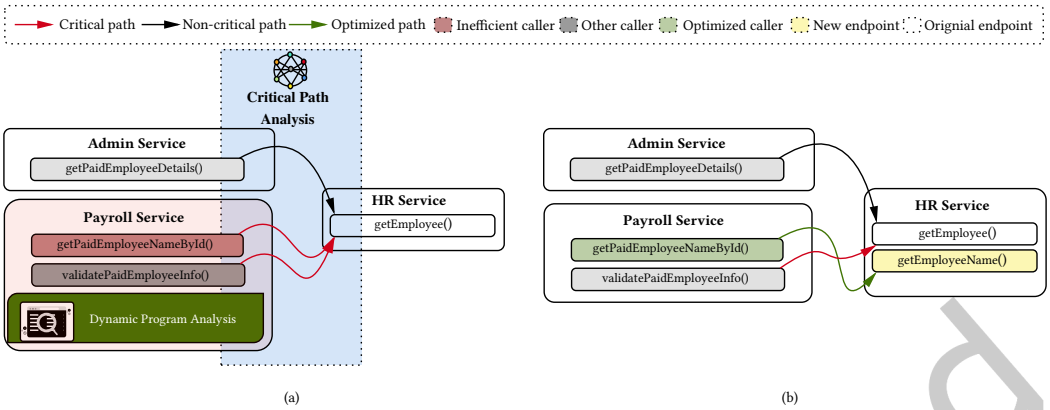

Fig. 1. (a) Critical path analysis narrows the search space to two critical paths. Subsequently, MicroProf's dynamic program analysis identifies inefficiency in `getPaidEmployeeNameByID`. (b) New endpoint introduced to avoid unnecessary data transfer.

5.1 Critical path analysis of microservices

We perform critical path analysis on the service dependency graph of the microservice application. For this purpose, we utilize distributed tracing [51], a technique that monitors and analyzes the flow of requests across multiple services involved in processing user requests. In recent years, several distributed tracing tools have been proposed, including Google's Dapper [54], Jaeger [29], Zipkin [71], Apache's HTrace [26], and LightStep [37]. In this paper, we leverage Jaeger to collect microservice call traces. Jaeger exposes the collected data using OpenTelemetry [46] [33] compliant format.

Each end-to-end OpenTelemetry trace consists of a list of spans. A span is an individual unit of work done in a distributed service. A span data structure consists of attributes such as operation start time, end time, and a list of child spans. As services call to other services via RPC, spans forms directed acyclic graphs (DAG). Fig 2-a shows an example of DAGs constructed from N-traces. From the DAGs of all traces, we further construct an aggregated calling context tree (CCT) [2], which captures the path and order of these service invocations, forming a tree-like structure. Fig 2-b depicts an example of CCT. We enhance the CCT nodes by incorporating two additional metrics: inclusive service time and exclusive service time. The inclusive service time encompasses the time spent on the service node itself as well as the time spent on the subsequent callee services. Conversely, the exclusive service time of a service node focuses solely on time spent on that service operation, excluding the service times of its callees.

A service that exhibits a higher exclusive service time plays a significant role in the overall request latency, indicating a need for further investigation into potential unnecessary data transfer. On the other hand, when a service node demonstrates inefficiency with a higher inclusive service time, it suggests that optimization efforts could potentially eliminate subsequent service calls in the call path, thereby reducing the overall request latency. To effectively prioritize both metrics, we visualize the service call paths within the CCT in a 2D space, plotting the exclusive and inclusive service times. Fig 2-c illustrates the plot. Subsequently, we calculate the Euclidean distance of each service from the center of this 2D space. The services are then ranked in descending order based on their distance from the center, as depicted in Fig 2-d. By following the prioritized list, we selectively analyze the services in order to identify unnecessary data transfer.

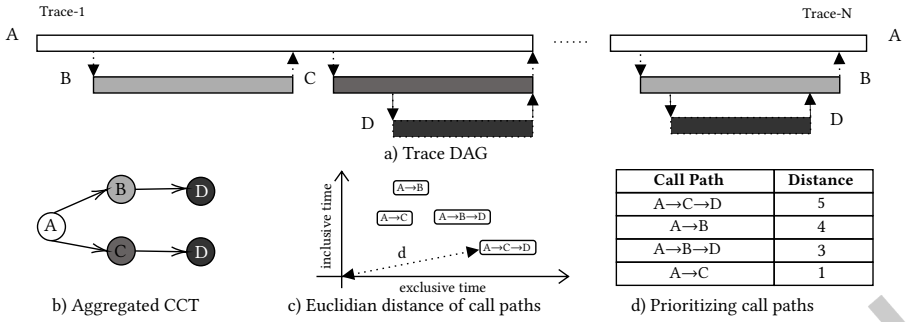

Fig. 2. Critical path analysis from distributed tracing.

5.2 Detecting unnecessary data transfer

Depending on the context of remote invocation and reference to the object, a remote object transfer may or may not result in an unnecessary transfer. For instance, from Listing 1, both `getPaidEmployeeNameById(id)` and `validatePaidEmployeeInfo(id)` receive a remote object of `Employee` class via invoking remote method `getEmployee(id)`. However, while `getPaidEmployeeNameById(id)` does not fully utilize the remote `Employee` object, `validatePaidEmployeeInfo(id)` utilizes all its fields for validation purposes. Therefore, to accurately identify unnecessary data transfer in the source code, it is crucial to (1) capture the call path of remote object allocation and (2) derive utilization metrics specific to each remote object allocation call path.

One approach to detect unused fields in remote objects is to monitor the field accesses of all the remote objects exhaustively. Such exhaustive monitoring will require instrumenting all the fields of remote objects and monitoring their accesses. However, this instrumentation-based mechanism will incur significant overhead in the microservice applications and become impractical in the production environment. In order to minimize the exhaustive tracing of remote objects, MicroProf samples a subset of remote objects to monitor memory references within the microservice process. Furthermore, MicroProf eliminates the need for exhaustive instrumentation by implementing address monitoring via hardware debug registers.

Statistical sampling. During the allocation of remote objects, MicroProf selects a subset of the objects via well-known Monte Carlo sampling algorithm [43] and records the call path of the allocation. We define this call path as `allocation context`. Sampling-based call path profiling is a well-established mechanism, frequently adopted in performance monitoring tools such as GProf [22], Oprofile [34], HPCToolkit [1], and Intel VTune [28]. Supported by the law of large numbers [25], as the quantity of independent and identically distributed (i.i.d.) random samples grows, the distribution of the samples gradually approximates the population distribution across all the allocation contexts. A theoretical analysis of sampling-based call path profiling can be found in [59] (Appendix A).

Suppose the probability of invoking an allocation context, C, is represented with $P[c]$. When dealing with a critical path (identified in Section 5.1) that involves numerous remote data transfer, a highly probable inefficient allocation context will significantly negatively impact performance. We approximate the probability by statistical sampling of allocation contexts and identify the hot allocation contexts contributing to many remote invocations. Suppose we collects N i.i.d. random samples: $x_1, x_2, ..., x_N$, where each x_i represents the allocation context observed on the i^{th} sampled event. Formula 1 calculates the observed probability for allocation context, where N_c represents the number of occurrences of allocation context. With a large sample size, N, the observed probability is close to the actual probability distribution.

$$P'[c] = \frac{N_c}{N} \tag{1}$$

However, monitoring references to all the fields of sampled objects simultaneously is difficult due to the constrained number of hardware debug registers. To overcome the limitation, MicroProf employs a strategy of uniform random sampling to select a subset of fields from the target object. MicroProf further collects many samples across microservice invocations and aggregates them by allocation context. This uniform random sampling ensures that all fields have an equal chance of being monitored. With a large number of independent and identically distributed (i.i.d.) random samples collected from microservice requests, the distribution of field accesses in the samples gradually converges to the overall population distribution. MicroProf monitors the references to these sampled fields via hardware debug registers and calculates a metric of the associated class in the allocation context. MicroProf uses this utilization metric to evaluate whether hot allocation contexts involve unnecessary data transfer.

Algorithm 1 Algorithm of unnecessary data transfer detection

Input: Microservice, M requesting the remote objects
Input: The class of the remote objects
Output: Class utilization
Output: Class field utilization
1: **repeat**
2: Intercept remote object allocation
3: Randomly select a remote object, O_i, during allocation
4: Randomly select a field, ${f_O}^i$ of the remote object O_i
5: Collect the allocation context, C of the remote object O_i
6: Record pair $< C, {f_O}^i >$
7: Set a trap to monitor the references to the field, ${f_O}^i$
8: **while** O_i is alive **do**
9: Intercept references to field ${f_O}^i$
10: **if** The field ${f_O}^i$ is accessed in Microservice, M **then**
11: Increment access count of the field, f^i of the class
12: Remove trap on the field, ${f_O}^i$
13: break
14: **end if**
15: **end while**
16: **if** The field ${f_O}^i$ was not accessed in Microservice, M **then**
17: Increment unused count of the field, f^i of the class
18: Remove trap on the field, ${f_O}^i$
19: **end if**
20: **until** Not sufficient sample collected
21: **return** class and field utilization

5.2.1 MicroProf workflow. Algorithm 1 presents the pseudo-code of MicroProf's detection of unnecessary data transfer. As the microservice receives a remote object over the communication APIs, the object gets allocated in the heap memory. On line 2, MicroProf intercepts these allocations of the remote objects (Section 6.1). At the allocation interception, MicroProf randomly selects a subset of the remote objects and a subset of their fields for further monitoring (Lines 3-4). For data-centric attribution, MicroProf further records the start memory address of the object allocation, the size of the allocated object, selected field offsets in the allocation range, and the `allocation context` of the selected object (Lines 5-6). On line 7, MicroProf sets traps to detect references to the selected fields (Section 6.2). Any reference to the field will cause a trap, and MicroProf intercepts the reference. During the access interception, MicroProf checks if the microservice code accessed the referenced field (Lines 9-10). If that is the case, MicroProf marks the field of the associated class used in the allocation context and increments the associated field access counter. Then MicroProf removes the trap from the field

(Lines 11-12). At the end of the life of the object, MicroProf identifies any remaining un-intercepted fields as unused fields. MicroProf increments the associated field's unused count in the allocation context and removes the trap on the field (Lines 16-19). MicroProf repeats the process until it monitors a sufficient number of samples in the allocation context (Line 20).

During the post-processing of the collected samples (Line 21), MicroProf calculates two utilization metrics on each allocation context to detect unnecessary data transfer: (1) Class utilization: What fraction of the class fields are utilized in the allocation context? (2) Field utilization: What fraction of a given field (say f) of the monitored remote objects are accessed by the requesting microservice at the allocation context? Equations 2 and 3 show the class utilization metric, ϕ and field utilization metric, ρ calculation respectively. In these equations, N_{ir} denotes the number of intercepted references to the class, N_t represents the number of traps set in the allocation context, N_r and N_r' stands for the number of references and non-references to f_i, respectively.

$$\phi = \frac{N_{ir}}{N_t} \tag{2}$$

$$\rho_i = \frac{N_r}{N_r + N_r'} \tag{3}$$

MicroProf sorts the allocation contexts by the allocation count and the class utilization metric, ϕ, in descending order. MicroProf finally reports the allocation context, class utilization, ϕ, and class field utilization, ρ. An allocation context with a large allocation count and low-class utilization is susceptible to unnecessary data transfer. Furthermore, field level utilization, ρ, identifies the underutilized fields.

Finally, we calculate the required number of samples for an acceptable confidence interval. We ask - what is the maximum number of samples per field that should be monitored to determine the field utilization, given a 95% confidence interval and a margin of error of 5%? Since this is a statistical point estimation problem with unknown population size, we can calculate the required sample size using the formula for infinite population size presented in [50] (Chapter 4.1.4), as demonstrated in equation 4. Here, z-score indicates how many standard deviations a data point is away from the mean of a normal distribution. For a 95% confidence interval, the z-score is 1.96. Given the unknown proportion p_u, we set it to 0.5 to generate a conservative variance estimate. Considering a margin of error (α) of 0.05 solving the formula gives a sample size (n) of 385.

$$n = (\frac{z - score}{\alpha})^2 \times p_u \times (1 - p_u.) \tag{4}$$

6 IMPLEMENTATION

MicroProf is implemented as a user-space tool to detect unnecessary data transfer in Java-based microservice applications. At the implementation level, it leverages the processor's hardware debug registers. MicroProf does not require any modification to the underlying JVM or the hardware. Fig. 3 shows the overview of MicroProf. MicroProf implements two agents: 1) a Java agent to intercept remote object allocation and 2) a JVMTI agent to set traps and intercept access to the fields. During the deployment of the microservice, MicroProf first attaches itself to the application. The agents are loaded into the same memory along with the target microservice application. MicroProf's Java agent instruments the remote object allocation regions. During a remote object allocation, MicroProf's Java agent intercepts and samples target objects. Based on the availability of debug registers, MicroProf samples all the objects of the class. MicroProf's JVMTI agent sets traps for the target fields of the object. During microservice execution, if the application accesses the target field, it causes a trap, and the JVMTI intercepts and records the access. When a sufficient number of samples are collected, MicroProf performs post-processing and reports the utilization metric.

6.1 Intercepting remote object allocation

Challenge: To enable microservice communications, a number of communication protocols have been proposed. Among these, RPC [16] [24] and RESTFul are two popular communication protocols. However, there are various service implementations of these communication protocols, such as Dubbo/gRPC service, Spring Cloud RESTFul service, or Kubernetes service. One approach to intercepting the incoming remote objects through these various communication protocol implementations is to write the driver code for each. With such an approach, one can monitor all the incoming remote objects without porting MicroProf's interceptor at the application level. However, due to a large number of protocol implementations, writing driver code for all the protocols is a challenging task. As an alternative approach, MicroProf exposes APIs for the developers to provide hints for remote invocations.

6.1.1 Annotation API. Listing 2 shows MicroProf's annotation API to guide the tool to monitor the remote object allocation. First, a developer annotates the functions that perform remote invocation to other services. The developer further provides the name of the remote object class as a parameter. This information helps the Java agent to instrument the target class and intercept the remote object allocation in the same context as the annotated function call.

```
------------ ClassName.java ------------%
@MICROPROFAnno(monitorObject="RemoteClass")
public RemoteClass remoteInvocation()
{
    ...
    List<RemoteClass> remoteObject= remoteClient.getRemoteObject();
    ...
}
```

Listing 2. MicroProf remote invocation annotation

6.1.2 The Java Agent. The Java agent instruments the target class at the bytecode level by leveraging the `java.lang.instrument` and `ASM` library [3]. During instrumentation, the Java agent registers a callback function for the allocation of the target class object. When the microservice receives a remote object, the object gets allocated in the heap memory. After heap allocation, the Java agent's callback function is invoked. The Java agent performs a random sampling to determine whether MicroProf will further monitor the allocated object. If the Java agent decides to monitor the object, it collects two pieces of information. 1) The start address of the object in the heap memory and 2) the list of fields and their offsets within the allocation region. The Java agent then passes this information to the JVMTI agent for setting traps on the object fields.

6.1.3 The JVMTI Agent. Once the Java agent hands over the target object to the JVMTI agent, MicroProf captures the program execution context of the object allocation. The JVMTI agent collects the program execution context as a dynamic call graph, C. JVMTI agent then randomly selects an object field, ${f_O}^i$, to monitor. MicroProf records the attribution of the object field, ${f_O}^i$ and the allocation context, C as a pair $< C, {f_O}^i >$. JVMTI agent then leverages a hardware debug register to set a trap on the chosen field ${f_O}^i$.

6.2 Intercepting field reference

Challenge: The modern processors implement a limited number of hardware debug registers, creating a challenge in monitoring many remote object fields concurrently. For instance, in our experimental environment, the servers have only four hardware debug registers per CPU core. Monitoring more than four remote object fields will require replacing the existing traps set by the debug registers. However, if MicroProf replaces an existing trap

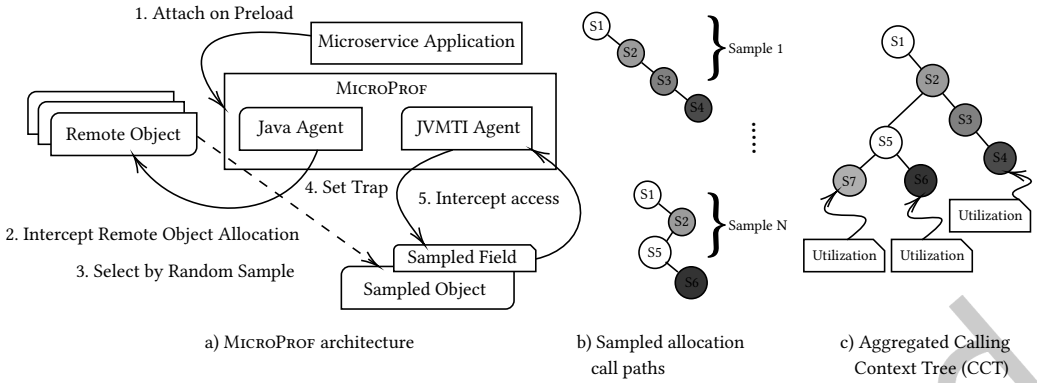

Fig. 3. Overview of MicroProf architecture and allocation path profiling

to monitor new fields, it will lose the observability of the replaced field. For completeness, it is important to monitor the object fields throughout the object's lifetime.

Since microservices are stateless in nature, any remote object allocated during a microservice invocation has a short lifespan. Therefore, when the service completes the request, the remote object becomes dead and waits for the garbage collector to collect it. Due to the short lifespan of remote objects, the JVMTI agent sets a trap for the target fields for a short period of time by setting a time-to-live (TTL) timer. This TTL approximates the lifetime of the object. If the microservice application refers to the field within this interval, the debug register will cause a trap and notify the JVMTI agent with a registered trap handler.

At the trap handler, the JVMTI agent identifies the pair $< C, fo^i >$ associated with the trap and increments the use counter of the field f^i at the allocation context C. At the same time, JVMTI makes the debug register available to monitor a new object field. If the microservice does not refer to the target field within the TTL period; the JVMTI agent checks if the object is dead at TTL expiration (Section 6.3). If the object is still alive, the JVMTI agent replaces the trap to monitor a new field.

6.3 Finding dead objects

Challenge: Java garbage collectors may or may not reclaim the free objects immediately. As a result, MicroProf cannot rely on the JVMTI garbage collection event to monitor the liveliness of an object.

JVM manages the reference of an object in a reference graph. Any alive heap object has a reference and thus is reachable from the top-level objects of the reference graph. MicroProf's JVMTI agent traverses this reference graph and performs a reachability analysis to identify if the heap object is reachable. For this purpose, during the object's allocation, the JVMTI agent tags the target object with a unique identifier using JVMTI `SetTag` API. When TTL expires, the JVMTI agent looks for the tag among all the reachable heap objects during reachability analysis using JVMTI `FollowReferences` API. If the agent cannot locate the target object using the unique identifier, it determines the object is dead. At this point, the JVMTI agent marks the expired field as unused and increments the unused counter of the field f^i at the allocation context C.

6.4 Post-processing

At the post-processing stage, MicroProf constructs an aggregated calling context tree of the sampled allocation contexts as shown in Fig. 3-c. Subsequently, it calculates the field and class utilization metrics for each allocation context. MicroProf further retrieves the source code line number and file name from the allocation context. Finally, it sorts the allocation context by the allocation count and utilization metrics and generates a report.

Table 1. Summary of Microservice applications evaluated in this paper. Latency is reported in milliseconds.

Real-world microservices	Version	# of services	Faulty microservice	Inefficient endpoint	Avg. latency original code (ms)	Avg. latency optimized code (ms)	Speedup (times)
TrainTicket	0.0.3	41	ts-seat-service	POST {/orderOther/tickets}	1860.86	405.49	**4.59**
			ts-route-plan-service	POST {/routePlan/quickestRoute}	13256.08	10608.05	**1.25**
Thingsboard	3.4.1	4	rule-engine	query to PostgreSQL DB	0.16	0.13	**1.20**
DayTrader	4.0.18	5	daytrader-portfolios	POST {/portfolios/{userId}/orders}	56.10	55.17	**1.02**
Eclipse Kapua	1.6.7	7	kapua-datastore	query to Elasticsearch	394.71	212.74	**1.86**

7 EVALUATION

We evaluate both the utility and impact of MicroProf. The evaluation aims to address the following questions:

- **Q1 (Detection):** Does the tool effectively identify unnecessary data transfer and pinpoint the responsible source code, along with its calling context? Additionally, does the tool report unutilized fields to offer guidance for optimization purposes?
- **Q2 (Speedup):** How much performance improvement can we anticipate through the utilization of MicroProf-guided optimization?
- **Q3 (Overhead):** What is the level of overhead incurred by MicroProf in its default setting?

Experimental setup. We evaluate MicroProf on clusters provisioned within CloudLab [48]. We choose CloudLab-Utah's m510 nodes while deploying the clusters [10]. Each m510 node is an 8-core, 16-thread Intel Xeon D-1548 (Broadwell) CPU clocked at 2.0GHz running Ubuntu 18.04. The device has a 12MB LLC cache and 64GB ECC Memory.

We study four microservice applications: TrainTicket, Daytrader, ThingsBoard, and Eclipse-Kapua. The applications are deployed on Kubernetes [31] clusters with 10, 3, 2, and 2 nodes, respectively. We configure the Kubernetes security context to privileged mode to enable the debug registers inside the Kubernetes pod. In the ThingsBoard and Eclipse-Kapua deployment, the database runs on a separate node from the backend services. Since MicroProf is compatible with JDK 11 and any of its successors, we update the build environments of the microservice applications to use JDK 11.

Evaluation methodology. Before evaluating MicroProf, we perform critical path analysis (Section 5.1) to prioritize queries that are deemed worthy of MicroProf's fine-grained analysis. For this purpose, we run the microservice applications and conduct application-specific operations on the client. These operations include booking tickets on TrainTicket microservice and trading stocks on DayTrader. Once we shortlist the queries, we start experiments to evaluate MicroProf against those queries. Each experiment precedes a 2-minute warm-up. We use Apache JMeter scripts with 16 user threads to request the TrainTicket and DayTrader microservices. For the other services, we use single-user settings. In each experiment, we send 10,000 queries to each of the services. To ensure the reliability of the results, we repeat each experiment five times. While reporting the results, we average the performance metrics across multiple experimental runs. We consider both the average and 99th-percentile latency as the performance metrics.

Summary of evaluation. Table 1 summarizes the microservice inefficiencies detected by MicroProf. Guided by MicroProf, we identify five inefficient data transfers in the microservice applications. To validate the findings, we further optimize the applications. Our experimental evaluation shows that microservices enjoy up to 4.59× speedup in average latency.

Fig. 4 further presents the latency distribution of the queries for each inefficient and optimized service, including the tail (99th percentile). From Fig. 4-a, the `ts-seat-service` (TrainTicket) latency is reduced from

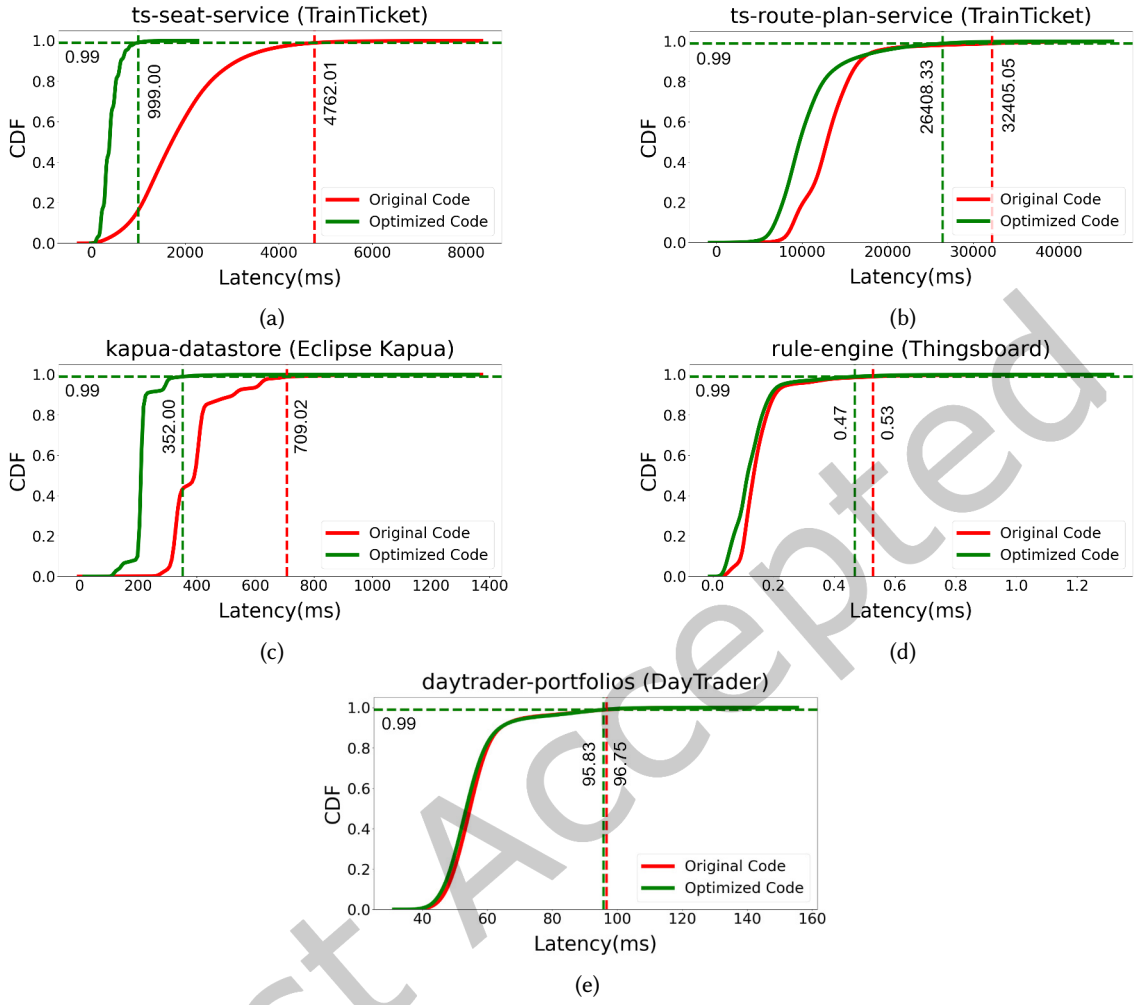

Fig. 4. The cumulative distribution function (CDF) of query latency before and after optimization of unnecessary data transfer.

4762.01 to 999 milliseconds through optimization, achieving a 4.77× improvement. In Fig. 4-b, the latency for the `ts-route-plan-service` (TrainTicket) is cut from 32405.05 to 26408.33 milliseconds, resulting in a 1.23× speedup. Similarly, Figs. 4-c, 4-d, and 4-e indicate that `kapua-datastore` service (Eclipse Kapua), `rule-engine` service (Thingsboard), and `daytrader-portfolios` service (DayTrader) experience an improvement in tail latency of 2.01×, 1.13×, and 1.01×, respectively. This result directly answers evaluation question **Q2**, showcasing the effectiveness of MicroProf in significantly enhancing microservice performance.

Overhead measurement. To answer the evaluation question **Q3**, we measure the runtime overhead of MicroProf on 14 services from the TrainTicket benchmark. These selected services represent the core microservices frequently invoked by the rest of the services in the benchmark. We carry out this assessment by executing each experiment five times with 16 user threads and then comparing the runtime ratio with and without MicroProf

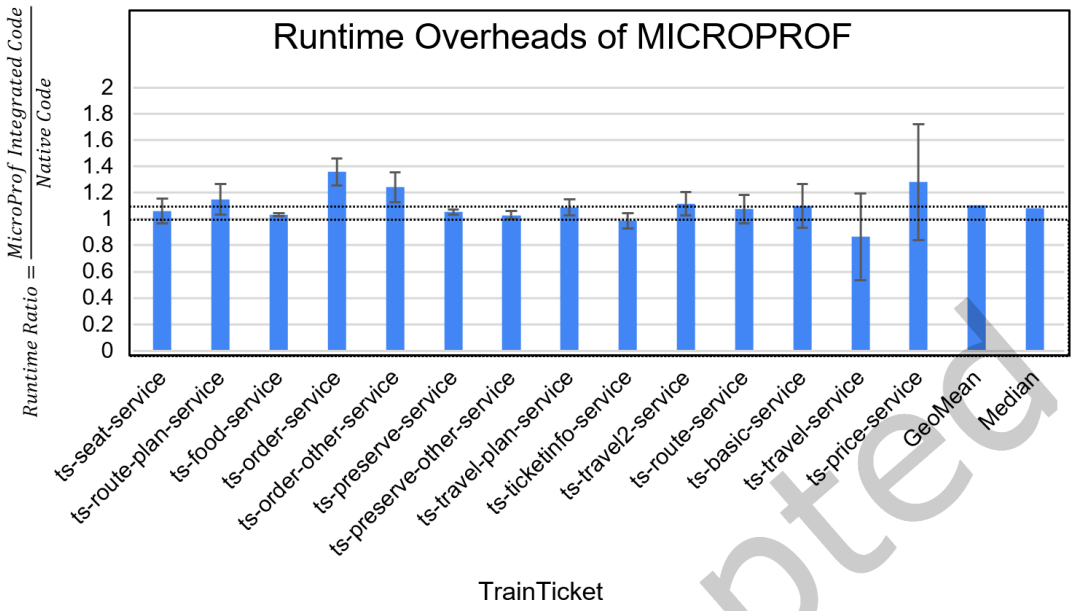

Fig. 5. Runtime overhead.

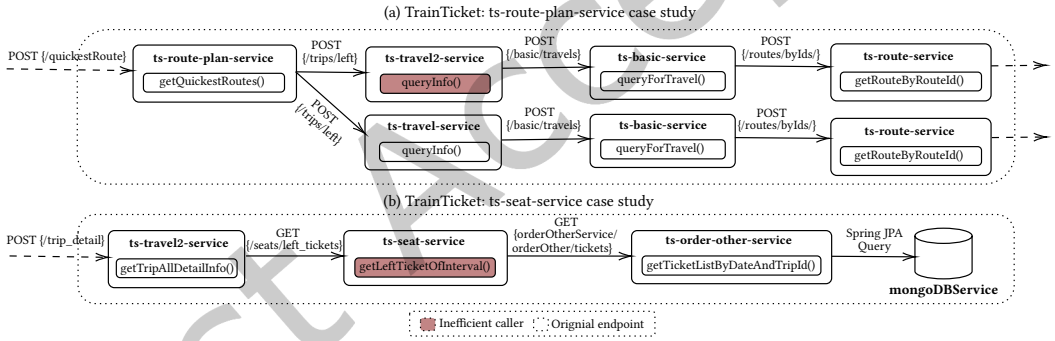

Fig. 6. REST communication between TrainTicket microservices. Inefficient microservices are shown in gray.

monitoring enabled. The findings are depicted in Fig. 5. The figure shows that most services experience a maximum overhead of 10%. To further minimize sampling overhead, developers can configure MicroProf's dynamic sampling rate selection. A similar approach is adopted by Uber's CRISP [68] and Jaeger.

7.1 Case Studies

To answer the evaluation question **Q1**, this section discusses the utility of MicroProf in greater detail. Through individual case studies, we demonstrate the unnecessary communication patterns in the microservice applications' source code. We further demonstrate MicroProf's capability to identify the source code's inefficiency accurately. Finally, guided by MicroProf, we optimize the unnecessary data transfer and verify the optimization by systematically comparing the original implementation.

Table 2. Summary of critical path analysis report. Latency is reported in seconds.

Rank	Critical path	Normalized exclusive latency (s)	Normalized inclusive latency (s)	Distance
1	TRAVEL-PLAN[getByQuickest]->TRAVEL-PLAN[POST]	1.00	0.95	1.38
2	TRAVEL-PLAN[POST]	1.00	0.88	1.33
3	TRAVEL-PLAN[getByQuickest]	0.01	1.00	1.00
4	TRAVEL-PLAN[getByQuickest]->TRAVEL-PLAN[POST]->ROUTE-PLAN[getQuickestRoutes]	0.01	0.32	0.32
5	TRAVEL-PLAN[POST]->ROUTE-PLAN[getQuickestRoutes]	0.01	0.32	0.32
6	...	...	...	...

7.1.1 TrainTicket. TrainTicket is an open-source microservice benchmark application. It implements a train ticket booking system with a total of 41 services. The services communicate with other services through REST API. The majority of the services of TrainTicket are written in Java using Spring Boot framework [56]. It offers versatile deployment facilities consisting of Kubernetes, Helm, and Docker and supports distributed tracing using Jaeger. We deploy TrainTicket-0.0.3 in a CloudLab Kubernetes cluster of 10 nodes.

Inefficiency in `ts-route-plan-service`

To identify the significant performance overhead, we first perform critical path analysis (CPA). Table 2 presents the first few lines reported by the CPA on the TrainTicket microservice. We evaluate each path according to the priority list using MicroProf and identify an inefficiency on the path of row 5.

The Scenario. The scenario is depicted in Fig. 6-a. The TrainTicket webpage searches for the quickest Route with a source and destination station. The request is sent to `ts-route-plan-service` using REST API `POST "/quickestRoute"`. The request handler at `ts-route-plan-service` handles the request by invoking REST APIs of two remote services, `ts-travel-service` and `ts-travel2-service`. These requests return high-speed and normal-speed train routes. The `ts-route-plan-service` sends the five quickest routes to display on the webpage. MicroProf identifies unnecessary data transfer from `ts-travel2-service` to `ts-route-plan-service`.

MicroProf insight Listing 3 shows MicroProf's report on two remote invocations from `ts-route-plan-service`. The first half of the listing shows that `ts-route-plan-service` receives the normal train routes from `ts-travel2-service`. The second half of the listing shows that `ts-route-plan-service` receives the high-speed train routes from `ts-travel-service`. Both `ts-travel2-service` and `ts-travel-service` return these routes as a list of `TripResponse` instances. However, the utilization report shows that the remote objects returned from `ts-travel2-service` suffer low utilization of 23%. Comparatively, the remote objects from `ts-travel-service` have higher utilization of 83.7%.

Listing 4 shows the partial code of the handler function `searchQuickestResult`. Lines 2 and 3 show the two remote invocations for high-speed train routes and normal-speed train routes. However, the code only returns the quickest five results. Manual code investigation shows that in most cases, these results are from high-speed train routes. As a result, returned `TripResponse` objects from the normal train routes become unnecessary.

The optimization. We optimize the `ts-route-plan-service` by conditionally avoiding the unnecessary transfer of normal train routes. After optimization, the service observes a 76.1% reduction in the number of fields requested and enjoys 1.25× speedup in average latency.

Inefficiency in `ts-seat-service`

The Scenario Fig. 6-b depicts the scenario of service invocations. The `ts-travel2-service` requests the `ts-seat-service` for a list of remaining tickets using an exposed REST API, `GET "/seats/left_tickets"`. `ts-seat-service`'s `getLeftTicket-OfInterval` implements the handler function to serve the request. For this purpose, `ts-seat-service` communicates with another external service `ts-order-other-service` through

```
% ---------- Context for normal train
...
plan.controller.RoutePlanController.getQuickestRoutes(RoutePlanController.java:39)
|_ plan.service.RoutePlanServiceImpl.searchQuickestResult(RoutePlanServiceImpl.java:98)
  |_ plan.service.RoutePlanServiceImpl.@getTripFromNormalTrainTravelService@(RoutePlanServiceImpl.java:329)
...
Field utilization: {'endTime': 100%,'startingTime': 100%,'confortClass': 0%,'economyClass': 0%,'
    ↪ priceForConfortClass': 0%,'priceForEconomyClass': 0%,'startingStation': 0%,'terminalStation': 0%,'
    ↪ trainTypeId': 0%,'tripId': 0%},
Class Utilization: {'TripResponse': @23.9%@}

% ---------- Context for high speed train
...
plan.controller.RoutePlanController.getQuickestRoutes(RoutePlanController.java:39)
|_ plan.service.RoutePlanServiceImpl.searchQuickestResult(RoutePlanServiceImpl.java:97)
  |_ plan.service.RoutePlanServiceImpl. #getTripFromHighSpeedTravelServive#(RoutePlanServiceImpl.java:313)
...

Field utilization: {'endTime':99%,'startingTime':100%,'confortClass':0%,'economyClass':0%,'
    ↪ priceForConfortClass': 100%,'priceForEconomyClass':100%,'startingStation':100%,'terminalStation'
    ↪ :100%,'trainTypeId':100%,'tripId':100%},
Class Utilization: {'TripResponse': #83.7%#}
```

Listing 3. MicroProf reports unnecessary data transfer from `ts-travel2-service` to `ts-route-plan-service` in TrainTicket. However, MicroProf did not find inefficiency in communication between `ts-travel-service` to `ts-route-plan-service`

```
public Response searchQuickestResult(...) {
    ArrayList<TripResponse> highSpeed = getTripFromHighSpeedTravelServive(queryInfo, headers);
    ArrayList<TripResponse> normalTrain = getTripFromNormalTrainTravelService(queryInfo, headers);
...
    int size = Math.min(finalResult.size(), 5);
    for (int i = 0; i < size; i++) {
        ...
        returnResult.add(finalResult.get(minIndex));
    }
...
```

Listing 4. TrainTicket's ts-route-plan-service.

```
...
seat.controller.SeatController.getLeftTicketOfInterval(SeatController.java:52)
|_ seat.service.SeatServiceImpl.getLeftTicketOfInterval(SeatServiceImpl.java:238)
  |_ seat.service.SeatServiceImpl.invokeOrderOtherTickets(SeatServiceImpl.java:301)
...
Field utilization: {'destStation': 100% 'seatNo': 0%, 'startStation': 0%},
Class Utilization: {'Ticket': 33.9%}
```

Listing 5. MicroProf reports unnecessary data transfer from `ts-order-other-service` to `ts-seat-service` in TrainTicket.

REST API `GET "/orderOther/tickets"`. Finally, `ts-order-other-service` retrieves the requested data from `mongoDBService`. MicroProf identifies unnecessary response data transfer from `ts-order-other-service` to `ts-seat-service`.

MicroProf insight. Listing 5 shows the snapshot of MicroProf's code analysis result for `ts-seat-service` microservice of TrainTicket. Lines 1-5 show the truncated allocation context of the unnecessary remote object. The allocation context represents the call path, source code file name, and line number. For instance, the context identifies the inefficient remote object is requested on Line 301 of `SeatServiceImpl.java`. Line 6 of the listing delineates the utilization of each field of the class, and Line 7 delineates the utilization of that entity class. From

the listing, MicroProf reports that the class `Ticket` has a low utilization of 33.9%. The report further states that only field `destStation` has full utilization, whereas the fields `seatNo` and `startStaion` are never used.

```
public Response getLeftTicketOfInterval(...) {
...
-  re3 = restTemplate.exchange("http://ts-order-other-service:12032/api/v1/
-    orderOtherService/orderOther/tickets", ...});
-  leftTicketInfo = re3.getBody().getData();
-  Set<Ticket> soldTickets = leftTicketInfo. getSoldTickets();
-  for (Ticket soldTicket : soldTickets) {
-      String soldTicketDestStation = soldTicket.getDestStation();
+  re3 = restTemplate.exchange("http://ts-order-other-service:12032/api/v1/
+        orderOtherService/orderOther/tickets/destStation/list", ...});
+  List<String> dStations = re3.getBody().getData();
+  for (String soldTicketDestStation : dStations) {
...
```

Listing 6. TrainTicket ts-seat-service before and after optimization.

Listing 6 shows the inefficient original code snippet as well as the optimized code of the `ts-seat-service`. The original code snippet is marked as red, and the optimized code snippet is marked as green. In the original code, the `getLeftTicketOfInterval()` function of the `ts-seat-service` serves the request by providing the number of remaining tickets. For this purpose, the function invokes a REST API of `ts-order-other-service`. `ts-order-other-service` returns a list of `Ticket` entity. However, as MicroProf reports, the function only utilizes the `destStation` field of the class by calling the getter function `getDestStation()`.

The optimization. Our implementation focuses on reducing redundant data transfer over the network. We implement a new optimized endpoint at the `ts-order-other-service` that retrieves only the `destStation` list from the MongoDB service and sends the response back to `ts-seat-service`. This optimization reduces the original implementation's transferred data to $\frac{1}{3}$. We conduct stress tests on inefficient and optimized code blocks to analyze performance improvement. We send a bulk of requests using Apache JMeter and measure the average latency of the `GET "/seats/left_tickets"` request. Guided by MicroProf, the optimization of the `ts-seat-service` reduces the API response size by 66.1% in terms of the number of fields and results in a 4.59× speedup in average latency.

7.1.2 ***ThingsBoard.*** ThingsBoard is an open-source IoT platform that enables developing and managing robust, scalable, and fault-tolerant IoT projects. The platform provides server-side infrastructure for IoT applications with transports, core, rules, and a web-based user interface. We deploy Thingsboard v3.4.1 using Docker in a cluster of 2 nodes.

The Scenario. ThingsBoard provides a rule engine, a framework for building event-based workflow. We implement a Thermostat rule chain to experiment using ThingBoard's rule engine. Fig. 7 depicts the rule chain. According to the rule chain, the related devices will be notified when the thermostat device posts telemetry. While testing the scenario, MicroProf identifies unnecessary data transfer from the database to the rule engine.

MicroProf insight. Listing 7 shows MicroProf's code analysis result for the `rule-engine` service of the ThingsBoard application. From the reported context, the inefficiency happens in `EntitiesRelatedDeviceIdAsyncLoader.java` file at `findDeviceAsync()` function. Line 6 of the listing further reports that `Device` class has a low utilization of 12.5%, and only `id` and `type` fields have full utilization. The function does not require access to the remaining 13 fields.

```
rule.metadata.TbAbstractGetAttributesNode.onMsg(TbAbstractGetAttributesNode.java:74)
|_ rule.metadata..TbGetDeviceAttrNode.findEntityIdAsync(TbGetDeviceAttrNode.java:50)
  |_ rule.util.EntitiesRelatedDeviceIdAsyncLoader.findDeviceAsync(EntitiesRelatedDeviceIdAsyncLoader.java
       ↪ :40)

Field utilization: {'id': 100%, 'type': 100%, 'additionalInfo': 0%, 'additionalInfoBytes': 0%, 'createdTime
    ↪ ': 0%, 'customerId': 0%, 'deviceData': 0%, 'deviceDataBytes': 0%, 'deviceProfileId': 0%, '
    ↪ externalId': 0%, 'firmwareId': 0%, 'name': 0%, 'softwareId': 0%, 'tenantId': 0%, 'label': 0%},
Class Utilization: {'Device': 12.5%}
```

Listing 7. MicroProf reports unnecessary data transfer from PostgreSQL DB to Rule-Engine service of ThingsBoard.

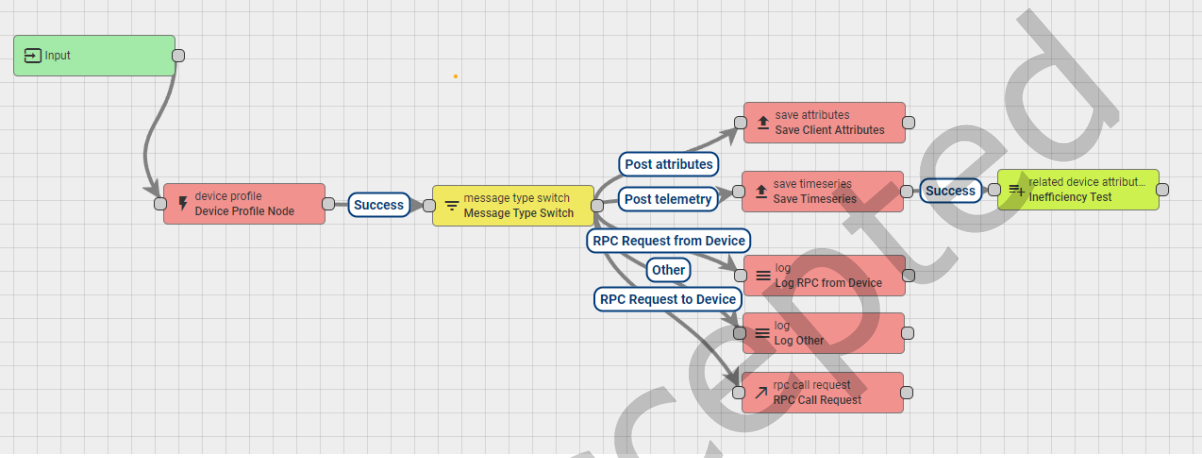

Fig. 7. ThingsBoard rule-chain.

```
public static ListenableFuture<DeviceId> findDeviceAsync(
    TbContext ctx, EntityId originator, DeviceRelationsQuery deviceRelationsQuery){
...
-  ListenableFuture<List<Device>> asyncDevices =
-  deviceService.findDevicesByQuery(ctx.getTenantId(), query);
-  return Futures.transformAsync(asyncDevices, d ->
-  CollectionUtils.isNotEmpty(d) ?
-  Futures.immediateFuture(d.get(0).getId()): Futures.immediateFuture(null),
-  MoreExecutors.directExecutor());
+  ListenableFuture<List<DeviceId>> asyncDevices =
+  deviceService.findDevicesIdByQuery(ctx.getTenantId(), query);
+  return Futures.transformAsync(asyncDevices, d ->
+   CollectionUtils.isNotEmpty(d) ? Futures.immediateFuture(d.get(0))
+       : Futures.immediateFuture(null), MoreExecutors.directExecutor());
```

Listing 8. Thingsboard Rule-Engine service before and after optimization.

Listing 8 shows the original code of interest and the optimization. Lines 5-7 show that the function retrieves the list of `Device` instances. However, lines 8-13 show that the `id` field is returned from the function. As a result, retrieving the other fields from the backend database is unnecessary.

The optimization. Guided by MicroProf, we optimize the code by only querying for the device id. The optimization reduces the response size by 87.5% in terms of the number of fields and improves the runtime by 1.20× on average.

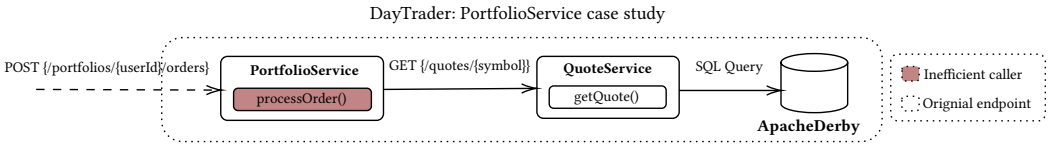

Fig. 8. REST communication between DayTrader microservices. Inefficient microservices are shown in gray.

```
daytrader.controller.PortfoliosController.processOrder(PortfoliosController.java:268 )
|_ daytrader.service.PortfoliosService.buy(PortfoliosService.java:578 )

Field utilization: {'price': 100% 'symbol': 100%, 'volume': 15%, 'change1': 16%, 'open1': 9%, 'companyName'
    ↪ : 0%, 'low': 0%, 'volume': 0%, 'change1': 0%, 'high': 0%, 'open1': 0%
Class Utilization: {'QuoteDataBean': 31.7%}
```

Listing 9. MicroProf reports unnecessary data transfer from PortfoliosService to Apache Derby in DayTrader.

```
public OrderDataBean buy(...) {
...
-  QuoteDataBean quoteData = quotesService.getQuote(symbol);
-    ...
+  BigDecimal QuotePrice = quotesService.getQuotePrice(symbol);
   createOrder(...);
...
}
private OrderDataBean createOrder(...){
...
stmt.setBigDecimal(6, quoteData.getPrice()...
stmt.setString(10, quoteData.getSymbol());
...
}
```

Listing 10. DayTrader Portfolios service before and after optimization.

7.1.3 DayTrader. DayTrader is an open-source benchmark application that simulates online stock trading. It was originally developed by IBM for WebSphere Trade. In the paper, we evaluate the microservice version of the application [45], which consists of 5 microservices.

The scenario. Fig. 8 illustrates DayTrader service invocation details in our study. The `PortfolioService` receives a buy request from the end user and sends a remote invocation to `QuoteService` via endpoint `GET "/quotes/symbol"`. `QuoteService` then returns a `QuoteDataBean` object to `PortfolioService`. MicroProf identifies unnecessary data transfer in the scenario.

MicroProf insight. Listing 9 depicts the inefficiency report of MicroProf on the `PortfolioService`. From the call path, `PortfoliosService.buy()` function is the source of the unnecessary data transfer. The listing further shows that the received `QuoteDataBean` remote objects have low utilization of 31.7%. MicroProf further reports that only `price` and `symbol` fields have full utilization.

Listing 10 represents the original code snippet as well as optimized code for the buy function. On Line 4, the function makes a remote invocation to `QuoteService` to return `QuoteDataBean`. The function then passes the quote to `createOrder` function. However, `createOrder` only accesses the `price` and `symbol` fields using the getter function. As a result, the other field transfer is unnecessary. We further identify that transferring `symbol` is also unnecessary since buy queries the quote using `symbol`.

Interestingly, MicroProf reports some usage of fields `volume`, `change1`, and `open1`. Further investigation shows that they are accessed during a stack dump due to a service exception.

```
@XmlType(propOrder = {
        "id", "datastoreId", "timestamp", "deviceId", "clientId", "receivedOn", "sentOn", "capturedOn", "
            ↪ position", "channel", "payload"
})
public <T> ResultList<T> query(...) {
...
-  request.setJsonEntity(json);
-  Response queryResponse = restCallTimeoutHandler(() -> getClient().
-         performRequest(request), typeDescriptor.getIndex(), "QUERY");
-  DatastoreMessage messageToBeDeleted = find(scopeId, id, StorableFetchStyle.FIELDS);
-  schemaMetadata = mediator.getMetadata(scopeId, messageToBeDeleted.getTimestamp().getTime());
+  String includeQuery=String.format("includes\":[\"%s\"]",field);
+  String queryToBeReplaced = "includes\":[\"*\"]";
+  json = json.replace(queryToBeReplaced,includeQuery);
+  request.setJsonEntity(json);
+  Response queryResponse = restCallTimeoutHandler(() -> getClient().
+          performRequest(request), typeDescriptor.getIndex(), "QUERY");
+  Date timestamp = findModified(scopeId, id, StorableFetchStyle.FIELDS);
+  schemaMetadata = mediator.getMetadata(scopeId, timestamp.getTime());
```

Listing 11. Eclipse Kapua Datastore service before and after optimization.

The optimization. In order to reduce unnecessary data transfer, we implement a new REST API requesting for Quote Price only. The optimization reduces the response size by 68.3% in terms of the number of fields and results in a 1.016× speedup in terms of average latency.

7.1.4 Eclipse Kapua. Eclipse Kapua is an open-source platform for integrating IoT devices and smart sensors. It manages and integrates devices and their data and provides a solid foundation for IoT services for any IoT application. Eclipse Kapua consists of 7 microservices. In our experiment, we deploy Kapua v1.6.3 in a 2-node CloudLab cluster.

The Scenario. The Eclipse Kapua performs bulk deletion of `DatastoreMessage` while performing an application integration test. Cucumber scripts initially send the request to `MessageStoreServiceImpl`. Then it forwards the request to Elasticsearch over `MessageStoreFacade`. MicroProf detects an inefficiency while Kapua performs a remote request to Elasticsearch during the test.

MicroProf insight. Listing 11 represents both the original and the optimized code. Lines 19-23 show that the original code calls the `find()` function. The `find()` function then invokes a remote elastic search service to retrieve the `DatastoreMessage` object. However, line 27 shows that it only accesses `timestamp` field via the getter function. MicroProf accurately detects this inefficiency.

The optimization. To eliminate the unnecessary data transfer over the network, we implement a customized Elasticsearch client query that only retrieves the `timestamp` instead of the whole `DatastoreMessage` object. We further optimize the Elasticsearch server to respond only to the `timestamp` field. This optimization resulted in a 1.88× speedup.

8 THREATS TO VALIDITY

Internal threats to validity. Our experiment demonstrates that a 95% confidence interval is effective in detecting both used and unused fields in objects, although the tool may miss some cases due to sampling. Additionally, it's worth noting that in various contexts and with different inputs, objects may still be accessed after the TTL expiration. Our evaluation of overhead focuses solely on the 14 microservices of the TrainTicket benchmark.

External threats to validity. Since open-source Java microservice applications are not widely available, we are only able to evaluate MicroProf on a limited number of applications. Proprietary applications, on the other hand, can be more intricate and employ a variety of frameworks rather than just Spring Boot. Due to the lack of ground truth data, we only manually evaluated the MicroProf reported inefficiencies.

Construct threats to validity. The manual annotation process can be both challenging and prone to human error, especially with a large number of RPC invocations. Missing or inaccurately identifying these invocations may compromise the tool's ability to accurately measure the underlying construct of RPC interaction, thus affecting the construct validity. In our future work, we intend to eliminate the annotation-based approach by implementing and/or leveraging monitoring capabilities directly within the RPC libraries.

Conclusion threats to validity. The end-to-end latency is often influenced by external factors such as network conditions, server load, and background garbage collection, creating conclusion threats to validity. To mitigate the issue, we carefully maintain the same experimental environment across experiments and use appropriate statistical methods to support the validity of our conclusions. We observe a significant variance reduction after optimization for all the cases. This reduction may be attributed to alleviating network load and garbage collection overhead resulting from the optimization of unnecessary data transfer.

9 CONCLUSIONS

In this paper, we propose MicroProf, a dynamic analysis tool to detect unnecessary data transfer in microservice applications. The tool pinpoints the inefficiency of the application source code. It further identifies the inefficient data structure fields. To demonstrate the utility, the paper evaluates MicroProf on four microservice applications. The tool detects five inefficiency patterns in the applications. Guided by MicroProf, we further optimize the applications and observe up to a 4.59× speedup. MicroProf will help the developer community to pinpoint the inherent inefficiencies in Java microservice communication and guide them toward optimization. Moving forward, we plan to evaluate our tool on proprietary applications with real-world experimental settings and explore its applicability to other programming languages.

REFERENCES

[1] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.

[2] Glenn Ammons, Thomas Ball, and James R Larus. 1997. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices* 32, 5 (1997), 85–96.

[3] ASM 2002. ASM. https://asm.ow2.io/. [Accessed 01-Sep-2022].

[4] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for request extraction and workload modelling.. In *OSDI*, Vol. 4. 18–18.

[5] Gleison Brito, Thais Mombach, and Marco Tulio Valente. 2019. Migrating to GraphQL: A practical assessment. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 140–150.

[6] Shuang Chen, Christina Delimitrou, and José F Martínez. 2019. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 107–120.

[7] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering*. 1001–1012.

[8] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. 2016. Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. *IEEE Transactions on Software Engineering* 42, 12 (2016), 1148–1161.

[9] Trishul M Chilimbi, Bob Davidson, and James R Larus. 1999. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. 13–24.

[10] CloudLabHardware 2014. Cloudlab Hardware. https://docs.cloudlab.us/hardware.html. [Accessed 01-Sep-2022].
[11] Oracle Corp. 2018. JVMTM Tool Interface. https://docs.oracle.com/en/java/javase/11/docs/specs/jvmti.html.
[12] DR 2005. Debug Registers. https://pdos.csail.mit.edu/6.828/2004/readings/i386/s12_02.htm. [Accessed 01-Sep-2022].
[13] Paul J Drongowski. 2007. Instruction-based sampling: A new performance analysis technique for AMD family 10h processors. *Advanced Micro Devices* (2007).
[14] EclipseKapua 2016. Eclipse Kapua. https://projects.eclipse.org/projects/iot.kapua. [Accessed 01-Sep-2022].
[15] Ev 2016. Evolution of microservices craft conference. https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference. [Accessed 01-Sep-2022].
[16] Finagle 2011. Finagle: An extensible RPC system for the JVM. https://twitter.github.io/finagle/. [Accessed 01-Sep-2022].
[17] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. 2007. {X-Trace}: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*.
[18] Kaihua Fu, Wei Zhang, Quan Chen, Deze Zeng, Xin Peng, Wenli Zheng, and Minyi Guo. 2021. Qos-aware and resource efficient microservice deployment in cloud-edge continuum. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 932–941.
[19] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.
[20] 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*. 19–33.
[21] Patric Genfer and Uwe Zdun. 2022. Avoiding Excessive Data Exposure Through Microservice APIs. In *Software Architecture: 16th European Conference, ECSA 2022, Prague, Czech Republic, September 19–23, 2022, Proceedings*. Springer, 3–18.
[22] Susan L Graham, Peter B Kessler, and Marshall K McKusick. 1982. Gprof: A call graph execution profiler. *ACM Sigplan Notices* 17, 6 (1982), 120–126.
[23] graphql 2022. Draft - GraphQL Specification. https://spec.graphql.org/draft/. [Accessed 01-Sep-2022].
[24] gRPC 2016. gRPC: A high performance open-source universal RPC framework. https://grpc.io. [Accessed 01-Sep-2022].
[25] Pao-Lu Hsu and Herbert Robbins. 1947. Complete convergence and the law of large numbers. *Proceedings of the national academy of sciences* 33, 2 (1947), 25–31.
[26] HTrace 2018. Apache HTrace. https://incubator.apache.org/projects/htrace.html. [Accessed 01-March-2023].
[27] IBM 2013. sample-daytrader. https://github.com/sample-daytrader. [Accessed 01-Sep-2022].
[28] Intel. 2022. VTune Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.
[29] jaeger 2017. Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/. [Accessed Mar-2023].
[30] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 152–166.
[31] kubernetes 2014. kubernetes. https://kubernetes.io/. [Accessed 01-Sep-2022].
[32] Nikita Lazarev, Neil Adit, Shaojie Xiang, Zhiru Zhang, and Christina Delimitrou. 2020. Dagger: Towards efficient rpcs in cloud microservices with near-memory reconfigurable nics. *IEEE Computer Architecture Letters* 19, 2 (2020), 134–138.
[33] Greg Leffler. 2022. {OpenTelemetry} and Observability: What, Why, and Why Now? (2022).
[34] John Levon. 2006. OProfile. *http://oprofile. sourceforge. net/* (2006).
[35] Bolun Li, Pengfei Su, Milind Chabbi, Shuyin Jiao, and Xu Liu. 2023. DJXPerf: Identifying Memory Inefficiencies via Object-Centric Profiling for Java. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO 2023)*. ACM, 81–94.
[36] Bolun Li, Hao Xu, Qidong Zhao, Pengfei Su, Milind Chabbi, Shuyin Jiao, and Xu Liu. 2022. OJXPerf: Featherlight Object Replica Detection for Java Programs. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. ACM, 1558–1570.
[37] Lightstep 2018. Lightstep. https://lightstep.com/. [Accessed 01-March-2023].
[38] Xu Liu and John Mellor-Crummey. 2011. Pinpointing data locality problems using data-centric analysis. In *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 171–180.
[39] Xu Liu and John Mellor-Crummey. 2013. A data-centric profiler for parallel programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12.
[40] Xu Liu, Kamal Sharma, and John Mellor-Crummey. 2014. ArrayTool: a lightweight profiler to guide array regrouping. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 405–416.
[41] 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 *Proceedings of the ACM Symposium on Cloud Computing*. 412–426.

[42] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 378–393.
[43] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. 1953. Equation of state calculations by fast computing machines. *The journal of chemical physics* 21, 6 (1953), 1087–1092.
[44] Netflix 2013. Announcing Ribbon: Tying the Netflix Mid-Tier Services Together. https://netflixtechblog.com/announcing-ribbon-tying-the-netflix-mid-tier-services-together-a89346910a62. [Accessed Sep-2022].
[45] Vikram Nitin, Shubhi Asthana, Baishakhi Ray, and Rahul Krishna. 2022. CARGO: AI-Guided Dependency Analysis for Migrating Monolithic Applications to Microservices Architecture. *arXiv preprint arXiv:2207.11784* (2022).
[46] OpenTelemetry 2022. OpenTelemetry. https://opentelemetry.io/docs/what-is-opentelemetry/. [Accessed March-2023].
[47] Tirthak Patel and Devesh Tiwari. 2020. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 193–206.
[48] Robert Ricci, Eric Eide, and CloudLab Team. 2014. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE* 39, 6 (2014), 36–38.
[49] Probir Roy and Xu Liu. 2016. StructSlim: A lightweight profiler to guide structure splitting. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 36–46.
[50] Thomas P Ryan. 2013. *Sample size determination and power*. John Wiley & Sons.
[51] Raja R Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H Sigelman, Rodrigo Fonseca, and Gregory R Ganger. 2016. Principled workflow-centric tracing of distributed systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 401–414.
[52] Yuri Shkuro. 2019. *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. Packt Publishing Ltd.
[53] Shopify 2022. Warn of over-fetching to GraphQL data to developers. https://github.com/Shopify/hydrogen-v1/pull/886. [Accessed Aug-2023].
[54] 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. (2010).
[55] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. 2018. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software* 146 (2018), 215–232.
[56] spring-boot 2022. spring-boot. https://spring.io/projects/spring-boot.
[57] Pengfei Su, Qingsen Wang, Milind Chabbi, and Xu Liu. 2019. Pinpointing performance inefficiencies in Java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 818–829.
[58] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandros Daglis. 2020. The NEBULA RPC-optimized architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 199–212.
[59] Nathan Russell Tallent. 2010. *Performance analysis for parallel programs from multicore to petascale*. Rice University.
[60] ThingsBoard 2016. ThingsBoard. https://thingsboard.io/. [Accessed 01-Sep-2022].
[61] Twitter 2013. Decomposing Twitter: Adventures in Service-Oriented Architecture. https://www.infoq.com/presentations/twitter-soa/. [Accessed 01-Sep-2022].
[62] Uber 2020. Introducing Domain-Oriented Microservice Architecture. https://www.uber.com/blog/microservice-architecture/. [Accessed 01-Sep-2022].
[63] Andrew Walker, Dipta Das, and Tomas Cerny. 2020. Automated code-smell detection in microservices through static analysis: A case study. *Applied Sciences* 10, 21 (2020), 7800.
[64] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. 2013. HOTL: a higher order theory of locality. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*. 343–356.
[65] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. Finding low-utility data structures. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 174–186.
[66] C-Q Yang and Barton P Miller. 1988. Critical path analysis for the execution of parallel and distributed programs. In *The 8th International Conference on Distributed*. IEEE Computer Society, 366–367.
[67] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. How not to structure your database-backed web applications: a study of performance bugs in the wild. In *Proceedings of the 40th International Conference on Software Engineering*. 800–810.
[68] 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 22)*. 655–672.
[69] Laiping Zhao, Yanan Yang, Kaixuan Zhang, Xiaobo Zhou, Tie Qiu, Keqiu Li, and Yungang Bao. 2020. Rhythm: component-distinguishable workload deployment in datacenters. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–17.

[70] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering* 47, 2 (2018), 243–260.
[71] Zipkin 2017. Zipkin. https://zipkin.io/. [Accessed 01-March-2023].