Draft – Please Do Not Share Or Distribute

Blackbox Request Tracing for Modern Cloud Applications

Sachin Ashok[1], Vipul Harsh[1], P. Brighten Godfrey[1], Radhika Mittal[1],
Srinivasan Parthasarathy[2], and Larisa Shwartz[2]

[1]University of Illinois at Urbana-Champaign
[2]IBM Research

Abstract

Monitoring and debugging modern cloud-based applications is challenging since even a single API call can involve many inter-dependent distributed microservices. To provide observability for such complex systems, distributed tracing frameworks track request flow across the microservice call tree. However, such solutions require instrumenting every component of the distributed application to add and propagate tracing headers, which has slowed adoption. This paper explores whether tracing requests can be achieved *without* any application instrumentation, which we refer to as request trace reconstruction. We present TraceWeaver, an optimization framework that incorporates readily available information from production settings (e.g., timestamps) and test environments (e.g., call graphs) to reconstruct request traces with usefully high accuracy of 90%. Evaluation with (1) benchmark microservice applications and (2) a production microservice dataset demonstrates high accuracy. We evaluate use cases for TraceWeaver, including A/B testing and finding performance outliers, showing effective results. Finally, we discuss potential future approaches which can aid in improving accuracy and ease of adoption of TraceWeaver.

1 Introduction

Modern "cloud native" applications built using a microservice architecture split their functionality into small logical units, which can be deployed as containers and automatically scaled (i.e., dynamically replicated to meet demand) with cluster management platforms like Kubernetes. In addition, compared to monolithic applications, individual microservices are more manageable to build and maintain, easier to individually redeploy in new versions, and easier to write in different languages and frameworks. The result is that modern applications are highly distributed, potentially involving hundreds or even over a thousand [34] individual microservice instances. Unfortunately, such highly distributed applications are challenging to operate and debug [12, 14]. For example, an operator may wish to determine which service is responsible for inflating latency for a certain user-facing API call for a specific subset of traffic. But an API call may produce *children*, i.e., further API calls to other services issued in service of the parent call. The resulting tree of API calls may ultimately comprise many stages of back end calls to various services, making it hard to isolate the component responsible for a particular problem. For such troubleshooting, it is indispensible to have a **request trace**, which we define as a record of each request (API call) into a microservice, and recursively all requests that it spawns (children, children of children, etc.) to other microservices. That is, each record within the trace includes a start and finish time (known as a "span") and pointers to its child request records.

Distributed tracing frameworks such as Jaeger [27] and Zipkin [43], along with commercial solutions like Instana [25] and Datadog [18] and earlier research proposals like XTrace [22], help developers deal with the above problem. Because network communication is externally visibile, it is easy for such frameworks to automatically log requests between microservices as isolated events. But to produce full request traces, such systems require application developers to instrument their code, carrying identifiers that associate each request with its children, which is known as *context propagation*. Even within a single service, instrumentation is needed at a module level to track execution flow, requiring a context object to be passed as an argument when modules are invoked [33].

This application-level code modification is required for request tracing, no matter what underlying framework is used by the application (service meshes, gRPC, etc). Only the application-level modules can track the execution flow (i.e. which incoming request spawns which outgoing requests). This context-specific information cannot be automatically made available to the underlying frameworks without any application support. The code modification for context propagation can involve a considerable investment of time, given dozens or hundreds of microservices written by different teams. Some microservices can be more challenging (e.g., legacy apps) or completely impossible to modify (e.g., proprietary, binary code), resulting in incomplete request traces.

Draft – Please Do Not Share Or Distribute

Even when feasible, adding tracing instrumentation is often buried under a long list of feature requests. This is one major factor[1] that has limited adoption. Gartner, evaluating the corresponding commercial space of application performance monitoring (APM), projected only 20% of enterprise applications would have deployed APM as of 2021 [38].

In this paper, we pose the question: to what extent can we accurately produce request traces without instrumenting applications? We refer to this problem as **blackbox request tracing**.

There is a large body of work on distributed tracing, that tackles a different, but related, problem – inferring that service X tends to depend on service Y, either using individual span-level information [11, 15, 35] or relying on request traces generated via context propagation [16]. In contrast, with blackbox request tracing, we seek finer-grained information about which specific requests resulted in which other specific requests, without relying on any application support (context propagation). This is vastly more valuable (e.g., to debug problems with specific requests, or important groups of requests) but also fundamentally more challenging. Applications concurrently serve many incoming requests, making it difficult to determine which incoming requests led to which child requests.

Blackbox request tracing therefore remains an unsolved problem that is highly desired by industry [10]. A few existing proposals do attempt to tackle this problem [36, 39], but make several simplifying assumptions about the application threading models, which severely restricts their applicability. Our work, in contrast, seeks a more general solution that is not tied to a specific threading model.

Our approach is based on two observations. First, while it might not be possible to construct fully accurate request traces without leveraging application support or without making restrictive assumptions about the application's threading model [36, 39], even *approximate* request tracing (with good enough accuracy) can be highly useful. Queries pertaining to a single request and its children require a fully accurate trace for that request. However, queries pertaining to a *sub-population* of requests (e.g. the set of requests with response latency in the tail 5%ile, or those belonging to higher priority users, etc) need request tracing to track the children of the requests belonging to the specified sub-population, but can still be meaningfully answered with approximately correct request traces.

Our second observation is that modern microservice environments provide new avenues for collecting useful information that we can leverage for blackbox request tracing. In particular, even though we treat the application itself as a black box, we can see detailed information about the communication in and out of that black box. Service meshes, whose sidecars act as proxies for each microservice instance, can

[1]Others include cost of commercial solutions and performance overhead.

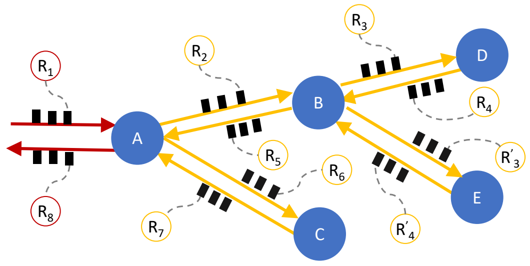


Figure 1: Traces and call graphs for an example application

observe layer 7 information like the endpoints and arguments in a request. eBPF, meanwhile, can analyze the application's interactions with the OS, such as the exact timing of opening or writing to a socket. Moreover, modern packaging of applications for Kubernetes environments makes it feasible to spin up applications in test environments, where they can be observed in a controlled way. The call graph – that is, the sequence of invocations of backend services initiated by request arrivals at a service – is one crucial property that can be reliably observed in such test settings, as it depends mainly on application code and system inputs.

We show that the above information sources, incorporated together, are sufficient for blackbox request tracing with useful accuracy. We design an optimization framework which can combine various types of constraints and hints to disambiguate request ancestry. Our prototype, in particular, incorporates timing and call graph constraints, as well as soft timing heuristics to boost accuracy. In our evaluation using applications from the DeathStarBench [23] benchmarking suite, we find that a simple baseline approach has 77% accuracy in linking spans, while our techniques boost this to 98%. Our preliminary analysis on a production dataset from Alibaba [8] running several customer-facing applications showcases a high accuracy in the range of 85%-95% even under variable, high system loads. Even though this accuracy is imperfect, the prototype may already be useful for certain tasks. We demonstrate two examples: determining which back-end service caused a particular subset of API calls to be slow, and detecting changes in a service's performance profile while A/B testing.

However, we believe this is not the end of the story: we expect accuracy could be pushed increasingly high, in increasingly difficult environmental circumstances, by incorporating more information into our (or a similar) framework. We describe several such necessary and promising directions in future work (§6). Overall, we believe blackbox request tracing is a promising way to provide developers with a "free" debugging tool, without additional effort on their part.

Draft – Please Do Not Share Or Distribute

2 Problem Description

2.1 Terminology

We refer to Figure 1 (representing a microservice based application) to explain the terms used throughout the paper.

Span. A span represents one execution of an API call at a running service instance, with metadata like the caller, callee, API endpoint, start time, and end time. Generally, the root of the tree is a span invoked by an external client. In the above example, the root span starts when request R_1 arrives at front-end service node A, and ends when the corresponding response R_8 is returned. The remaining spans involve subsequent request-response pairs, e.g R_2-R_5, R_3-R_4, etc. which are child spans of the R_1-R_8 pair.

Call graph. At each service X, we refer to the order in which back-end calls are made to respond to an incoming request as the *call-graph* at X. For the above example, the call graph at A tells us that to respond to an incoming request, A calls B and C *sequentially* and once it receives a response from C, it returns a response. Similarly, the call graph at B tells us that to answer a request from A, B calls D and E *in parallel* and returns a response once it hears back from D and E.

Parent-child relationship. A parent-child relationship $R_1 \rightarrow R_2$ indicates that in the course of processing R_1, the microservice invoked R_2. In the above example, in order to process and respond to the incoming request R_1, A calls service B (R_2). Once it gets a response R_5 from B , it calls service C (R_6). Once it receives the response R_7 from C, it does some processing of its own and returns the response R_8 to the user. This makes the requests R_2 and R_6 children of the request R_1. Likewise, at Service B, follow-up requests R_3 and R'_3 are children of the corresponding request R_2.

Request Trace. A request trace captures the parent-child relationship. The record for each request within the trace includes its span (i.e., the start and finish time) and pointers to its child request records.

2.2 Request Tracing

The key problem we target in this paper is, given log files with individual spans, [2] can we construct complete *request traces*, i.e. map each request arriving at each service with the corresponding child requests? In the context of our example, we wish to map R_1 to R_2 and R_6 at A; R_2 to R_3 and R'_3 at B, and so on. We refer to this problem of obtaining individual request traces for all requests as *request tracing*.

2.2.1 Why is request tracing useful?

Tracking the journey of a request through several service components of a distributed application, that work in tandem to generate a response, is critical for various debugging and troubleshooting tasks. For example, let's say an operator is interested in knowing which microservice is primarily contributing to the delay for a small set of high priority client requests. To produce the answer, one must be able to map which backend requests were made to produce the response, for each of those high priority requests at each service.

[2]Such span-level information, used as an input to our system, can be obtained using service meshes or eBPF hooks, as discussed in §4.

2.2.2 Whitebox request tracing: why is it hard?

One way to keep track of a request's journey through the application is to instrument all service components so that a unique *context identifier*, attached to an incoming request, is carried onto any requests to backend services that result from the incoming request. For example, R_1 carries a unique identifier that service A propagates to R_2, from which service B propagates to R_3, and so on. This is known as *context propagation*, and has been extensively explored through an active line of research [22,27,28,30,37,43]. Given support for context propagation, spans from each microservice can be grouped by their context identifiers and spans with the same identifier can be stitched together to form a request trace.

Such context propagation must be inherently supported in the application logic, since only the application can track the execution flow across function boundaries. It cannot be provided as a plug-in feature by the underlying frameworks (e.g. the service mesh or RPC frameworks) which do not have the required visibility into the application logic and its internal execution flow.

Although copying identifiers seems simple at first glance, it involves major practical hurdles: all software components, maintained by different teams or different vendors, have to appropriately propagate context. This involves agreeing on an API to use consistently, but more importantly, instrumenting internal microservice code to carry identifiers across function calls and data structures.

OpenTelemetry [32] defines a standard API and distributed tracing frameworks like Jaeger and Zipkin (and other commercial solutions) provide instrumentation tools for context propagation, either in common libraries or via hooks that developers can explicitly call. However, it is up to individual app developers to use the APIs or leverage the available tools, and context propagation continues to be a difficult task that increases app development burden. As a result, it has seen limited adoption, especially for legacy applications which require significant developer effort to modify existing code (only 20% of enterprise applications, as of 2021 [38]). Furthermore, an application could use proprietary third party service components which may not be possible to instrument.

If software components have not yet been instrumented to propagate context, one could use program analysis techniques to automatically modify software components so that they pass request context. This approach, taken by systems such as [20], is error-prone and still requires developers to go through the changes suggested by the tool and approve them.

Draft – Please Do Not Share Or Distribute

2.2.3 Blackbox request tracing

In the absence of context propagation, another strategy to construct request traces is to infer the request traces based on the information available in the spans. Specifically, one can match incoming requests at a service, say A, to outbound requests from A based on information contained in the requests such as timing data, thread-level data or header parameters. One benefit of this approach is that very little from the application developer is required. This approach, which we refer to as *blackbox request tracing*, is the subject of our work.

2.2.4 Existing approaches for blackbox request tracing

Existing works such as DeepFlow [36] or vPath [39] solve blackbox request tracing for a restrictive set of applications assuming a specific threading model for the application. Specifically, for the threading model, it is assumed that there are no hand-offs between threads and no asynchronous calls, so that every outgoing request from a thread can be mapped to the most recent outstanding request picked up by that thread. Using this assumption, DeepFlow (and vPath) is able to record the thread t that picked up a request r_{in}, and for any subsequent request r_{out} sent by t until t picks up the next request, associate r_{in} with r_{out}. While this scheme is effective for applications that follow this threading model, these assumptions do not hold for several common application types – applications which hand over requests to communication threads of RPC libraries such as Node.JS [4] or gRPC [1], or any application that employs asynchronous communication or batching.

2.2.5 Our approach

We seek a more general solution for blackbox request tracing that is not restricted to specific threading models. We observe that modern microservice environments provide new avenues (e.g. service mesh and/or eBPF, access to test environment, etc) for collecting information. As we discuss in § 2.4, while these avenues cannot, on their own, provide explicit information to construct request traces in the absence of application-level context propagation, they can be used to obtain other useful information. Our solution, TraceWeaver, exploits these avenues to infer the call graph and span timings, and then applies a novel timing analysis algorithm to construct *approximate* request traces from this information. TraceWeaver can be applied to each service individually, and can also co-exist with other solutions for blackbox request tracing (e.g. completing the request trace by stitching spans at specific services that do not support context propagation, or at services that have threading models that do not comply with the assumptions of Deepflow or vPath).

2.3 Related "Distributed Tracing" Problems: How they differ?

Several works under the umbrella term "distributed tracing" solve a variety of problems distinct from request tracing. One such problem, that multiple existing work try to tackle, is to obtain the service-level dependencies such as service A calls service B to answer inbound requests at A but not vice-versa. We refer to this problem of obtaining dependencies between services as "dependency mapping". The Mystery Machine [16] is a system for dependency mapping. It assumes context propagation and uses the request-traces thus obtained to derive a model of how services talk to each other. As discussed in § 2.2.2, context propagation is burdensome for developers and hence has limited adoption in the real-world. Orion [15], Sherlock [11] and WAP-5 [35] also tackle dependency mapping – they take as input span level data, and obtain the dependencies between services via analyzing delays in network traffic between when A receives a request and when A talks to B.

In contrast to dependency mapping, for request tracing, we seek finer-grained request traces, linking a specific request inbound at service A to another specific request outbound from A (and similarly at other services). Request tracing is more valuable than dependency mapping since request traces can be used to analyze performance of a single request or more generally, any subset of requests (see § 3.3). At the same time, disambiguating between far more numerous possibilities for request traces for every single request is fundamentally more challenging than finding which other services, a particular service depends on. We tried to repurpose one of the above dependency mapping systems, WAP-5 for blackbox request tracing and found its accuracy to be low (§ 5). Sherlock and Orion employ similar analysis, hence we expect them to have the same problems as WAP-5.

Many systems take as input request traces, and analyze them to improve various aspects of the system. For instance, Snicket [13] optimizes sampling of request traces based on developer queries for efficient storage. [42] uses logs from services, tagged with identifiers, to learn models that can help in various storage related decisions. DQBarge [17] injects critical system information (e.g. load) onto requests, which is passed around so that services can optimize for quality/performance trade-off. All of the above systems assume capabilities similar to context propagation, but they can be used in conjunction with blackbox request tracing which can make request-traces available for their analysis.

2.4 Available information for blackbox request tracing

We briefly outline commonly available monitoring components and information that can be leveraged for blackbox request tracing.

2.4.1 Service Mesh

Service meshes (e.g. Istio [26], Consul [24]) are application layer management software that sit in front of application services. So, when an app performs a remote API call, instead of directly communicating with the remote service instance, the request passes through a *sidecar* paired with the application instance. Service meshes provide a variety of communication

Draft – Please Do Not Share Or Distribute

functionalities such as discovering service instances, load balancing among those instances, retrying failed requests, etc.. They have two properties that are of interest to us: (1) they can be deployed transparently to the application regardless of programming language or framework, and can even be retrofitted onto existing apps. (2) As every inbound and outbound API call passes through the service mesh, it provides an opportune insertion point for telemetry [9]. Sidecars can see requests, associated responses, and API contents (e.g., HTTP headers) and can even modify requests. They do not, however, directly see which requests spawn which other requests. In other words, sidecars can easily see *spans*, but cannot directly track *request traces*. Thus, we have more work to do.

2.4.2 eBPF

eBPF [21] is an instrumental technology that allows users to capture the system calls made by a program, without any source code modification. eBPF can be used to obtain detailed insight into the inner workings of a service. For example, via eBPF hooks, one can obtain highly accurate timestamps of when a request/response arrives at a socket buffer by hooking on kernel APIs used for reading (e.g., read, recvmsg) or writing data (e.g., write, sendmsg). We note that both service meshes and eBPF monitoring can provide timing data about spans, hence either can be used– both are not needed.

2.4.3 Standardized test environments

Test environments designed to mirror production deployments assist developers in various tasks – functionality testing, load and failure handling, fault injection, and understanding different policy trade-offs. While such *mock* environments have existed for a long time, the advent of containerization technologies (e.g., Docker [19]) and orchestration frameworks (e.g., Kubernetes [29]), which allow for a standardized approach to package and run software, has made it possible to mimic production (albeit at a smaller scale) accurately.

One challenge in transparent tracing is automatically inferring what call graph an API call results in. Systems such as [6, 35] seek to obtain the call graphs from production environments from span data, without request-level traces. However, such procedures can have inaccuracies due to many concurrent requests and the inability to run tests to explore different hypotheses. Test environments allow for isolated and automated analysis – one can send a single query to API endpoints (using inputs from production traces) to observe the precise sequence of backend services that are invoked and obtain the call graph.

Test environments can also facilitate other investigations that can help in blackbox request tracing, such as obtaining typical processing delays at a service, relationships between HTTP request parameters, and learning threading patterns. It is important to note, however, that acquiring these hints provides only a useful building block rather than a complete solution, as we are still faced with the more challenging task of accurately mapping concurrent requests to their respective child spans in the production environment.

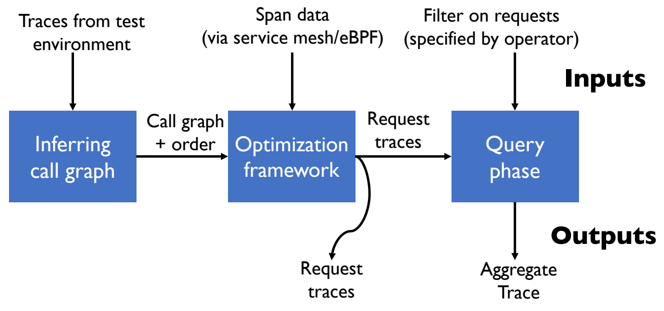


Figure 2: Inputs and outputs to each phase in TraceWeaver.

3 Design

There are three phases to the design of TraceWeaver (Figure 2).

1. Inferring Call Graph: the first part constructs the service call graph and the ordering within the call graph via test-environments (§ 3.1).
2. Optimization: using the call graph and the span data (obtained from sidecar proxies or eBPF hooks), TraceWeaver casts the blackbox request tracing problem as an optimization problem that needs to find the most appropriate trace for each request (§ 3.2).
3. Querying: the operator can specify a filter on the set of requests to see an "aggregated trace" annotated with the aggregated delays over the subpopulation of requests that match the filter (§ 3.3).

3.1 Inferring Call Graph

Data Collection. We use test environments to learn constraints associated with an <application/ API endpoint> pair. For example, in order to test the call-graph associated with querying appname.com/user-registration, we run queries to this endpoint. By analyzing the ensuing data-flow within this isolated test environment, the service-level dependencies in the call-graph can be ascertained, e.g. A calls B and C, and B calls D and E (Figure 1).

However, more subtle constraints such as does service A invoke B or C serially or in parallel are more challenging to learn. Since the application doesn't already have tracing, multiple requests arriving at A send out numerous backend requests to B and C muddying any potential analysis about sibling relationships. Our solution therefore involves running requests one at a time in isolation, in the test environment, to observe these relationships which would guarantee accurate mapping of a request from the starting to the ending span (as it is the only request in the system). Now, we can run various tests to learn about the pattern of backend request invokation.

For example, to validate whether invoking C depends on the completion of B (serial order), we can artificially delay the response from B and analyze the change in start time of the request sent to C. Running several such tests produces a trace dataset which we can analyze offline to learn useful constraints for our optimization framework.

Offline analysis: Looking at the request tree of a single isolated request allows us to infer service-level dependency structure. We conduct the following analysis to then infer the relationships between siblings at a service (i.e. their respective ordering) by looking at a collection of such isolated traces. We model the endpoints invoked by service A (endpoints B, C, and D) as vertices in a graph and add directed edges to indicate ordering relationships. For example, an edge from B $\rightarrow$ C indicates that A's request to C depends on the response it gets from B. Initially, we add directed edges between every pair of vertices to generate a fully connected graph (as every relationship can be possible). Now by analyzing the traces from our carefully constructed dataset, for every violating example of a relationship, we remove the corresponding edge from the graph. For example, if a particular trace indicates that C was invoked before or during B's execution, we remove the C $\rightarrow$ B edge. After iterating through all traces, we are left with a directed acyclic graph with a set of edges which signify genuine sibling relationships.

3.2 Optimization phase

We frame the problem of blackbox request tracing as an optimization problem and propose an algorithm that approximately solves it. The optimization problem uses as input: (i) Span-level information, that includes the request-response pairs (e.g., R_1-R_8 from Figure 1) and the corresponding timing information. This information can be obtained from service meshes (sidecar proxies) that intercept all requests and responses. It can also, alternatively, be obtained from eBPF hooks that can intercept syscall (reads and write to socket buffers) as exemplified in §4. (ii) The call-graph (with service-level dependencies and ordering), obtained as explained in §3.1. We use knowledge of the call graph to impose request causality constraints at each service. These include the more evident parent-child causal relationships (e.g., incoming span R_1 at A must arrive before outgoing span R_2 to B is sent out), and much less evident "sibling" relationships (e.g., endpoint B (span R_2) must be invoked before invoking C (R_6)). With these constraints in place, we run our optimization algorithm (described in §3.2.2) to accurately stitch together spans to obtain traces.

3.2.1 Assumptions

Our solution makes the following assumptions: (i) A parent span's response is sent out only after all its child spans finish processing. (ii) Every successful request has a response, i.e., we assume call-return behaviour commonly employed by REST and gRPC endpoints. (iii) Call graphs are either static with a well-defined structure, or have a limited form of dynamism where a request follows a subset of the call graph (e.g. due to caching). We leave tackling other forms of dynamism, e.g. due to retries, failures, and quorums to future work (§6). (iv) A service's processing time is well-captured by a standard distribution (e.g., Gaussian, exponential) in a non-trivial proportion of the time (we discuss how this assumption can be relaxed in §6).

We describe the optimization framework next.

3.2.2 Basic optimization (for uniform call graphs)

For ease of exposition, we first describe the optimization framework when all requests follow the same call graph order, that is, there's no dynamic behaviour induced due to caching or errors. We then describe how we can fold dynamism into the optimization.

Breaking the problem down to each service. In order to reconstruct the request trace, it is enough to solve the problem locally at each node: mapping all incoming spans at node X to the outgoing back-end spans from X; e.g., at node A in Figure 1, this means mapping incoming spans from the external client (e.g. R_1) to outgoing spans to B and C (e.g. R_2 and R_6 respectively). Recall that we know the mapping between the request and the response of each span. Therefore, we can consider all mapping problems within a request-response pair as an independent problem and obtain local mappings at each node. These local mappings can then be stitched together to reconstruct the entire request trace.

In order to reconstruct the request trace, it is enough to solve the problem locally at each node: mapping all incoming spans at node X to the outgoing back-end spans from X; e.g., at node A in Figure 1, this means mapping incoming spans from the external client (e.g. R_1) to outgoing spans to B and C (e.g. R_2 and R_6 respectively). The local mappings at each node can then be stitched together to reconstruct the entire request trace. We now describe our approach for solving the problem locally at each service X, again refering to Figure 1.

We define an ***assignment*** for an incoming request as a mapping between the incoming request at service X and its children requests from service X.

Scoring an assignment: In Figure 1, for service A, let incoming span R_1-R_8 be mapped to outgoing spans R_2-R_5 and R_6-R_7 according to an assignment S. We define the score of assignment S as

$$\begin{aligned} &Score(S) \\ &\quad = Score_{AB}(t_1,\ t_2) + Score_{BC}(t_5,\ t_6) + Score_{CA}(t_7,\ t_8) \end{aligned}$$

where t_1 is the time at which R_1 arrived at A, t_2 is the time at which R_2 was sent, t_5 is the time at which R_5 was received, t_6 is the time at which R_6 was sent, t_7 is the time at which R_7 was received and t_8 is the time at which R_8 was sent. We say that S is *feasible* if $t_1 < t_2 < t_5 < t_6 < t_7 < t_8$.

To compute $Score_{AB}$, we learn the distribution for the *processing delay* between when a request was received from the external client at service A, and a corresponding request was sent to B. We fit a Gaussian distribution for this delay t: $N(\mu_{AB},\ \sigma_{AB})$ and set $Score_{AB}(t_1,\ t_2) = \log P[t = t_2 - t_1|N]$, where $P[t|N]$ is the probability distribution function of $N(\mu_{AB},\ \sigma_{AB})$.

Estimating μ_{AB} and σ_{AB} is non-trivial, since we don't have the actual mappings between requests arriving at A and child requests from A to B, that could have given us the true values of the processing delays for AB. We note that μ_{AB} can still be estimated exactly without knowing the precise mapping. Since the mean of the differences equals the difference of the means, we can take the difference between the mean of arrival times of k external requests at A and the mean of departure times of k requests from A to B.

To estimate σ_{AB}, we divide the k incoming and outgoing requests into $M = 10$ batches after sorting them according to their start times. For each batch, we compute the empirical mean and calculate the standard deviation across all batch means. Multiplying that with an appropriate factor, we get the estimate for σ_{AB}. Note that this estimate is approximate since the batching is not perfect as the outgoing spans corresponding to incoming spans at the boundary of a batch could be in the previous or the next batch. Nevertheless, as long as the batch size is somewhat large to minimize this boundary effect, we can get an accurate estimate. We periodically estimate μ_{AB} and σ_{AB}. We similarly compute $Score_{BC}$ and $Score_{BC}$.

Optimization problem Next, we cast the problem of finding assignments for each incoming span as an optimization problem where the goal is to find assignments that maximize the total score of all chosen assignments, subject to two constraints: (i) all chosen assignments should be feasible, and (ii) an outgoing request should only be assigned to one incoming request (its parent). This optimization turns out to be an instance of the multidimensional assignment problem which is NP-hard [31].

Online Algorithm. We propose an online algorithm that iterates over all incoming requests in the order they arrive. One way to obtain the mapping is iterate over each incoming request, and greedily assign each request the highest score assignment S, removing the mapped outgoing requests in S, so that they don't get assigned to subsequent incoming requests. However, this greedy approach does not optimize score across multiple requests. Hence, we build upon this greedy algorithm to optimize the total score. We describe the full algorithm below,

1. First, we sort the incoming requests by their start time and divide the incoming requests in small batches. To decide the boundary of a batch, when we see a significant gap between subsequent incoming spans (> the mean span latency), we end the current batch there. If the gap doesn't appear before hitting some threshold (10 spans), we close the window to keep the batch size small.

2. Within each batch V, we use a technique to approximate the global maxima, attempting to find assignments for all incoming requests in V that maximize the total score in V. Once the assignments have been chosen for batch V, the outgoing requests of the assignments are deleted so that they can't be assigned again to any span in subsequent batches.

3. The technique we use to approximate the global maxima in the batch V, in step (2), is as follows. We find the top $K = 5$ assignments for each request. Then we create an instance of finding a maximum cost independent set (MIS), where the top K assignments for each request in V are nodes and two assignments have an edge between them if they both have the same outgoing request for some service. As each batch is small, we can solve this MIS exactly or nearly optimally.

3.2.3 Incorporating call-graph dynamism

Dynamic call graphs, where requests deviate from the typical call-graph, can arise due to caching, errors, retries etc. TraceWeaver handles one class of deviations, where a request can follow a subset of the usual call graph. Such dynamism can arise due to caching or requests which don't get passed on to backend services due to failure at an upstream service.

In such cases, incoming spans will be higher in number than the outgoing spans (at the service where the dynamism is triggered). We fill up such discrepancies using phantom "skip" spans which allow some spans at A to not make a call to B, by mapping those spans to a skip span.

Consider the discrepancy between incoming and outgoing spans in a given batch (corresponding to the optimization window of our online algorithm in §3.2.2). Using this discrepancy as an estimate of the number of skip spans can be inaccurate due to boundary conditions. This inaccuracy is more acute for small batch size, and can be reduced by counting the number of skip spans over a longer time window. Hence, we calculate the discrepancy over a reliable large window (10 seconds worth of spans) and use that as the maximum skip span quota over that window. We next distribute these skip spans to the individual optimization windows (smaller batches of requests) in the larger 10s window. We use a waterfilling algorithm [41] for doing so. This algorithm iteratively distributes the spans to the most needy batch (i.e. the batch for which allocated number of skip spans is farthest from the discrepancy between the outgoing and incoming spans), stopping when it runs out of the maximum skip spans quota. This ensures that the error in estimation is distributed across batches.

A final thing we have to take care of is how do we estimate delay distributions if there are skip spans. E.g. in the application from Figure 1, if for a request, A skips the call to B and directly calls C, we need to know the delay distribution between A and C in order to score an assignment that uses a skip span for B. Hence, we build additional distributions for any pair of endpoints that A talks to and A itself, using an algorithm similar to WAP5 [35].

Draft – Please Do Not Share Or Distribute

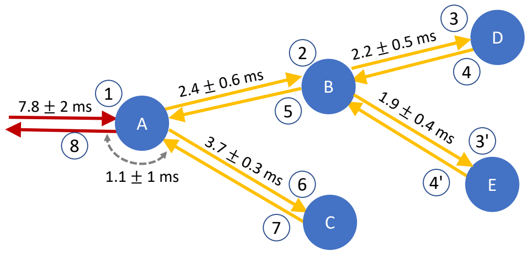


Figure 3: Example aggregate trace for a subset of requests. We also produce the processing delay at each service (the dashed curved arrow, not all such delays are shown)

3.2.4 Per-service Confidence Scores

For each service A, we compute a confidence score for A, equal to the fraction of incoming spans at A that either remained unmapped or weren't assigned their top choice mapping of backend-requests. We found that this confidence score correlated well with the ground-truth accuracy (Figure 6a).

3.3 Querying phase: Generating aggregate traces

Without any other assumptions, relying only on timing/thread-level or header parameters for matching spans for a request will inevitably result in some errors in the request traces obtained via the algorithm above. However, even an approximate scheme for blackbox request tracing can be very useful for operators. Operators are often interested in aggregate-level statistics about a subset of requests, such as for A/B testing or to analyze performance for requests from a client among others (§ 5.4). Hence, we provide the following feature to allow operators to take advantage of the request traces obtained above– the operator can specify a filter on requests that only a subset of requests S will satisfy. We can then output an aggregated trace for the subset of requests with delays at each service, aggregated over requests in S (see Figure 3).

4 System Challenges

Key inputs required by TraceWeaver include the span-level information and the call graph. We discussed our algorithm for inferring call-graph from the test-environment in §3. Span-level information, on the other hand, must be obtained from live production traffic.

In deployments that support service meshes, this information is readily available. The sidecar proxies intercept the HTTP connections, and have full visibility into the request and response headers to gather span-information (i.e. the requsest-response mapping and the corresponding timings).

We can employ eBPF hooks to obtain span-level information in deployments that do not support service meshes. We implemented this eBPF-based approach for a test application that we evaluate. To capture span-level information at each service, we employ eBPF to hook onto networking syscalls (e.g., accept, recvmsg, sendmsg, close). For each syscall, there are two kprobes: an entry and a return probe. The entry probe

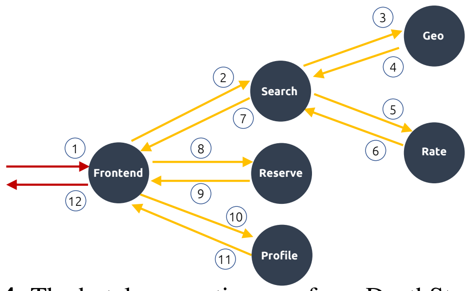


Figure 4: The hotel reservation app from DeathStarBench.

is used to store the syscall arguments in a BPF hashmap and the return probe is used retrieve these arguments on function return. For example, when an accept() syscall is made to extract a connection request, the eBPF hook can capture its arguments such as the sockaddr to identify the unique session. Similarly, when the application reads/writes from/to the socket, the probes corresponding to those syscalls can capture the data being read/written and attach it to the right session in the hashmap. This mechanism enables capturing information on the wire without interfacing with or instrumenting the application.

Capturing request/ responses from gRPC, which is built on top of HTTP/2 [3] (the next generation of HTTP protocol), is a challenge. This is due to their stateful header compression scheme (HPACK [2]) which makes request/ response headers uniquely difficult to parse. The HPACK-based compression feature involves the client and server maintaining a dictionary, mapping previously seen headers to unique numeric codes, and only passing these codes in further communication. Without access to this dictionary, we need to resort to sniffing from the beginning of a HTTP connection in order to not miss important connection information available only at initial phase. We then perform "man-in-the-middle" style dictionary updates to keep track of the dictionaries actively being updated at the client/ server. In our implementation, instead of storing this map in kernel space, we collect the raw data with eBPF, pass it to userspace and use standard HTTP libraries to examine this data which allows us to read the plaintext HTTP header and payload. We currently do not handle encrypted traffic but existing software solutions like Pixie [5], which are built on eBPF, provide the ability to trace the API between the application and the SSL/TLS library to capture it before it is encrypted. We assume this capability for encrypted traffic.

5 Evaluation

5.1 Setup

We evaluate TraceWeaver on two benchmark apps from the DeathStarBench [23] suite – HotelReservation and Media Microservices, a node.js based microservice demo [7], and on production traces from the Alibaba cluster dataset [8].

We illustrate the HotelReservation app from the DeathStarBench in Figure 4 as an example. The microservice is orchestrated using Kubernetes [29], and Istio [26] is used for inter-service communication. The app runs on 3 VMs running Ubuntu 16.04, each provisioned with 4 cores and 4 GB of RAM. All VMs run within a bare-metal 32-core Intel Xeon Silver server. We use the wrk2 [40] load generator to generate load that hits the application's frontend. We employ Jaeger tracing [27] to collect ground truth end-to-end traces. Within this setup, we consider the /hotels API, an endpoint hit by clients via HTTP-based GET requests to get profiles of available hotels within a specified geographic location and date range. The intermediate API calls $1 \rightarrow 12$ in Figure 4 describe a serial call graph for this API request. We have a similar setup for the Media Microservice app which comprises 14 different services alongside their caching and database components (Memcached, Redis, and MongoDB) and the node.js benchmark app comprising 7 different services.

Baselines: We compare TraceWeaver against the following baselines:

(i) WAP5: which tackles a weaker problem of dependency mapping by using the span-level information to compute the probability with which services invoke each other. To this end, WAP5 tries to maps child spans to the most probable parent span based off approximated inter-span distributions. We re-purpose this part of the algorithm to map the outgoing request to the probable incoming request to construct traces.

(ii) DeepFlow/vPath: As previously mentioned, DeepFlow assumes a specific threading model where threads pick up requests and process them to completion before switching to the next request. It assumes no request handoffs between threads, no asynchronous calls, and access to the thread identifier for the thread processing a request. While this can effectively stitch spans at services where these assumptions hold, it fails at services where it doesn't. vPath employs a similar assumption, hence in the cases we test, vPath is also represented by the DeepFlow results.

(iii) FCFS: We also construct a strawman solution where external requests arriving at a service A are assigned to the outgoing requests at service B based on their respective arrival and departure orders. The responses from B to A might arrive in a different order than requests from A to B – the order assigned to requests from A to C is the order in which responses arrive from B to A, and so on.

5.2 Benchmark applications

5.2.1 Accuracy vs. load

Figure 5a shows the accuracy of end-to-end tracing for varying load (requests per second (RPS)) for TraceWeaver, and the baselines, WAP5, DeepFlow/vPath and FCFS for the hotel reservation, media microservices, and node-js benchmark apps. As the load increases, requests are reordered (w.r.t. initial client request) more frequently at the individual services, so FCFS cannot keep up, but TraceWeaver holds steady. Our test applications employed gRPC framework where DeepFlow/vPath's assumptions break. This approach managed to get 80% accuracy at low loads (when the number of concurrent requests were lower), but the accuracy quickly degraded with increasing load. WAP5, whose statistical algorithm was designed for a weaker usecase, experienced similar degradation in accuracy with increasing loads.

Top K accuracy: We also analyze how accurate we can get if one of the top K mappings outputted to the user is correct and we find that we do extremely well on this metric. A Top K (for low K, we use K = 5) output is very useful to the developer as they're almost guaranteed to obtain the right mapping for a request within the ranked list of 5 possibilities which would enable them to inspect and debug requests at a very fine granularity.

5.2.2 Accuracy vs response times

Developers are often interested in end-to-end traces for anomalous requests, for instance, those with high response times. Figure 5b illustrates the accuracy for requests binned by their response time percentiles at load level = 125 RPS (response times ranged from 40 ms to 225 ms for our application). We can see that for the tail 10%ile requests, the accuracy of other baselines suffers, but TraceWeaver recovers the end-to-end traces accurately for >95% of the requests.

5.2.3 Accuracy under heavy caching

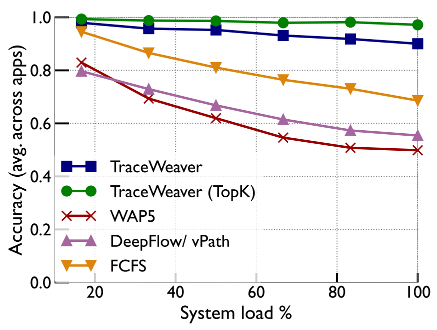
Behaviors such as caching and errors result in a dynamic call graph which differs non-deterministically from the general, static call graph we learn. To evaluate the algorithm's resilience to this, we artificially insert caching into the search service of Hotel Reservation, varying the cache hit probability from 5% to 80%. As we allow spans to skip certain endpoints optionally as mentioned in Section 3, we are able to get good accuracy in this dynamic scenario as illustrated in Figure 6b. Other approaches like FCFS and WAP5 fail to gracefully decline in accuracy.

5.2.4 Accuracy in asynchronous settings

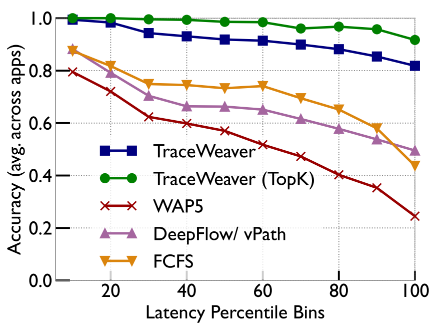
Another aspect of interest is what happens when asynchronous behavior like async I/O is present which can interleave requests within a thread by varying degrees. This experiment is illustrated in Figure 6c where we increase the standard deviation of the file size distribution to introduce more interleaving between the requests. Approaches like DeepFlow which depend on a synchronous model fail to perform well in such settings while TraceWeaver avoids the assumption and performs well.

5.3 Production traces

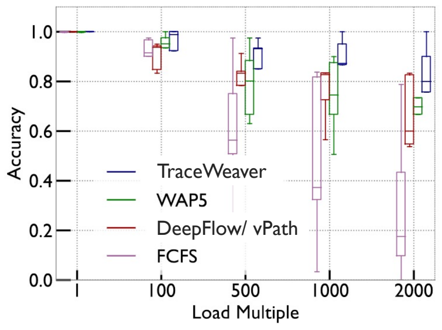
Next, we evaluate on production data from the Alibaba cluster dataset [8] in a similar way as on the benchmark applications (accuracy vs. load). Our analysis considers a dataset spanning ten services where requests are traced in a period of 12 hours. The traces contain parent-child relationships between spans, and start and response times of the individual spans. However, the production data is sampled at a rate of 0.5%, making the



(a) Accuracy vs. load (across multiple benchmark apps).

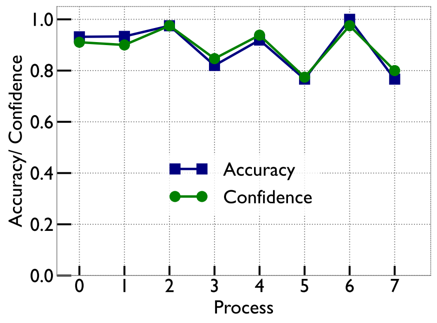


(b) Accuracy vs Latency bracket (across multiple apps).

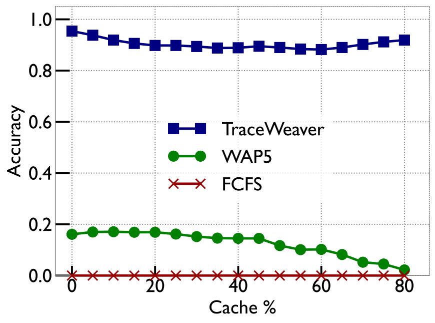


(c) Performance on Alibaba's dataset (across 10 services).

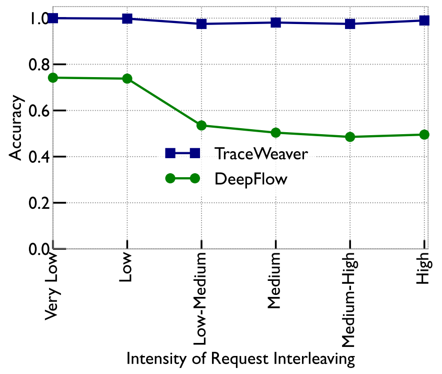
Figure 5: (a) Trace-reconstruction accuracy for all requests with increasing system load. (b) Trace-reconstruction accuracy across increasing load on Alibaba's dataset (boxplots presents accuracy %iles for 10 services).



(a) Confidence scores across services in the Alibaba dataset.



(b) Accuracy in the presence of dynamic behavior (caching)



(c) Accuracy in the presence of arbitrary storage I/O

Figure 6: (a) Confidence score compared to the accuracy at a given service (b) Accuracy across increasing caching rates at one endpoint within the Hotel Reservation application, (c) Accuracy with increasing intensity of I/O tasks which create interleaving requests within a single thread causes inaccuracy in DeepFlow.

tracing challenge extremely easy (due to large spacings between adjacent requests in the sampled set). To counteract this sampling effect, we artificially increase the number of traces by duplicating existing requests. Additionally, we compress the time intervals between these requests while preserving the essential delay characteristics of the service. In particular, if two requests, R1 and R2, have arrival times t1 and t2 and durations d1 and d2, we adjust the spacing (t2 - t1) to be much smaller while leaving d1 and d2 unchanged. In essence, smaller spacings lead to many more plausible candidates and pose a harder challenge for our tracing algorithm. We refer to the factor by which we reduce the spacings to enhance the effective RPS as the "load multiple." Note, to undo the sampling effect, a load multiple of 200-300 is sufficient, but we stress test it further to evaluate the algorithm's breaking point. Figure 5c shows our results on this dataset. As the load multiple increases, the accuracy drops for all algorithms, but TraceWeaver is able to get a high accuracy in the range of 85%-95% that is still practically usable.

5.4 Using approximate distributed tracing for debugging

We next highlight two use cases showing how the operator can use the approximate end-to-end traces derived from our approach for debugging tasks.

5.4.1 Troubleshooting delays for slow requests

We emulate a performance anomaly scenario by artificially inflating the request-response span latency at the "Reserve" and "Profile" services by 40ms for 10% of randomly selected requests. The operator is trying to localize which service(s) have the anomaly and wants to know the answer to the following question: *which service contributes most to the delay experienced by the slowest 2% of the requests?*

Answering this query requires the complete trace for the requests with an end-to-end response latency at 98%ile or higher and then computing the response time of each span associated with these requests. Figure 7a presents our results. Stitching the spans using the ground-truth trace reveals Reserve and Profile as the culprits. Traces stitched with our TraceWeaver's algorithm also correctly yield Reserve and Profile as the slowest services. Note that this answer which is of interest to the operator, is not affected by a few inaccurate traces. In the absence of distributed tracing, only individual span-level logs are available. If a developer simply looks at the tail latencies for spans at each individual service to answer this query, Figure 7a shows that they would find that all of Search, Geo, Reserve, and Profile have high response times, thus leading the operator's debugging process astray.

5.4.2 A/B testing of a recommendation engine

In this scenario, an operator wants to run A/B tests for a new version (=B) of a recommendation algorithm and wants to measure the effect of the new algorithm on user engagement. As is common, the upstream service is modified so that a small percentage (=x) of requests are redirected from the older version (=A) to the newer version (=B). Thereafter, the two versions are compared using statistical p-val tests. If $p < 0.05$, the operator can conclude that B results in better user satisfaction than A.

Without request traces, the operator can't determine which user request was redirected to A or B, and hence the only way to measure user-satisfaction is to consider the aggregate user-satisfaction score across all requests. If there's an increase in the overall user-satisfaction, one can conclude that the increase is due to the $x\%$ of the requests which were redirected to B and used the new algorithm. However, since B is untested, its common to have a small x, say 1%. If only a small number of requests are redirected to B, then the aggregate user satisfaction does not change much and one is not able to conclude whether B made a difference The blue line in Figure 7b shows that the p-value remains high for small x, indicating that there was inconclusive evidence for B being better. As x increases, the aggregate user-satisfaction score will pronouncedly reflect the changes due to B.

With request traces, the operator can separate the individual requests going to A and B (albeit with some error). Figure 7b shows that even when the individual requests to A, B are separated with an accuracy of only 90%, the accuracy for the troubleshooting task, in determining whether B improves user satisfaction, remains very high (since the p-val remains below 0.05) in comparison to the analysis without request traces, described above.

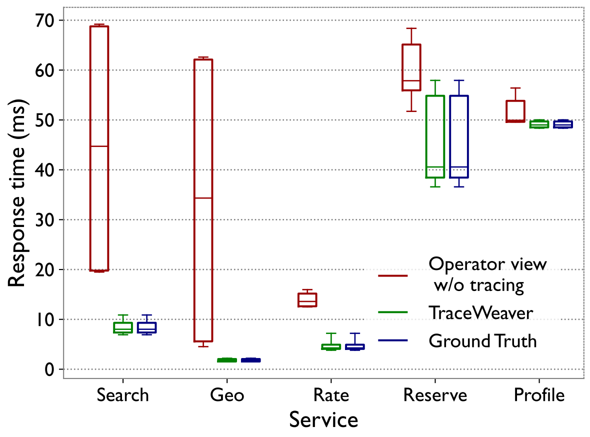
6 Future directions and limitations

Identifying thread affinity: Deepflow [36] and vPath [39] make restrictive assumptions about threading models, assuming that there are no hand-offs between threads and no asynchronous calls. We find that thread hand-offs are common in modern microservices (e.g. those using RPC libraries such as Node.JS [4] or gRPC [1]). Tracking a request across such hand-offs (potentially via eBPF programs hooked to socket syscalls) can provide useful information for request tracing. Even when a thread handles multiple requests concurrently and asynchronously, the ability to map requests to individual threads across hand-overs can help prune plausible candidates that our optimization algorithm must consider, thereby boosting accuracy. However, such fine-grained taint-tracking can be extremely challenging, and we leave a detailed exploration of this to future work.

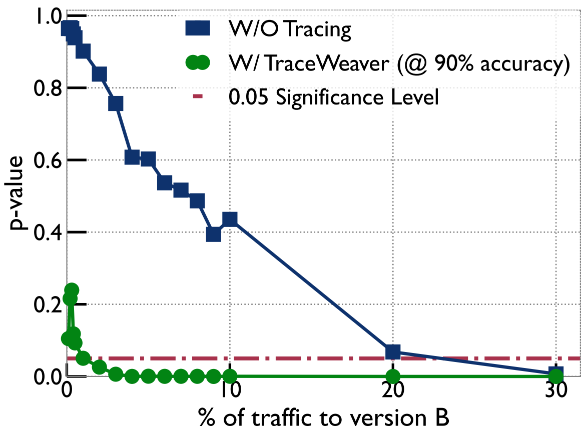
Targeted instrumentation: The confidence scores mentioned in §5 can also guide selective instrumentation. Analyzing confidence scores per service makes it possible to deduce which individual services pose the toughest challenge for transparent tracing. Developers can then explicitly instrument just these services, bolstering the overall accuracy significantly. Note that this can be significantly less burdensome than instrumenting every service.

Handling variations in the call graph: It will be crucial to handle the fact that real applications can deviate from

Draft – Please Do Not Share Or Distribute



(a) Latency profile analysis



(b) A/B tests using request-traces via TraceWeaver.

Figure 7: Troubleshooting using aggregate traces (a) Latency profile at each service for requests above the 98%ile latency bracket in a specific time period. Each boxplot represents the [5, 25, 50, 75, 95]%iles of the corresponding distribution. The developer view shows the distribution of the top 2% latency requests for each service, whereas the distribution for TraceWeaver is computed via the corresponding spans of requests of interest. (b) p-value for the statistical test comparing user satisfaction when $x\%$ of requests were sent to B, a better recommendation algorithm (the green line) and when all requests were sent to A (the blue line). Lower p-value implies that the evidence is more conclusive that B is better than A. If p-value < 0.05, then one can statistically conclude that B is better than A.

the call-graph because of multiple code paths in application logic, exceptions, or nondeterministic factors like retries and caching. TraceWeaver can currently only tackle the variations that stem from caching. Handling other forms of variations remains an interesting challenge for future research. Potential directions include looking at error codes in response headers to infer the number of retries.

Sampling: Distributed tracing with explicit context propagation can enable sampled tracing, where to reduce overhead, only a certain percentage of API calls are traced, but each is traced across its whole call tree. This may be difficult with our prototype, which can only attempt reconstruction after all the plausible children spans have executed. To deal with this we could could slightly oversample (logging all spans that may have been children of a span that was to be sampled) or sampling at a different granularity (e.g., all spans within a specific 1-second time window).

7 Conclusion

Distributed tracing has rapidly increased in importance with the evolution of modern cloud-native applications, yet its potential impact has been held back in part by the need to modify code. We have shown that useful tracing is possible without application instrumentation. Our preliminary results indicate this can be a very practical direction, with numerous questions that need to be answered, as well as numerous research opportunities.

References

[1] gRPC. `https://grpc.io/`.

[2] Hpack specification. `http://http2.github.io/compression-spec/compression-spec.html`.

[3] Http2. `https://http2.github.io/`.

[4] Node.JS. `https://nodejs.org/`.

[5] Pixie. `https://docs.px.dev/`.

[6] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 74–89, New York, NY, USA, 2003. Association for Computing Machinery.

[7] Algun. Node.js microservice. `https://github.com/algun/jaeger-nodejs-example`, 2019.

[8] Alibaba. Alibaba cluster trace program. `https://github.com/alibaba/clusterdata`, 2021.

[9] S. Ashok, P. B. Godfrey, and R. Mittal. Leveraging service meshes as a new network layer. In *Twentieth ACM Workshop on Hot Topics in Networks (HotNets)*, November 2021.

[10] O. Azizi. OpenTelemetry or eBPF? That is the Question. `https://cloudnativeebpfdayna22.sched.com/event/1Auyh/opentelemetry-or-ebpf-that-is-the-question-omid-azizi-new-relic-pixie`, 2022.

[11] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, page 13–24, New York, NY, USA, 2007. Association for Computing Machinery.

[12] P. Bailis, P. Alvaro, and S. Gulwani. Research for practice: Tracing and debugging distributed systems; programming by examples. *Commun. ACM*, 60(7):46–49, jun 2017.

[13] J. Berg, F. Ruffy, K. Nguyen, N. Yang, T. Kim, A. Sivaraman, R. Netravali, and S. Narayana. Snicket: Query-driven distributed tracing. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, HotNets '21, page 206–212, New York, NY, USA, 2021. Association for Computing Machinery.

[14] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst. Debugging distributed systems: Challenges and options for validation and debugging. *Communications of the ACM*, 59(8):32–37, Aug. 2016.

[15] X. Chen, M. Zhang, Z. M. Mao, and V. Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *OSDI*. USENIX, January 2008.

[16] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 217–231, Broomfield, CO, Oct. 2014. USENIX Association.

[17] M. Chow, K. Veeraraghavan, M. Cafarella, and J. Flinn. DQBarge: Improving Data-Quality tradeoffs in Large-Scale internet services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 771–786, Savannah, GA, Nov. 2016. USENIX Association.

[18] Datadog. Datadog. `https://www.datadoghq.com/`.

[19] Docker. Docker. `https://www.docker.com/`, 2013.

[20] DynaTrace. DynaTrace. `https://www.dynatrace.com/`.

[21] eBPF. ebpf: Extended berkeley packet filter, 2023.

[22] R. Fonseca, G. Porter, R. H. Katz, and S. Shenker. X-Trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, Cambridge, MA, Apr. 2007. USENIX Association.

[23] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud amp; edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[24] HashiCorp. Hashicorp consul. `https://www.consul.io`, 2021.

[25] Instana. Instana. `https://www.ibm.com/cloud/instana`.

[26] Istio. Istio. `https://istio.io`, 2021.

[27] Jaeger. Jaeger. `https://www.jaegertracing.io/`.

[28] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Venkataraman, K. Veeraraghavan, and Y. J. Song. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 34–50, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Kubernetes. Kubernetes. `https://kubernetes.io/`, 2014.

[30] LightStep. LightStep. `http://lightstep.com/`, 2023.

[31] D. M. Nguyen, H. A. Le Thi, and T. Pham Dinh. Solving the multidimensional assignment problem by a cross-entropy method. *Journal of Combinatorial Optimization*, 27(4):808–823, 2014.

[32] OpenTelemetry. OpenTelemetry. `https://opentelemetry.io/`.

[33] Oracle. Oracle. `https://docs.oracle.com/en-us/iaas/Content/Functions/Tasks/functionstracing.htm`, 2023.

[34] M. Raney. What I Wish I Had Known Before Scaling Uber to 1000 Services. In *GOTO Chicago*, 2016.

[35] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. Wap5: Black-box performance debugging for wide-area systems. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, page 347–356, New York, NY, USA, 2006. Association for Computing Machinery.

[36] J. Shen, H. Zhang, Y. Xiang, X. Shi, X. Li, Y. Shen, Z. Zhang, Y. Wu, X. Yin, J. Wang, M. Xu, Y. Li, J. Yin, J. Song, Z. Li, and R. Nie. Network-centric distributed tracing with deepflow: Troubleshooting your microservices in zero code. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 420–437, New York, NY, USA, 2023. Association for Computing Machinery.

[37] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[38] F. D. Silva and C. Rich. Broaden application performance monitoring to support digital business transformation, 2018.

[39] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang. Vpath: Precise discovery of request processing paths from black-box observations of thread and network activities. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, page 19, USA, 2009. USENIX Association.

[40] G. Tene. Wrk2: a HTTP benchmarking tool based mostly on wrk. `https://github.com/giltene/wrk2`, 2019.

[41] Wikipedia. Water filling algorithm. `https://en.wikipedia.org/wiki/Water_filling_algorithm`.

[42] G. Zhou and M. Maas. Learning on distributed traces for data center storage systems. In *4th Conference on Machine Learning and Systems (MLSys 2021)*, 2021.

[43] Zipkin. Zipkin. `https://zipkin.io/`.