The Benefit of Hindsight: Tracing Edge-Cases in Distributed Systems

Lei Zhang
Emory University and Princeton University

Zhiqiang Xie
Max Planck Institute for Software Systems

Vaastav Anand
Max Planck Institute for Software Systems

Ymir Vigfusson
Emory University

Jonathan Mace
Max Planck Institute for Software Systems

Abstract

Today's distributed tracing frameworks are ill-equipped to troubleshoot rare edge-case requests. The crux of the problem is a trade-off between specificity and overhead. On the one hand, frameworks can indiscriminately select requests to trace when they enter the system (head sampling), but this is unlikely to capture a relevant edge-case trace because the framework cannot know which requests will be problematic until after-the-fact. On the other hand, frameworks can trace everything and later keep only the interesting edge-case traces (tail sampling), but this has high overheads on the traced application and enormous data ingestion costs.

In this paper we circumvent this trade-off for any edge-case with symptoms that can be programmatically detected, such as high tail latency, errors, and bottlenecked queues. We propose a lightweight and always-on distributed tracing system, Hindsight, which implements a *retroactive sampling* abstraction: instead of eagerly ingesting and processing traces, Hindsight lazily retrieves trace data only *after* symptoms of a problem are detected. Hindsight is analogous to a car dash-cam that, upon detecting a sudden jolt in momentum, persists the last hour of footage. Developers using Hindsight receive the exact edge-case traces they desire without undue overhead or dependence on luck. Our evaluation shows that Hindsight scales to millions of requests per second, adds nanosecond-level overhead to generate trace data, handles GB/s of data per node, transparently integrates with existing distributed tracing systems, and successfully persists full, detailed traces in real-world use cases when edge-case problems are detected.

1 Introduction

Troubleshooting failures and performance problems in large-scale distributed systems is crucial. On one side, tiny performance misbehavior in a production system could be costly [1, 2, 19]. On the other side, exacerbated by growing system complexity, diagnosing problems takes onerous effort from system developers and requires significant engineering resources. Distributed tracing was invented as the solution of troubleshooting distributed systems by recording detailed, end-to-end traces of request executions, and have been proved helpful for a wide range of use cases [59, 62].

Prior distributed tracing works have demonstrated a wide range of use cases. Common-case analysis focuses on aggregated system behaviors, such as monitoring resource usage [46, 59, 60, 62, 70]. In contrast, edge-case troubleshooting (§2.1), the topic of this paper, focuses on rare and outlier system behavior, such as tail latency [18, 38, 48, 69, 74].

Since an edge case is rare by definition, tracing edge cases requires trace coverage of all requests. In typical production environments, tracing *every* request—including transmitting, processing, and storing comprehensive telemetry—requires enormous backend infrastructure and storage that is unacceptable to infrastructure operators. State-of-the-art tracing frameworks manage this overhead by collecting a small sample (0.001%) of traces [9, 34, 37, 64]. Though previous works practically reduce tracing overhead through head sampling [34, 64] and tail sampling [36, 37] techniques, they cannot trace edge cases at scale (§2.3).

In this paper, we resolve the problem of tracing edge-case requests in production environments. To achieve this, we focus our attention on *symptomatic* edge cases, where the performance effects of the problem manifest shortly after its causes and where the impacts can be observed programmatically. We propose **retroactive sampling** to collect telemetry data *back in time* from the present moment of detection from all machines that serviced the request. The key idea is to generate all trace data but only collect useful data through a retrieval mechanism.

To implement retroactive sampling, we built Hindsight—an always-on, lightweight distributed tracing system that is compatible with existing tracing APIs—as a practical tool for edge-case analysis. Under retroactive sampling, all trace data is recorded locally but only reported when a symptom is detected, allowing applications to generate copious trace data in case they are needed without encumbering the tracing system's backend collection infrastructure. Retroactive sampling ultimately reports the same volume of trace data as other sampling methods, but ensures that edge-case traces are not missed. To provide efficient and coherent retroactive sampling, Hindsight's design separates its dataplane, *e.g.* generating trace data into fast local memory, from control logic, *e.g.* for indexing metadata, coordinating among machines, and triggering collection for symptomatic requests on demand.

As demonstration, we apply Hindsight on three use cases corresponding to our running examples. We run experiments on the DeathStar Microservices Benchmark [24], the Hadoop Distributed File System [63], an Alibaba benchmark derived from production traces [42], and on several microbenchmarks. We have integrated Hindsight with OpenTelemetry [52] and as a replacement collection component for X-Trace [23]. Our experimental results show that Hindsight imposes nanosecond-scale overhead when generating trace data, can scale to 55 GB/s of data per node, rapidly reconstructs traces when triggered, and coherently captures problematic traces (>99%), as well as related lateral traces, within 100 ms of identifying a symptom.

In summary, our paper makes the following contributions.

- We describe the retroactive sampling abstraction for capturing traces of symptomatic edge-cases.
- We present the design of Hindsight, a distributed tracing system that implements retroactive sampling. Hindsight is compatible with existing tracing APIs and can be transparently integrated with existing applications.
- We apply Hindsight on real-world use cases and show that efficiently collecting edge-case requests is practical.
- We evaluate Hindsight on multiple benchmarks and real systems, showing that it can achieve nanosecond-level overhead on trace data generation and handle GB/s data per node while collecting coherent traces.
- We illustrate that Hindsight is compatible and performs better than state-of-the-art tracing systems (X-Trace and Jaeger) with more efficient trace-data generation and lower overhead, while providing edge-case tracing.

2 Motivation

2.1 Edge-Case Troubleshooting

Consider the following three examples of real-world use cases UC1–UC3 of edge-case troubleshooting from prior work.

Error diagnosis (UC1). Hardware failures, component errors, exceptions, and programming mistakes abound in large production systems [73]. Application developers often play the role of detective, to identify root causes of errors. An error might only arise due to a specific, rare combination of factors and code paths exercised; the symptoms of a problem often manifest far from the root causes [22, 41, 45], and the potential root causes are manifold, perhaps combined software or hardware problems on many nodes or network links [35].

Tail-latency troubleshooting (UC2). Distributed systems track a wide range of high-level health metrics, such as API distributions, latency percentiles, resource utilization, and many others [33, 34]. An operator may observe an unusual metric jump, say the 99^{th} percentile latency has spiked for some important API. However, knowing about the spike is not enough; the application developer must identify the specific service, code paths, or conditions that contribute to the peak to address any underlying problems [18, 38, 48].

Temporal provenance (UC3). Many modern distributed systems respond to requests through an architecture of loosely coupled microservices [62]. Application developers need tools for tracking queuing issues when the number of components in a distributed system is large [3–6, 8], since a request R exhibiting symptoms (*e.g.* prolonged queueing time) may not be the true culprit for the backlogged queue. Rather, the developer wants to follow the *temporal provenance* of R to determine *lateral traces* of other related requests with which R interacted through shared components and queues [72].

2.2 Distributed Tracing

Distributed tracing frameworks are in widespread use in both open-source [31, 52, 78] and major internet companies [34, 58, 64] to chronicle *end-to-end requests*. A trace is a recording of one request, and each trace contains spans, events, and annotations, along with timing and ordering, generated from every machine visited by the request. Compared to traditional logs and metrics, the key distinction of distributed tracing is that a trace captures the full end-to-end structural flow of request execution across all components visited.

Advantages. Distributed tracing is thus particularly useful for troubleshooting cross-component problems in large systems, since the request traces explicitly tie together the individual slices of work performed across different machines, enabling an operator to observe how the work done by one machine influences, and is influenced by, work done on others [23, 58, 64, 68]. Prior research on distributed tracing demonstrates a range of use cases, including common-case analyses centered on aggregate system behavior, distributions over data, and relationships between system components [34, 59, 60, 62].

Limitations. Since edge-case troubleshooting concerns rare and outlier system behavior, the symptoms and evidence of a problem might only manifest in a very small fraction of requests. Unfortunately for the operator, this sparsity may yield exceptionally few exemplar traces of edge-case behaviors and symptoms, owing to the design of modern distributed tracing frameworks. Let us look closer at how traces are captured before returning to this problem.

Current designs. Fig. 1 depicts a typical distributed tracing framework [34, 52, 58]. When a new request arrives at the application, the tracing framework assigns it a unique `traceId` (①). Every request is assigned a `traceId`, but not every request is actually traced; the framework sets a per-request `sampled` flag to indicate as such. From this starting point, the application then propagates the `traceId` and `sampled` flag alongside the request at the application level and includes them with all inter-process communication (②).

Any component that handles the request can generate trace data (*e.g.* spans, events) using the tracing framework's client library (*e.g.* OpenTelemetry [52])—trace data is only generated if `sampled` is set. Trace data gets explicitly annotated with the `traceId`, thereby associating the data with the request (③).

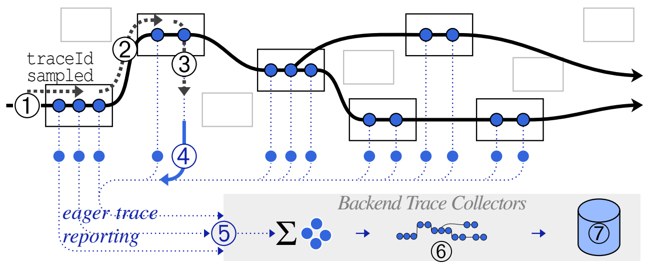


Fig. 1: **Distributed tracing** (§2.2). A request (solid black line) traverses system processes, depositing trace data that is eagerly ingested into the trace collector backends. End-to-end trace objects are constructed from trace data, processed, and stored in a database.

Ultimately there may be many components and machines that handled the request and contribute trace data. At the same time, many requests may execute concurrently (*e.g.* in different server handler threads), generating temporally-interleaved data with different `traceId`s.

The framework's client library eagerly enqueues, serializes, and transmits trace data (④) to its centralized backend collection infrastructure, or *backend* for short (⑤). The backend is distinct from the traced applications and is responsible for continually receiving, processing (⑥), and storing (⑦) trace data generated across all of the application's components. The backend uses the `traceId` to *join* data that was dispersed across many machines but belongs to same request into a single coherent trace object. The backend finally persists that trace object in a database if it decides to retain the trace.

Overhead vs. incompleteness. Traces can be detailed and produced at high volume, risking overheads. Traces at Google, for instance, are typically more detailed than debug-level logging [64]; each traced request at Facebook, similarly, generates several MBs of tracing data and approximately 1 billion traces are captured per day [34]. At high rates, tracing frameworks may encounter several potential bottlenecks: when generating data within the traced application (③); when transmitting trace data over the network (④); and in backend processing and storage (⑤–⑦).

To reduce overheads, the de facto practice is to capture fewer traces. Here, operating at the granularity of an entire trace maintains *trace coherence*: if a request is sampled, then the whole trace is kept including all data across all machines; otherwise nothing is kept. Coherent traces are essential for distributed tracing – a partial or fragmented trace has limited value in diagnosis [23, 29, 30, 64] because it loses the end-to-end visibility that makes the trace useful in the first place [58, 59, 68]. There are two main approaches for foregoing traces coherently: the system may decide to omit a request at ① before tracing and ingestion (*head sampling*) or the traces may be filtered after collection at ⑥ (*tail sampling*).

Head sampling reduces overheads by simply tracing fewer requests in the first place, *i.e.* by setting the `sampled` flag for only a small fraction of requests (①). By leaving `sampled` unset for the majority of requests, trace data will not be recorded for most requests, thus avoiding application overheads to generate data, ingestion overheads to transmit and process data, and storage overheads (③–⑦). Head sampling is widely used in practice; it is enabled by default in Jaeger [31] with a 0.1% sampling probability, and some production systems sample as few as 0.001% [34, 64].

Tail sampling is used to drop traces at the trace backends (⑥). Unlike head sampling, the application will still trace all requests and will incur all expenses of generating and ingesting the trace data (③–⑤). Tail-based sampling primarily allows backends to lower the trace storage costs by selectively dropping traces after combining them into trace objects but before committing them to storage [36, 37, 53].

2.3 Edge-Case Troubleshooting Troubles

Recall that edge-case problem symptoms only manifest in a small fraction of requests, which are undetermined until the problem takes place. We argue that current approaches are ineffectual at getting traces of edge-cases.

Head sampling sacrifices edge-cases. Indiscriminate sampling decisions made at the *beginning* of a request (①), while useful for curbing overhead, *cannot* know a priori whether a request will encounter a rare edge-case problem and should be traced. For edge-case troubleshooting this presents an obstacle: a low head-sampling probability (*e.g.* 0.1%) means a trace of the problem will exist with low probability (*i.e.* 0.1%). The developer may thus have reports that errors took place (UC1) yet the corresponding 'rare' requests were not sampled when those requests began—they lack the detailed cross-machine data necessary for finding the error's root cause. Likewise, the application's high-level metric monitoring may indicate a spike in end-to-end tail-latency (UC2); the developer is thus aware that these high-latency outliers exist, yet without a trace, they cannot localize the problem to a particular component or request class. The situation is even more problematic when investigating bottlenecked queues via temporal provenance (UC3): since each request was sampled independently, the tracing system will have only a vanishing probability that traces of *all* relevant requests in the queue were captured.

Tail sampling sacrifices overheads and scalability. Practitioners have long pointed out a discord between what traces are interesting and what traces get head-sampled [7, 9, 11, 54, 55]. Fortunately, many common edge-case symptoms, including error codes (UC1) and high end-to-end response time (UC2), can be recorded directly within the trace data itself. This enables tail-samplers to explicitly seek out edge-case traces, because at this point (⑥) they can directly inspect the constructed trace object. Today's tail-samplers support filtering traces based on span attributes or metrics, thereby targeting a range of outlier symptoms such as high tail latency, unexpected error codes, uncommon attributes, rare code paths, and undesirable behavior such as RPC retries [7, 10, 32, 40, 49, 53, 58, 67].

Tail-sampling entails enormous costs, however: they must trace all requests and ingest all trace data in order to make informed decisions. Application latency and throughput can suffer if tracing libraries lack optimization (*e.g.* 2× throughput reduction using OpenTelemetry tail-sampling (§6.4)). Ingesting all traces consumes substantial network bandwidth between applications and collectors, interfering with latency-sensitive application traffic (*e.g.* up to 200 MB/s per node (§6.4)). Tail-sampling demands large backend infrastructure investment, deploying enough collectors to receive and process all incoming traces (*e.g.* even one chatty RPC server can overwhelm an OpenTelemetry collector (§6.4)). Even assuming perfect horizontal scaling, tail-sampling requires *e.g.* 100× the collector capacity of 1% head-sampling. Lastly, tracing frameworks are also not robust to bottlenecks mid-way through ingestion (*e.g.* network backpressure) and quickly lose trace coherence when overloaded.

Practitioners sacrifice edge-cases. The justified pragmatism of avoiding large overheads means that head sampling reigns supreme in real-world distributed tracing deployments [21, 26, 30, 34, 64]. Even tail-sampling features of commercial products have low thresholds on data ingestion (*e.g.* <350 kB/s per host [65], <34 spans/s per host [50], <6 MB/s per collector [39]) after which vendors will automatically enable head-sampling or incoherently drop spans. Ultimately, the operator who wishes to troubleshoot edge cases is left unfulfilled.

3 Approach

Hindsight aims to overcome today's trade-off between overheads and edge cases. Our goal is to enable practitioners to target edge-case traces with the flexible criteria of tail sampling, while retaining overheads similar to that of head-sampling, *i.e.* without high application overheads or substantial additional backend infrastructure. We now describe several insights that lead us to Hindsight's *retroactive sampling* approach.

It is not expensive to generate trace data. We don't know a priori whether a request will be an interesting edge-case; only after symptoms manifest. Paradoxically, once we observe symptoms, it is too late to just enable tracing from that point on, as we have already missed the events that led to the anomaly. The only sure-fire way of obtaining coherent traces for any edge-case is to record trace data from the very beginning of the request, for every request.

Tail-sampling does just that—with high overheads and steep infrastructure costs. However, these costs are primarily because today's tracing frameworks tightly couple trace generation with trace ingestion. Ingesting data is expensive, incurring network and backend infrastructure costs. Generating data into local memory is not—outside of distributed tracing, *e.g.*, we observe new technology like Intel PT can generate 100–200 MB/s of processor telemetry per core at 5–15% runtime overhead [28]; likewise method-tracing techniques for Android applications exhaustively record all function entries and exits with <1 ns per tracepoint and <3% runtime overhead [43]. We believe that comparable overheads should be possible for distributed tracing. With careful client library design, applications should be able to generate detailed trace data locally into memory, in anticipation of that data being useful if a problem occurs.

Retroactive sampling: nodes generate, but do not ingest, all trace data.

Symptoms are locally observable. Although root causes are many, varied, and difficult to predict, the same is not true of *symptoms* of problems. For example, error codes, tail latency, and exceptions are easily-observed indicators of potential problems. Many symptoms are localized, programmatically detectable, and manifest quickly at some point during or shortly after a request was served [27, 53, 58]. For example, tail sampling techniques, by definition, require that some span in the trace was explicitly annotated with the symptom of an anomaly, and typically wait only 10 seconds to accumulate trace data [51, 53]. For these common cases it is not necessary to ingest and construct full trace objects when the symptom is so readily detectable at the source. Moreover, since symptoms can be detected independent of traces in the first place, we do not need the expensive indirection of writing symptoms into trace data only to later extract and filter them. We believe that the key to capturing edge-cases is to decouple detection of symptoms from collection of traces.

Retroactive sampling: applications embed **triggers** that programmatically observe symptoms and signal after-the-fact that a trace is an edge-case.

Triggers are local but trace data is distributed. Prior distributed tracing frameworks ingest traces eagerly. We instead believe that traces should be lazily ingested, only in response to a trigger fired at some point during or soon after a request. However, triggers are local – only one machine might detect a symptom, yet the trace data for the request will be dispersed across memory of all machines that serviced it. To splice together a coherent end-to-end trace, all of these other machines need to learn of the trigger and send their slice of the trace to the backend collectors. To identify and notify all relevant machines of a trigger, we thus need the ability to *back-track* the end-to-end path of a request.

Retroactive sampling: requests propagate and deposit **breadcrumbs** so triggers can be shared with all relevant machines.

Trace data will eventually expire. Applications generate trace data into local memory where it incurs no further processing. We only send trace data to collector backends if a trigger fires. However, we cannot predict *when* a trigger might fire – even if a request has finished executing locally, we cannot easily know that the request isn't still executing on some other machine(s) or that a trigger won't fire remotely. Thus, trace data must remain in memory on each machine indefinitely. Over time this will fill memory and eventually we will need to free up space. The intuitive choice is thus to expire trace data for the least-recently-seen request. We call the implicit time duration between generating data and overwriting it the ***event horizon***. We believe that retroactive sampling should not require a large event horizon – as low as tens of seconds is reasonable – because triggers are automatic and shared quickly. In the majority of cases a machine should learn of a trigger within a matter of seconds or milliseconds. Thus retroactive sampling should be feasible even with large and detailed traces or constrained memory.

Retroactive sampling: triggers are best effort; we assume we will see triggers quickly if at all.

4 Design

Overview. Hindsight is a distributed tracing framework that implements retroactive sampling. Whereas typical distributed tracing frameworks eagerly ingest trace data, Hindsight lazily ingests data only after a trigger, thus allowing retroactive sampling of edge-case traces without paying the overhead costs of ingesting all trace data. Hindsight remains compatible with existing head-sampling and tail-sampling policies. Hindsight trivially implements head-sampling policies by firing an immediate trigger upon a positive head-sampling decision (or if the `sampled` flag is set). Hindsight is opaque to backend trace collectors and tail-sampling policies, and existing ingestion pipelines require no changes. Likewise, Hindsight is transparently compatible with existing OpenTelemetry APIs and instrumentation [52], and piggybacks breadcrumbs with OpenTelemetry's context propagation.

Walkthrough. Fig. 2 shows a high-level diagram of Hindsight's main components.

① On request arrival (solid black line) Hindsight generates a unique `traceId` and thereafter propagates it alongside the request, as done by existing frameworks (§2.2).

② Applications record trace data (*e.g.* events, spans) using Hindsight's `tracepoint` client API. This leaves the request's trace data scattered across the machines it visited.

③ A Hindsight ***agent*** runs on each machine to manage trace data. Hindsight agents do not inspect, process, or eagerly report trace data to backends – instead, agents index metadata by `traceId` and await further instruction. For most traces nothing further happens, the trace is not reported, and agents eventually evict old trace data.

④ If an application node observes an outlier symptom (*e.g.* erroneous response, high latency, or a bottlenecked queue) it invokes Hindsight's trigger API and passes the request's `traceId`.

⑤ The local Hindsight agent receives the triggered `traceId`. The full trace remains dispersed across many Hindsight agents, so the local agent informs Hindsight's logically centralized *coordinator* service of the `traceId`. Hindsight's coordinator recursively contacts the set of machines that serviced this request, soliciting breadcrumbs deposited by the request at each machine; a breadcrumb is

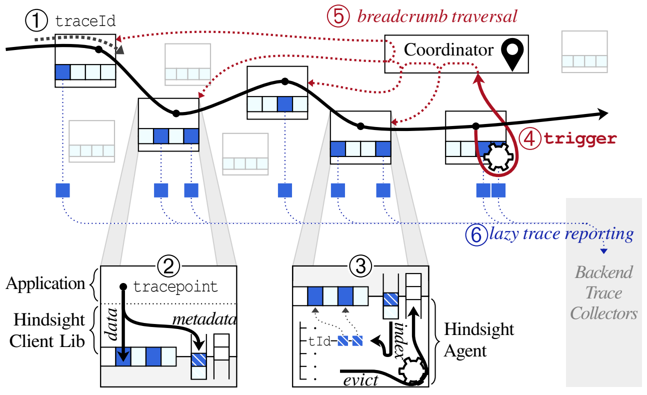


Fig. 2: **The end-to-end lifecycle of a trace in Hindsight** (§4).

a pointer to another machine involved in the request (*e.g.* to the RPC caller or callee).

⑥ Each agent contacted will set aside its slice of data belonging to the `traceId`, and asynchronously send it to the backend collector.

Design decisions. Hindsight is most shaped by three key design choices. First, to prioritize trace coherence as a primary objective throughout the architecture. Second, to maintain an efficient data and control plane split to enable tracing 100% of requests. Finally, to support lightweight programmatic trigger mechanisms.

4.1 Trace Coherence

Coherence is a top-level requirement for distributed tracing (§2.2). As soon as any machine drops data for a trace, the trace is incoherent and effectively useless for troubleshooting. Hindsight's design avoids incoherence in several places.

At ③, agents continually evict old trace data to free up space for new data. Agents do this atomically at the granularity of a trace; there is no point in only dropping part of a trace. However, for a single trace, its data is non-contiguous and fragmented in memory. Agents carefully organize and index metadata about where each trace's data resides and do not simply evict old data in a LIFO manner.

At ⑤, the coordinator must contact all agents that handled a request before those agents overwrite their slice of trace data. Breadcrumbs are a lightweight and scalable solution – the coordinator recursively follows breadcrumbs and only contacts the specific agents known to have serviced the request. This approach takes only a few milliseconds in our evaluation. Breadcrumb traversal is independent of reporting the trace data; agents set aside and asynchronously send trace data to the collector backends after learning of a trigger.

At ⑥, agents can potentially experience network congestion or backpressure from the collector backends, such as in response to a trigger-happy application that fires too many triggers and causes a backlog of unreported trace data on many, or all, agents. Eventually even triggered data must be dropped. Hindsight agents do not drop data arbitrarily (*e.g.*, skipping a full queue) because different agents would then tarnish different victim traces—it only takes one agent dropping its slice of a trace to render the remaining data on other agents practically worthless due to incoherence. Instead, in several places agents use priority queues, with priority determined by consistent hashing of `traceIds`. A given `traceId` will enjoy the same priority across all agents and queues, and the same traces will be dropped by all agents in the face of a bottleneck.

Finally, at ④, applications may fire multiple different triggers for a diverse range of symptoms, using a developer-provided `triggerId` to distinguish different trigger types. Hindsight will prevent a profuse trigger from stifling trace collection of other, low-frequency triggers: agents implement weighted fair sharing for reporting and evicting trace data, with user-defined weights and rate-limits for each `triggerId`.

4.2 Efficient Data Management

Lazy ingestion significantly reduces the volume of trace data sent from agents to the backend trace collection infrastructure. However, within an individual machine, retroactive sampling requires the application generate trace data into local memory for *all* requests (③). The most sensitive performance bottleneck for Hindsight is thus between client applications generating data (`tracepoint`) and the local Hindsight agent that manages trace metadata. Our design establishes a clear split between *control* and *data* activities, which congregates general-purpose data and efficiency in the data plane, and embeds all logic in the control plane.

Data plane. Hindsight's data plane is concerned with efficiently writing trace data from client applications. Using `tracepoint`, applications write trace data to a large shared memory pool subdivided into *buffers*. Different threads write to different buffers; each buffer may only belong to one `traceId` at a time, and threads acquire new buffers when full or when the active `traceId` changes. Consequently, the buffer pool is not consumed sequentially and a single trace may be fragmented across several non-contiguous buffers.

Control plane. Hindsight's agent process encapsulates control plane activities, continually circulating *metadata* about buffers to the application, via two shared memory queues. Applications poll for available buffers and push full buffers; agents poll for full buffers, index metadata of full buffers grouped by `traceId`, and push evicted buffers back to the application. Agents receive triggers and communicate with Hindsight's coordinator, manage breadcrumbs linking the trace data that is strewn across many agents, extract triggered trace data, and report data asynchronously to the backend trace collection infrastructure. Hindsight's control and data distinction yields an efficient agent implementation because agents only touch metadata.

4.3 Triggers

Applications initiate retroactive sampling via Hindsight's trigger API (⑤). In the common case, symptoms are easy to detect and localize: top-level error codes; high latency; increased queue time. Such symptoms can be readily recognized and cheaply computed without the trigger mechanism needing the trace data itself. For example, this may entail adding a trigger call within a service's exception handler, or after checking for outlier latency upon a request's completion. Hindsight provides a library of automatic triggers based on metric percentiles, categorical features, and exceptions. All of our use cases (UC1–UC3) can be implemented using Hindsight's autotriggers. Likewise all existing tail-sampling policies can be implemented using autotriggers, as span-local attribute and metric filters directly translate to metric and categorical autotriggers.

By separating triggers from traces, developers can also implement custom symptom detectors to explicitly decide the conditions for triggering. This further leads to a straightforward integration of triggers into existing metric-monitoring and outlier-detection systems regardless of their architecture.

Lateral traces. Outlier behavior may not map directly to a single request; instead there may be several other related *lateral* requests. For example, to diagnose a bottlenecked queue (UC3), a trigger needs to capture traces for the previous N requests to understand what led to queue buildup [72]; to diagnose a write-ahead log, we desire all requests blocking on a log sync [4, 8]; to diagnose resource contention we require all requests contending for a slow disk or network [3, 5, 6]. By separating triggers from traces, we enable more comprehensive trigger conditions based on factors beyond just a single trace, and triggers that can capture multiple related traces simultaneously. Hindsight enables an application to atomically trigger a group of related lateral `traceIds`; internally Hindsight will ensure that the group as a whole is coherently collected. By comparison, tail sampling cannot easily express cross-trace triggers or sample lateral traces, because `traceId`-based sharding in collector backends is fundamentally at odds with sharing state between traces.

5 Implementation

We have implemented Hindsight's client library in ≈4 KLOC of C and Hindsight's agent and coordinator in ≈5.5 KLOC of Go. We chose C for dataplane efficiency and Go for its ease of use for the more complex control plane logic.

5.1 Data Plane Buffer Pool

Each Hindsight agent pre-allocates a fixed-size *buffer pool* in shared memory for applications to directly write trace data. Hindsight logically subdivides the buffer pool into fixed-size buffers (default 32 kB). Client applications write trace data to buffers via Hindsight's client API. The agent process does not touch data in the buffer pool except when reporting triggered traces. At each point in time, a buffer can only contain trace data of a single request; no two different requests will write trace data to the same buffer at the same time. A single trace will thereby comprise (1) multiple non-contiguous buffers on

`begin(traceId)`	Request begins in the current thread.
`tracepoint({payload})`	Record data for the current trace; `payload` is of arbitrary size in bytes.
`breadcrumb(address)`	Adds a breadcrumb to the current trace, pointing to some other node `address`.
`serialize()`	Obtain the current `traceId` and a `breadcrumb` to the current node.
`end()`	Request ends processing in current thread; flush and remove buffers.
`trigger(traceId,triggerId lateralTraceIds...)`	Instruct Hindsight to collect `traceId` and zero or more `lateralTraceIds`

Table 1: **Hindsight client API.** Applications can invoke the API directly, or indirectly using Hindsight's OpenTelemetry [52] tracer.

each agent and (2) many buffers scattered across numerous agents. Buffers are the granularity of data management within Hindsight. Within clients and agents, a buffer is addressable by its `bufferId`—its offset into the buffer pool.

5.2 Client Library

Table 1 outlines Hindsight's client API. Applications can interact with this API directly, or use Hindsight's OpenTelemetry tracer which acts as a wrapper.

Writing trace data. When a request begins executing in a thread, it must call `begin`; subsequently it may call `tracepoint` an arbitrary number of times; and finally when it completes executing in a thread, it must call `end`. This usage pattern is typical of distributed tracing frameworks. The `tracepoint` function accepts an arbitrary byte payload if called directly; conversely Hindsight's OpenTelemetry tracer serializes trace events as payload. Hindsight internally maintains thread-local state including the current `traceId` and a pointer to a buffer. `tracepoint` writes directly to the thread-local buffer without synchronization. Synchronization is only required when acquiring or returning buffers; these operations touch shared-memory queues but are infrequent. A buffer is acquired during `begin`, returned during `end`, and replaced when filled.

Communicating with agents. The client library acquires `bufferIds` by polling a shared-memory *available queue*; if the queue is empty clients immediately return and instead write trace data to a special 'null buffer' that is simply discarded. When the client fills a buffer, it writes its `traceId` and the `bufferId` to a shared-memory *complete queue*. The agent continually drains the complete queue, and likewise continually returns fresh buffers to the available queue. Shared memory queues are lock-free and support batch operations; using batch operations, agents are robust to queue contention from multiple client writer threads.

This paired channel design forms a natural separator between control and data with two desirable properties: (1) queues only communicate metadata—a single integer `bufferId` represents, by default, a 32 kB buffer; (2) communication is infrequent, occurring only when buffers are filled or a thread switches over to execute a different request, thereby minimizing synchronization. From the client library's perspective, it cheaply and blindly writes trace data into shared memory and forwards only the control metadata to agents; conversely agents are agnostic to buffer contents—they do not inspect data in the shared memory pool and use only the metadata communicated via the complete queue.

`PercentileTrigger(`p`)`	Clients call `addSample(traceID, measurement)`. Trigger fires for measurements >percentile p. (*e.g.* high latency or resource consumption)
`CategoryTrigger(`f`)`	Clients call `addSample(traceID, label)`. Trigger on categorical data that is less frequent than threshold f (*e.g.* rare API calls or attributes)
`ExceptionTrigger`	Trigger on an exception or error code
`TriggerSet(`T`,`N`)`	Tracks the most recent N `traceIds` and includes as `lateralTraceIds` when T fires.

Table 2: **Hindsight autotrigger API** can automatically trigger traces based on certain conditions.

Depositing breadcrumbs. A breadcrumb is an address of a Hindsight agent. When a request arrives at a node, it carries the breadcrumb of the previous node. During trace context deserialization, the `traceId` and breadcrumb is written to a shared memory breadcrumb queue. Agents poll this queue and index breadcrumbs alongside buffer metadata. Agents do not forward or act upon breadcrumbs until a trace is explicitly collected with a trigger. When a request departs a node, it takes that node's breadcrumb. Clients can additionally establish forward-breadcrumbs to a named destination node prior to communication. By following breadcrumbs, we can reconstruct the full request graph starting from any node, including for requests with arbitrary concurrency and fan-out.

Triggering trace collection. Applications initiate trace collection by invoking trigger, which writes the `traceId`, `triggerId` and zero or more `lateralTraceIds` to a shared-memory trigger queue. In addition, Hindsight will propagate the fired trigger with the request similar to the `sampled` flag (cf. Fig. 1) so that later nodes immediately learn of the trigger.

A developer can implement custom outlier detection and invoke trigger directly, or they can make use of Hindsight's autotrigger library (Table 2), a separate collection of triggers that track simple conditions over time and automatically invoke trigger when a condition is met. `TriggerSet` is noteworthy as a building block for lateral tracing; it includes N most recent traces whenever T fires.

5.3 Agent

Trace index. The trace index is a map of metadata, keyed by `traceId`. The metadata for a `traceId` includes a list of `bufferIds` and a list of breadcrumbs. Agents also maintain metadata of the triggers that have fired. Agents continually update the trace index with recently-written buffers, by polling `traceIds` and `bufferIds` from the complete queue. The agent will evict traces when the index exceeds a threshold of buffer pool capacity (default 80%) by removing the least-recently used untriggered `traceId` and returning all of its `bufferIds` to the available queue.

Local triggers. Agents poll the local trigger queue and immediately forward triggers to the coordinator. Agents include the breadcrumbs of the triggered `traceId`, enabling the coordinator to begin recursively disseminating the trigger to other agents. Meanwhile the agent schedules the trigger to be reported. In the case of a spammy local trigger, if the trigger exceeds a per-`triggerId` rate-limit, the agent will immediately discard the trigger instead of forwarding and scheduling it.

Remote triggers. Agents receive remote triggers fired by other agents via the coordinator. To facilitate rapid trigger dissemination, the agent immediately responds to a remote trigger by providing any breadcrumbs it has for the `traceId` and `lateralTraceIds`. Unlike local triggers, agents do not rate-limit remote triggers—they are all scheduled for reporting.

Reporting traces. When a trigger is scheduled for reporting, its `traceId` and lateral `traceIds` can no longer be evicted by the regular buffer eviction cycle. The trigger is inserted into a per-`triggerId` reporting queue. In the normal case when an agent is not backlogged, the reporting queue will be empty. The agent asynchronously pulls triggers from the queues; reads buffers of the `traceId` and `lateralTraceIds` from the buffer pool; sends the buffer contents to the backend collectors; and finally returns the `bufferIds` to the available queue. A trace remains triggered even after reporting its data, in case the request is still generating trace data locally.

Ignoring triggers during overload. If the network or backend collectors are overloaded, reporting queues in an agent can fill up. During overload, the agent continues to report traces as described above for the normal case. The agent implements weighted fair queueing over the reporting queues and supports global and per-`triggerId` reporting rate limits. From a reporting queue, the agent dequeues the *highest-priority trigger*, by using consistent hashing of `traceId`, and reports its data as described above for the normal case.

Simultaneously, past a configured threshold, the agent must begin abandoning triggers to free up buffers. Abandoning a trigger entails removing it from its reporting queue and returning buffers to the available queue. Agents coherently select the *lowest-priority trigger* to abandon, by using the same consistent hashing of `traceId`. In the case of multiple reporting queues, agents will ensure that a well-behaved `triggerId` is not impacted by a spammy `triggerId`: agents implement weighted max-min fair-sharing across reporting queues to choose a queue from which to drop triggers.

Trigger priority ensures coherence during overload. Reporting queues are priority queues that use consistent hashing of `traceId` to determine priority. Across all agents, a given `traceId` will enjoy the same priority relative to other `traceIds`. Thus if multiple agents experience overload, they will coherently bias towards reporting the same high-priority `traceIds` and abandoning the same low-priority `traceIds`.

6 Evaluation

We now evaluate how effectively Hindsight overcomes the fundamental problem of head-based tracing methods in examples (UC1)–(UC3) and meets the goals of retroactive sampling to provide lightweight and effective request tracing.

Systems. We evaluate Hindsight on three distributed systems. To validate our motivating use cases (UC1–UC3), we integrate Hindsight with the Hadoop Distributed File System (HDFS) [63](with a ≈300 LOC JNI-based Java client library) and the DeathStar Social Network Microservices Benchmark (DSB) [24]. To assess Hindsight at greater scale and load, we develop a flexible, configurable RPC benchmark called **MicroBricks**.

MicroBricks is a microservice benchmark written in ≈3 KLOC C++ using gRPC's high-performance `async` library. A MicroBricks deployment comprises a topology of RPC services such that each client request will traverse multiple services. A call to a service will execute for some amount of time, then concurrently call zero or more other RPC services with some probability. Each service is independently configured with its own set of APIs, each with their own execution times, child dependencies, and child call probabilities. We evaluate using several different topologies. In particular, we use Alibaba's microservice trace dataset [42] to derive realistic topologies by calculating per-service execution time distributions, service dependencies, child call probabilities, and client workloads.

Baselines. We configure OpenTelemetry [52] with Jaeger [31] under head-sampling (1% unless indicated) and tail-sampling.

Instrumentation. We instrument MicroBricks with OpenTelemetry to create spans and events for RPC calls and child calls. We use DSB's existing OpenTracing instrumentation and add support for Hindsight. We use Hadoop's existing X-Trace instrumentation [23] and update X-Trace to write its trace data to Hindsight.

Summary. Our experiments demonstrate the following:

- Hindsight effectively addresses the overhead vs. edge-cases trade-off faced by existing tracing frameworks.
- Hindsight captures relevant edge-case traces across real use-cases (UC1–UC3).
- Hindsight is lightweight and not a bottleneck for client applications, unlike OpenTelemetry [52] and Jaeger [31]. Hindsight's trace API imposes nanosecond overheads; Hindsight's impact on end-to-end application latency and throughput is <3.5% when tracing 100% of requests and generating >200 MB/s of trace data per node.
- Hindsight's control/data split provides up to 55 GB/s write throughput.

6.1 Overhead vs. Edge-Cases

In this experiment, we evaluate Hindsight in a large-scale setting with a realistic microservice topology derived from

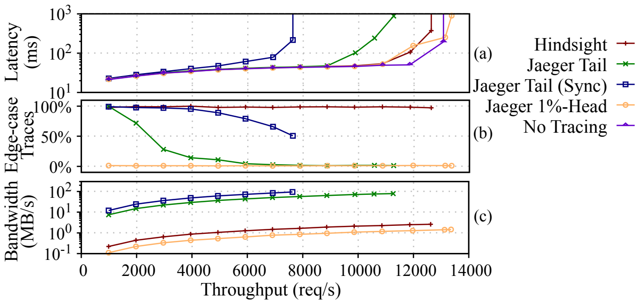


Fig. 3: **Overhead vs. edge-cases** on a 93-service Alibaba MicroBricks topology with 1% edge-cases (§6.1). For different tracing configurations we show: (a) application end-to-end latency-throughput curves; (b) the rate of coherent edge-trace cases captured; and (c) network bandwidth.

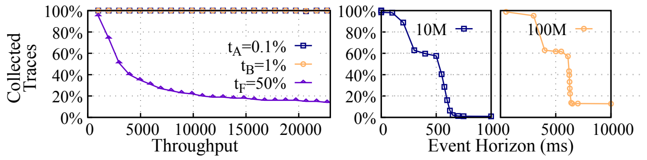
Alibaba request traces [42]. We show that Hindsight overcomes the limitations of head-sampling and tail-sampling.

We deploy MicroBricks with a 93-service Alibaba topology in a 544-core private cluster (comprising 10×Dell R920 48-core 1.5 TB machines and 4×Dell M620 16-core 256 GB machines). We deploy each service in a separate container. We use separate machines to **(i)** generate workload and **(ii)** run the OpenTelemetry collector/Hindsight coordinator+collector.

To directly control the number of edge-case traces, we randomly decide with low probability (1%) to designate a request an edge-case when it completes (later experiments consider autotriggers). We annotate the root span of edge-cases with an additional attribute so that tail-sampling can filter traces on this attribute. Hindsight directly fires a trigger for edge-cases from within MicroBricks. We repeat the experiment multiple times, analyzing results under four tracing configurations:

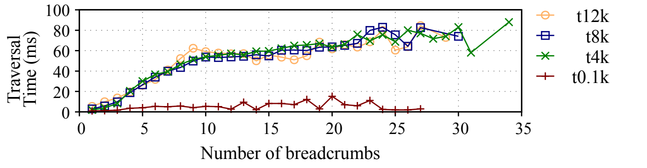
Head sampling (Jaeger 1%-Head). Fig. 3a shows the average request latency and throughput as we vary the offered load from 0 to 14,000 requests/sec (r/s). Jaeger 1%-Head has comparable peak throughput and latency as No Tracing, since it traces only 1% of requests, thus amortizing the tracing overhead. Fig. 3b plots the percentage of coherent edge-case traces captured per second. Since head-sampling cannot discriminate, it only captures ≈1% of all edge-case traces, peaking at 1.64 per second. Fig. 3c shows the network bandwidth consumption between application nodes and the OpenTelemetry collector. With few requests being traced, Head-sampling only consumes a maximum of 1.4 MB/s of network bandwidth.

Tail sampling (Jaeger Tail). Tail-sampling imposes more burden on the traced application than head-sampling, attaining 14% lower peak throughput (Fig. 3a). At low load (1,000 r/s), tail-sampling successfully captures ≈100% of edge-case traces, at 9.9 per second (Fig. 3b). However, a load of just 2,000 r/s is sufficient for clients to encounter backpressure from the network and the OpenTelemetry collector, and they begin incoherently dropping spans: at 2,000 r/s only 71% coherent edge-cases are captured; at 3,000 r/s only 28%; and so on. Tail-sampling rapidly deteriorates and at peak load captures *fewer* coherent edge-case traces than head-sampling



(a) Coherent traces captured when overloaded with a spammy trigger t_F.

(b) Event horizon for constrained bufferpools (10MB and 100MB).



(c) Breadcrumb traversal time as trace size varies, triggering 0.1% (t0.1k) or 50% (t4k, t8k, t12k) traces on different workloads.

Fig. 4: **Scalability and overload.**

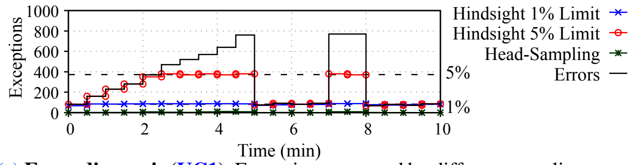
(1.44 edge-cases/s), because 98.8% of captured traces are incoherent. Tail-sampling consumes up to 78 MB/s of network bandwidth (Fig. 3c).

Tail sampling (Jaeger Tail Sync). Jaeger clients asynchronously send spans to OpenTelemetry collectors, and as we just observed, drop spans when client-side queues fill up. We repeat the experiment with a synchronous variant, whereby clients send spans to OpenTelemetry synchronously. Backpressure then manifests as additional critical-path request latency. This approach inevitably increases request latency and reduces peak throughput by 42% (Fig. 3a). However, we can observe the collector ultimately captures more edge-case traces, peaking at 47 edge-cases per second at 6,000 r/s (Fig. 3b) and 72.2 MB/s of network. Beyond this, the OpenTelemetry collector is saturated and cannot process a higher rate of traces; it begins indiscriminately dropping incoming spans, reducing the fraction of coherent edge-case traces.

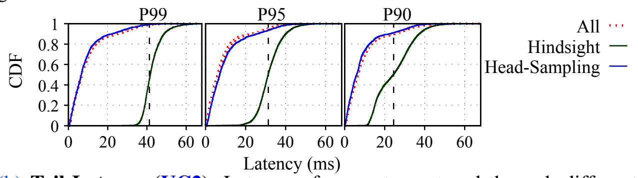
Hindsight. Hindsight achieves comparable peak throughput to No Tracing (<3.5%), and minimal impact on request latency below peak load (Fig. 3a). Hindsight captures 99–100% of edge-case traces at all throughputs (Fig. 3b). Hindsight consumes a maximum of 2.6 MB/s of network bandwidth since only edge-case traces are being collected (Fig. 3c).

6.2 Scalability and Overload

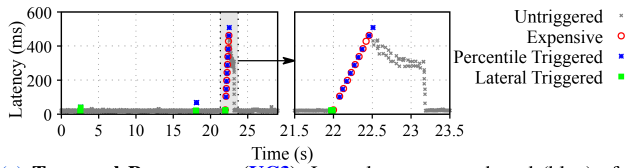
We now focus on two aspects of Hindsight's scalability: its breadcrumb traversal mechanism and its ability to rate-limit spammy triggers. We deploy the 93-service Alibaba topology as described in §6.1. To reach a higher request and trace throughput, we scale down the computation performed at each service and increase offered load up to 28,000 r/s. We install three triggers with probabilities t_A=0.1%, t_B=1%, and t_F=50%. t_F represents a faulty trigger—it fires for 50% of requests and thereby adds substantial load to Hindsight's breadcrumb traversal mechanism. We rate-limit Hindsight's collector bandwidth to 1 MB/s per agent to backlog the agents and inhibit Hindsight's ability to collect traces; thus t_F triggers far more traces than Hindsight can collect.



(a) **Error diagnosis (UC1).** Exceptions captured by different sampling strategies as the error rate varies.



(b) **Tail-Latency (UC2).** Latency of requests captured through different sampling strategies with different tail-latency triggers (left to right).



(c) **Temporal Provenance (UC3).** Lateral requests gathered (blue) after triggering on slow requests (red) due to an overfull queue in HDFS.

Fig. 5: **Hindsight applied on use cases UC1–UC3** (see §2.1).

Coherent rate-limiting. Fig. 4a plots the percentage of coherent traces captured for $\mathtt{t_A}$,$\mathtt{t_B}$ and $\mathtt{t_F}$ as the offered load increases. Throughout the experiment, Hindsight captures approximately 100% of traces triggered by $\mathtt{t_A}$ and $\mathtt{t_B}$, since they fire infrequently. By contrast, $\mathtt{t_F}$ triggers far more traces than can be collected. In absolute terms, Hindsight collects ≈2,000 coherent traces per second throughout the experiment, with $\mathtt{t_F}$ using capacity not used by $\mathtt{t_A}$ and $\mathtt{t_B}$. Thus, higher request rates result in more traces dropped for $\mathtt{t_F}$ in both relative and absolute terms.

Breadcrumb traversal. Fig. 4c plots the average breadcrumb traversal time based on the trace size – *i.e.* the number of Hindsight agents that were recursively contacted. We show results for four experiment iterations and label them based on their approximate trigger rates: `t12k`, `t8k`, and `t4k` correspond to triggering 50% traces on 24k, 16k and 8k r/s workloads (≈12k, 8k, and 4k triggers/second respectively). To compare to a non-overloaded setting we also include `t0 .1k`, a 12k r/s workload from §6.1 (≈0.1k triggers per second). Traversal time is elevated for `t12k`, `t8k` and `t4k` (up to 86 ms) since spammy triggers substantially increase the load on Hindsight's coordinator. Conversely, traversal time for `t0.1k` is <13 ms since triggers are relatively infrequent. For each experiment, traversal time increases with trace size, but sub-linearly since breadcrumbs can be gathered concurrently from different branches in requests that have fan-out. However, even under the extremely overloaded circumstance, the longest traversal time, which is less than 100 ms, is far smaller than the event horizon as described in the following section and thus is still manageable.

Event horizon. We lastly measure Hindsight's event horizon. Here, we introduce a delay when an agent receives a local trigger. We vary the delay added to triggers and measure how many coherent traces are ultimately collected. At a certain point, triggers will have too much delay and trace data will have been evicted before the trigger even fires. Fig. 4b plots the percentage of coherent traces captured for $\mathtt{t_B}$ as we vary the trigger delay. We repeat this experiment with small buffer pools (100 MB and 10 MB per agent) to exacerbate the event horizon effect. Even a 10 MB buffer pool can capture nearly 100% coherent traces in the absence of added delays, but a 500 ms delay drops coherence to 58% and at 600 ms, coherence is <20%. A larger buffer pool improves the tolerance to delays: with a 100 MB buffer pool, coherence surpasses 90% with up to 3s delay, but drops to <20% by 6.4 s. In practice, we believe our default 1 GB pool is a reasonable choice, bringing an event horizon around 1 minute.

6.3 Case Studies

We now turn our attention to the case studies introduced in §2.1, and demonstrate how Hindsight's local triggers are able to support these use cases.

Error diagnosis (UC1). We deploy DSB Social Network, a microservice system with 12 microservices and 17 backends [24], on 13 CloudLab `c6320` nodes [20]. We add an `ExceptionTrigger` from Hindsight's autotrigger library to the ComposePostService, and run DSB's default workload with 300 r/s[1]. We randomly inject exceptions in the ComposePostService module, with exception rates ranging from 1% to 10%. We repeat the experiment twice and rate-limit Hindsight's collector to approximately 1% and 5% of the total trace data generated by the experiment. Fig. 5a plots the exception rate, and the number of coherent exceptional traces captured, for each 30 s time window. When there are few exceptions, Hindsight captures all traces; when the exception rate exceeds collector bandwidth, Hindsight coherently captures as many traces as possible within this limit.

Tail-latency (UC2). We add a `PercentileTrigger` from Hindsight's autotrigger library to the ComposePostService module in the same setting as above, invoking `addSample` at the end of each ComposePost RPC call and providing the measured RPC duration. We set p to 99, 95, and 90, as different thresholds for tail latency. We inject 10% requests at random with 20–30 ms latency. Fig. 5b plots the latency distribution of requests captured by different strategies; the vertical dotted lines mark the tail-latency percentile threshold. Hindsight is able to specifically target traces with high-percentile latency. By contrast, head-sampling is random and thus its captured latency distribution resembles that of all requests – useful for aggregate analysis but not for edge-case troubleshooting. We note that Hindsight does not sacrifice this aggregate analysis use-case; it supports both simultaneously (cf. §6.1).

Temporal provenance (UC3). We add a `QueueTrigger`

[1]We measure a maximum attainable DSB throughput of ≈350 r/s.

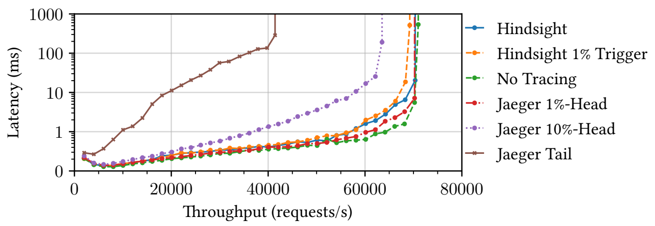


Fig. 6: **End-to-end latency and throughput** for a 2-service MicroBricks topology configured with various tracers, showing minimal application impact for Hindsight despite tracing 100% of requests.

from Hindsight's autotrigger library to the HDFS NameNode queue — the `QueueTrigger` combines a `TriggerSet` with a `PercentileTrigger`, parameterized to capture $N = 10$ most recently dequeued lateral requests when 99.99th percentile queueing latency is observed. We deploy HDFS on 10 machines (8 DataNodes, 1 NameNode, and 1 client) and run a Hindsight agent on each machine. We run a closed-loop workload of random 8 kB reads with 10 concurrent requests.

Fig. 5c (left) shows NameNode queue latency over time. We inject a burst of 10 expensive `createfile` requests 21 seconds into the trace that briefly saturate the queue—Fig. 5c (right) zooms in on this time window. The figure shows high-latency requests (•), requests that fire the autotrigger (**X**), and the additional lateral requests that were triggered to Hindsight (**X**). The first expensive request occurred at 22 seconds, followed by a pause while it was executed. Upon dequeuing the subsequent `read8k` request, `QueueTrigger` fired due to high queue latency, and Hindsight retroactively sampled the 10 prior traces leading up to the trigger. The sample included the culprit expensive request. Overall, all 10 expensive requests were sampled, 8 unrelated requests prior to the first expensive request, and 9 additional `read8k` requests. Moreover, several intermittent latency spikes occurred unrelated to the experiment (Fig. 5c, left), which Hindsight also captured; upon investigation, these were due to garbage collection.

Unlike UC1 and UC2, temporal provenance is unsupported in existing tail-samplers. Moreover, temporal provenance is fundamentally difficult to support with tail-sampling due to scalability issues. Temporal provenance requires knowledge of lateral traces (*e.g.* the 10 previous traces); by implication those traces must all route to the same collector instance. However in practice, tail-sampling necessarily uses `traceId` for routing decisions – thus related traces may arrive at different, oblivious collectors.

6.4 Hindsight Performance

End-to-end application overheads. Hindsight generates trace data for all requests; thus low overheads are a key goal of Hindsight's design. In this experiment, we measure the impact of Hindsight on end-to-end application latency and throughput. We deploy a two-service MicroBricks topology with a 100% call probability from the first service to the second. To highlight tracing overheads, neither service performs additional compute. We vary the offered load and measure

API Call	T=1	T=4	T=8	API Call	T=1	T=4	T=8
`begin`	72.7	194.8	237.9	`tracepoint`	7.9	8.4	8.6
`end`	70.7	205.8	216.6				
`Category(.01)`	45.8	44.9	46.7	`tracepoint` 8 B	3.9	4.0	4.8
`Percentile(99)`	275.3	293.5	306.9	`tracepoint` 128 B	11.5	13.5	13.0
`Percentile(99.9)`	407.1	441.9	512.2	`tracepoint` 512 B	37.7	43.1	40.9
`Percentile(99.99)`	629.4	875.8	1134.0	`tracepoint` 2 kB	160.2	192.9	174.7
`TriggerSet(10)`	6.57	44.1	52.2				

Table 3: Latency measurements (nanoseconds) for Hindsight client API and autotriggers for a microbenchmark application configured with **1**, **4**, and **8 Threads** (§6.4). Default `tracepoint` writes a 32 kB trace event; we also measure 8–2048 B `tracepoint` payloads.

end-to-end request latency and throughput (§A.1).

Fig. 6 plots latency-throughput curves under several different tracing configurations. The lowest latency and highest throughput is achieved with No Tracing, peaking at an average 71.0 k requests/s. Similar throughput is achieved by Jaeger when configured with 1% Head-sampling, at 70.2 k r/s. Hindsight peaks at 70.4 k r/s – a decrease of only 0.9% compared to no tracing. Hindsight generates on average 330 MB/s of trace data at peak request throughput, with an event horizon of 5.2 s, and consumes a combined 0.3 CPU cores across agents, coordinator, and collector. By comparison, Jaeger configured with Tail-sampling peaks at only 41.4 k r/s, an overhead of 41.7%; moreover, the workload over-saturates the OpenTelemetry collector, resulting in 94% of trace data being dropped while consuming 4.5 CPU cores.

Client API and autotrigger microbenchmarks. We run a benchmark application that generates traces and measures the overhead of calls to Hindsight's client API and autotrigger library. The benchmark writes traces by calling `begin` to start the trace, writing a total of 16 kB per trace by repeatedly calling `tracepoint`, then calling `end` to finish the trace. Each `tracepoint` call writes a 32-byte event `struct` (3 metadata fields and a timestamp) using Hindsight's OpenTelemetry library. Following each trace, the benchmark invokes five different autotriggers. The benchmark runs a custom number of threads to generate traces; each thread independently runs a continuous loop generating traces, and every thread writes a different trace. We configure Hindsight to use 32 kB buffers and a 1 GB buffer pool, and run a Hindsight agent. We run 1 minute per experiment (≈10–50 million traces).

As shown in Table 3, autotrigger overheads vary. CategoryTrigger is relatively cheap (<47 ns) and TriggerSet adds relatively little overhead to the wrapped trigger (6–53 ns). By contrast, PercentileTrigger overheads grow proportional to the percentile: up to 307, 512, and 1,134 ns respectively for tracking 99th, 99.9th, and 99.99th percentile latency due to larger internal data structures for tracking order statistics.

Table 3 also shows API latency for 1, 4, and 8 threads. Overall, Hindsight achieves nanosecond-scale API latency, and by design the expensive API calls (`begin`, `end`, and autotriggers) are limited to once per trace. `begin` and `end` vary from 70–230 ns, proportional to the number of threads due to contending on shared-memory queues to acquire and return buffers. By contrast, `tracepoint` call latency is mostly independent of the number of threads, between 7.9–8.5 ns (reduced to ≈4 ns when omitting timestamps). We also measure `tracepoint` latency for larger payloads up to 2 kB; latency increases only up to 175 ns per tracepoint since `tracepoint` is primarily a memory copy into the thread-local buffer established by `begin`.

7 Discussion

We next discuss items peripheral to Hindsight's core design.

7.1 Triggers

Mitigating spammy triggers. Hindsight's design currently isolates triggers based on a trigger ID, whereby different symptom detectors would use different trigger IDs, ensuring that a symptom detector that fires infrequently is not affected by one that fires too often.

Lateral trace IDs. All of Hindsight's autotriggers are lightweight symptom detectors that run within the application itself. In principle, the logic to decide when to trigger, and which `traceIds` to trigger, is arbitrary. Hindsight's autotriggers are classes which the application can instantiate that track state over time. For a `PercentileTrigger` that tracks latency, for example, the application must instantiate the autotrigger within the request handler, and add the new latency sample at the end of each request's execution. A `TriggerSet` can wrap any trigger; internally it maintains a sliding window of the *N* most recently-seen `traceIds` that tested the wrapped trigger. When we applied the `TriggerSet` in the UC3 experiment (Fig. 5c), for example, we measured queueing latency and the TriggerSet internally held the *N* most recent `traceIds` that were dequeued. When the wrapped autotrigger finally calls trigger, it will include all *N* of the `traceIds` in the trigger call.

7.2 Consistent Hashing

Hindsight agents have several forms of queuing and scheduling internally, primarily to decide which traces to evict and which to report to avoid any unbounded queue. Simple queueing (*e.g.* used by OpenTelemetry) indiscriminately drops data when the queue is full. When multiple agents have full queues, each independently dropping arbitrary data seriously compromises trace coherence. Instead, when Hindsight agents are at capacity, they bias towards dropping data from the same victim traces by preferentially discarding items from lowest priority traces. Thus even though Hindsight agents are operating independently, they seek to retain the same high-priority trace IDs when under load.

7.3 The Event Horizon

Parameters. Several factors influence Hindsight's event horizon: (i) the buffer pool size of each agent; (ii) the rate of new trace data being generated; (iii) the time between a request completing and a trigger firing. Inevitably, if there is too much trace data, or if triggers are too slow, Hindsight may be unable to keep the trace before its data is overwritten. For some use cases this means Hindsight cannot use retroactive sampling. However, head-sampling or tail-sampling would still be viable options, equivalent to existing distributed tracing frameworks.

Extending the event horizon. The solution is either to increase the memory available to Hindsight or to scale down the percentage of traced requests using Hindsight's optional *trace percentage*. Trace percentage is a separate configuration knob (defaulting to 100%) that controls the percentage of requests that generate trace data in the first place. The starting premise for Hindsight is that 100% tracing is acceptable, so we used 100% as the default and described Hindsight as such throughout this paper. However, if an application has overhead constraints or limited memory for a buffer pool, the percentage of requests that are traced in the first place can be scaled back. Hindsight enforces scale-back coherently across agents through consistent `traceId` hashing: *e.g.* 50% trace percentage will halve the trace data throughput and double the event horizon.

Mismatched and dynamic event horizons. The global event horizon of an application is dictated by the shortest event horizon among the constituent processes, since the whole trace becomes incoherent the moment the first agent evicts any of its data. This fundamental property of Hindsight can be addressed by enlarging the buffer pool memory on higher throughput nodes. Moreover, the buffer pool needs not be of fixed size. We considered implementing a dynamically-sized buffer pool, *e.g.* that can be configured with a target event horizon, but ultimately chose a fixed-size buffer pool to better bound memory overheads – a desirable property for telemetry systems [71].

Shared buffer pools. In our current design, we deploy one Hindsight agent per traced application process. If multiple containers share a machine, as in our experiments, several agents may run on the same machine. There is no reason why applications could not share a single machine-wide buffer pool, enabling processes to stock their buffer pool capacities and average out the differences between their event horizons.

7.4 Comparison with Tail Sampling

Event horizons. Hindsight's event horizon has an analogue in tail sampling. Since trace collectors cannot immediately perform tail sampling the instant trace data arrives, and must wait for all of the slices of a trace to arrive from all of the machines the request visited. Today, this is done with a timeout (*e.g.* 30s by default in OpenTelemetry [52]), after which the trace objects are constructed and tail samplers can be evaluated. If the application generates a high volume of trace data, then the trace collector can potentially run out of memory while awaiting data for to do tail sampling.

Tail sampling expressivity. Today's tail samplers focus on filters and outliers applied to span attributes and metrics. Yet a tail sampling decision for one trace cannot influence the sampling decision of other traces. By contrast, Hindsight's lateral traces enable a trigger to specify other, related traces, in addition to the one exhibiting a symptom, allowing it to support use cases like temporal provenance (UC3). Tail samplers do not support such use cases, and they would be challenging to introduce due to the way trace data for different traces route to different collectors based on their `traceId`.

7.5 Robustness

Application failures. If the application process crashes (*e.g.* SEGV/NPE-type crashes), then Hindsight preserves problematic traces since Hindsight's agent continues to run and the trace data is preserved in memory in the shared buffer pool. The agent will also be able to continue responding to breadcrumbs. This is a secondary benefit of externalizing trace data on the critical path of requests, and is currently supported by Hindsight. By contrast, existing distributed tracing frameworks buffer trace data in application memory and would lose unreported data upon an application crash.

Agent failures. If Hindsight's agent crashes (irrespective of whether the application process crashes), then the buffer pool will still exist in-memory on the machine and could be later retrieved to inspect the state just prior to the crash. Hindsight does not currently implement such a recovery process. In addition, if an agent crashes, it will by default prevent Hindsight's coordinator from following breadcrumbs through this crashed agent. This can be overcome by extending Hindsight to propagate breadcrumbs for the last N visited nodes instead of just one; this would both avoid $(N-1)$-hop failures and also speed up Hindsight's collection process.

Kernel and hardware failures. In the case of kernel crashes or hardware failure, application-level traces are only useful if it was the application's behavior that triggered the crash. In this case, Hindsight's data would be lost.

8 Related Work

Distributed tracing. Numerous prior works identify end-to-end requests as a useful granularity for slicing telemetry data and troubleshooting distributed systems. Example use cases include detecting anomalous request structures [37,64,72], diagnosing changes in the steady-state [16,57,61], modeling workloads [46,70], and identifying resource and queue contention [25,44,72]. Distributed tracing systems have been presented in industry [34,64], as open-source tools [31,52,56,78], and in academia [23,45]. Edge-case troubleshooting stands in tension with overheads in distributed tracing, and head-sampling and tail-sampling offer alternative points in this space (§2.2).

Logging frameworks. Distributed tracing is the cousin of log ingestion frameworks that collect and store application-level log data [13,66]. Log ingestion frameworks are agnostic to concepts like requests, do not record or group log data by requests, and cannot control head-sampling decisions coherently for requests – instead applications generate simple sequential streams of log data all at the same level of logging detail. Consequently, logs are typically far less detailed than distributed tracing and log ingestion frameworks handle a lower volume of data. For example, Chukwa reports on average 10kB/s per node [13]; Splunk limits to 330 kB/s per node [66]; Amazon CloudWatch limits to 5MB/s per log stream [12]. Early distributed tracing works rejected the idea of building distributed tracing atop logging, citing coherence challenges from brittle data, enormous post-processing costs, and fundamental scalability bottlenecks [17, 34, 64]. In practice, trace detail is typically far greater than even non-production debug-level logging [64], and it is easy to see why: head-sampling gives operators leeway to instrument their applications at fine detail, because they can amortize the high cost of a single trace by scaling down the number of collected traces. By comparison, log ingestion frameworks have no such opportunity.

Network provenance. Hindsight is similar in spirit to network packet provenance systems that chronicle the history of network state, enabling use cases such as tracking the origin or path traversed by a packet across the network. Earlier systems, like ExSPAN [77] and SNP [75], adopt this abstraction; more recent works like SyNDB [35] and SPP [14] apply network provenance for packet-level root-cause analysis on Internet scale. Packet provenance systems primarily trace only packet metadata, which is well-structured and can be summarized in-band; these systems tackle additional trust challenges outside of Hindsight's purview. By contrast, handling metadata to reconstruct the path of a trace is but one concern for Hindsight; Hindsight is focused on handling arbitrary payloads (*i.e.* trace data), and the resulting performance, coherence, and fairness challenges. Hindsight also draws inspiration from works focused on temporal provenance [76] and packet reputation [15] in distributed systems, although Hindsight's tracing abstractions operate entirely at the application level.

9 Conclusion

Hindsight circumvents the false dilemma between overhead and usefulness for diagnosing symptomatic edge cases by providing developers detailed traces from the recent past when they encounter symptoms of failures. We believe the retroactive sampling abstraction, and our Hindsight implementation of it, can shift the conversation around tracing away from mechanism (how to collect traces) to a question of policy (what traces should be collected), and allow distributed tracing systems to support edge-cases analysis: a key use case for which they were originally conceived.

Acknowledgements

We are grateful to our shepherd, Harsha Madhyastha, and the anonymous reviewers for their insightful feedback that helped improve our work.

References

[1] Businesses Losing $700 Billion a Year to IT Downtime, Says IHS. Retrieved April 2022 from https://www.businesswire.com/news/home/20160125005188/en/Businesses-Losing-700-Billion-a-Year-to-IT-Downtime-Says-IHS.

[2] Recent AWS outage and how you could have avoided downtime. Retrieved April 2022 from https://medium.com/@datapath_io/recent-aws-outage-and-how-you-could-have-avoided-downtime-7d9d9443d776.

[3] HDFS-3751: DN should log warnings for lengthy disk IOs. Retrieved April 2022 from https://issues.apache.org/jira/browse/HDFS-3751, 2014.

[4] HBASE-8228: Investigate time taken to snapshot memstore. Retrieved April 2022 from https://issues.apache.org/jira/browse/HDFS-8228, 2015.

[5] HBASE-8744: Enable HBase to log the entire latency profile for HDFS packets resulting in slow writes. Retrieved April 2022 from https://issues.apache.org/jira/browse/HDFS-8744, 2016.

[6] HDFS-11461: DataNode Disk Outlier Detection. Retrieved April 2022 from https://issues.apache.org/jira/browse/HDFS-11461, 2017.

[7] Jaeger Issue 425: Discuss post-trace (tail-based) sampling. Retrieved April 2022 from https://github.com/jaegertracing/jaeger/issues/425, 2017.

[8] HDFS-6110: adding more slow action log in critical write path. Retrieved April 2022 from https://issues.apache.org/jira/browse/HDFS-6110, 2018.

[9] Jaeger Issue 1861: Delayed Sampling. Retrieved April 2022 from https://github.com/jaegertracing/jaeger/issues/1861, 2019.

[10] Annanay Agarwal. How Grafana Labs enables horizontally scalable tail sampling in the OpenTelemetry Collector. Retrieved April 2022 from https://grafana.com/blog/2020/06/18/how-grafana-labs-enables-horizontally-scalable-tail-sampling-in-the-opentelemetry-collector/, 2020.

[11] Narayanan Arunachalam. Zipkin Secondary Sampling. Retrieved April 2022 from https://github.com/openzipkin-contrib/zipkin-secondary-sampling, 2019.

[12] AWS. AWS CloudWatch Logs quotas. Retrieved April 2022 from https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/cloudwatch_limits_cwl.html, 2022.

[13] Jerome Boulon, Andy Konwinski, Runping Qi, Ariel Rabkin, Eric Yang, and Mac Yang. Chukwa, a large-scale monitoring system. In *Proceedings of CCA*, volume 8, pages 1–5, 2008.

[14] Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. One primitive to diagnose them all: Architectural support for internet diagnostics. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 374–388, 2017.

[15] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 115–128, New York, NY, USA, 2016. Association for Computing Machinery.

[16] Mike Y Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. In *1st USENIX Symposium on Networked Systems Design & Implementation (NSDI'04)*, pages 23–23, 2004.

[17] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas 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)*, 2014.

[18] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013.

[19] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[20] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 1–14, 2019.

[21] elastic. Transaction Sampling. Retrieved April 2022 from https://www.elastic.co/guide/en/apm/guide/current/sampling.html#sampling.

[22] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS)*, 30(4):1–35, 2012.

[23] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. X-trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI'07)*, 2007.

[24] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, pages 3–18, 2019.

[25] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, pages 19–33, 2019.

[26] honeycomb.io. Getting At The Good Stuff: How To Sample Traces in Honeycomb. Technical report, honeycomb.io, 2019.

[27] Peng Huang, Chuanxiong Guo, Jacob R Lorch, Lidong Zhou, and Yingnong Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[28] Intel Corporation. *Intel 64 and IA-32 architectures software developer's manual*, volume 3 (3A, 3B, 3C & 3D): System Programming Guide. Intel, 2016.

[29] Irving Popovetsky. Getting At The Good Stuff: How To Sample Traces in Honeycomb. Retrieved April 2022 from `https://www.honeycomb.io/blog/getting-at-the-good-stuff-how-to-sample-traces-in-honeycomb/`, 2020.

[30] Ivan Topolnjak. Kamon: How to Keep Traces for Slow and Failed Requests. Retrieved April 2022 from `https://kamon.io/blog/how-to-keep-traces-for-slow-and-failed-requests/`, 2021.

[31] Jaeger: Open Source, End-to-End Distributed Tracing. Retrieved April 2022 from `https://www.jaegertracing.io/`.

[32] Jeremy Castile. What You Need to Know About Distributed Tracing and Sampling. Retrieved April 2022 from `https://thenewstack.io/what-you-need-to-know-about-distributed-tracing-and-sampling/`, 2020.

[33] Chris Jones, John Wilkes, Niall Murphy, and Cody Smith. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016. `https://landing.google.com/sre/sre-book/chapters/service-level-objectives/`.

[34] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)*, pages 34–50, 2017.

[35] Pravein Govindan Kannan, Nishant Budhdev, Raj Joshi, and Mun Choon Chan. Debugging transient faults in data centers using synchronized network-wide packet histories. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 253–268. USENIX Association, April 2021.

[36] Pedro Las-Casas, Jonathan Mace, Dorgival Guedes, and Rodrigo Fonseca. Weighted Sampling of Execution Traces: Capturing More Needles and Less Hay. In *9th ACM Symposium on Cloud Computing (SOCC '18)*, 2018.

[37] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. Sifter: Scalable sampling for distributed traces, without feature engineering. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'19)*, pages 312–324, 2019.

[38] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[39] Lightstep. Learn about Microsatellites: How many Microsatellites do I need? Retrieved April 2022 from `https://docs.lightstep.com/docs/learn-about-micro-satellites`.

[40] Lightstep. OpenTelemetry-Collector Issue #4758: Tail-Based Sampling Scalability Issues. Retrieved April 2022 from `https://github.com/open-telemetry/opentelemetry-collector-contrib/issues/4758`, 2020.

[41] Liang Luo, Suman Nath, Lenin Ravindranath Sivalingam, Madan Musuvathi, and Luis Ceze. Troubleshooting transiently-recurring errors in production systems with blame-proportional logging. In *2018 USENIX Annual Technical Conference (USENIX ATC'18)*, pages 321–334, 2018.

[42] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.

[43] Yu Luo, Kirk Rodrigues, Lijin Jiang, Bing Xia, David Lion, and Ding Yuan. Hubble: Performance Debugging with In-Production, Just-In-Time Method Tracing on Android. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[44] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted Resource Management in Multi-Tenant Distributed Systems. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, 2015.

[45] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *25th ACM Symposium on Operating Systems Principles (SOSP'15)*, 2015.

[46] Gideon Mann, Mark Sandler, Darja Krushevskaja, Sudipto Guha, and Eyal Even-Dar. Modeling the parallel execution of black-box services. In *Proceedings of USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'11)*, 2011.

[47] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[48] Pulkit A Misra, María F Borge, Íñigo Goiri, Alvin R Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. Managing tail latency in datacenter-scale file systems under production constraints. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

[49] New Relic. Technical distributed tracing details: Tail-based sampling algorithms. Retrieved April 2022 from https://docs.newrelic.com/docs/distributed-tracing/concepts/how-new-relic-distributed-tracing-works/#tail-sampling-strategy.

[50] New Relic. Technical distributed tracing details: Trace limits. Retrieved April 2022 from https://docs.newrelic.com/docs/distributed-tracing/concepts/how-new-relic-distributed-tracing-works/#limits.

[51] New Relic. Tail-based sampling (Infinite Tracing). Retrieved April 2022 from https://docs.newrelic.com/docs/understand-dependencies/distributed-tracing/get-started/how-new-relic-distributed-tracing-works#tail-based, 2020.

[52] OpenTelemetry: An Observability Framework for Cloud-Native Software. Retrieved April 2022 from http://opentelemetry.io/.

[53] OpenTelemetry. Tail Sampling Processor. Retrieved April 2022 from https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/processor/tailsamplingprocessor.

[54] OpenTelemetry Specification Issue 307: Allow samplers to be called during different moments in the Span lifetime. Retrieved April 2022 from https://github.com/open-telemetry/opentelemetry-specification/issues/307, 2019.

[55] OpenTelemetry Enhancement Proposal 115: Allow Additional Sampling Hooks. Retrieved April 2022 from https://github.com/open-telemetry/oteps/pull/115, 2020.

[56] OpenTracing: Vendor-Neutral APIs and Instrumentation for Distributed Tracing. Retrieved April 2022 from http://opentracing.io/.

[57] Krzysztof Ostrowski, Gideon Mann, and Mark Sandler. Diagnosing latency in multi-tier black-box services. In *4th International Workshop on Large-Scale Distributed Systems and Middleware (LADIS'11)*, 2011.

[58] Maulik Pandey. Building Netflix's Distributed Tracing Infrastructure. Retrieved April 2022 from https://netflixtechblog.com/building-netflixs-distributed-tracing-infrastructure-bb856c319304, 2019.

[59] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, and Rebecca Isaacs. *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media, 2020.

[60] Raja R Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H Sigelman, Rodrigo Fonseca, and Gregory R Ganger. Principled workflow-centric tracing of distributed systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 401–414, 2016.

[61] Raja R Sambasivan, Alice X Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R Ganger. Diagnosing performance changes by comparing request flows. In *8th USENIX Symposium on Networked Systems Design & Implementation (NSDI'11)*, volume 5, pages 1–1, 2011.

[62] Yuri Shkuro. *Mastering Distributed Tracing*. Packt Publishing, Feb 2019.

[63] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

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

[65] Splunk. Observability Cloud Usage, Subscription Limits Enforcement, and Entitlements. Retrieved April 2022 from `https://www.splunk.com/en_us/legal/usage-subscription-limits-enforcement-and-entitlements.html`, 2022.

[66] Splunk. Splunk Enterprise Capacity Planning Manual - Summary of performance recommendations. Retrieved April 2022 from `https://docs.splunk.com/Documentation/Splunk/8.2.6/Capacity/Summaryofperformancerecommendations`, 2022.

[67] Splunk. Use cases: Troubleshoot errors and monitor application performance using Splunk APM. Retrieved April 2022 from `https://docs.splunk.com/Observability/apm/apm-use-cases/apm-use-cases-intro.html#nav-Use-cases:-Troubleshoot-errors-and-monitor-application-performance`, 2022.

[68] Cindy Sridharan. *Distributed Systems Observability*. O'Reilly Media, 2018.

[69] Kun Suo, Jia Rao, Luwei Cheng, and Francis CM Lau. Time capsule: Tracing packet latency across different layers in virtualized systems. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 1–9, 2016.

[70] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. Stardust: Tracking Activity in a Distributed Storage System. In *2006 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '06)*, 2006.

[71] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*, Bordeaux, France, 2015.

[72] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. Zeno: diagnosing performance problems with temporal provenance. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, pages 395–420, 2019.

[73] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, pages 159–172, 2011.

[74] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 456–468. IEEE, 2016.

[75] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure network provenance. In *Proceedings of the twenty-third ACM symposium on operating systems principles*, pages 295–310, 2011.

[76] Wenchao Zhou, Suyog Mapara, Yiqing Ren, Yang Li, Andreas Haeberlen, Zachary Ives, Boon Thau Loo, and Micah Sherr. Distributed time-aware provenance. *Proceedings of the VLDB Endowment*, 6(2):49–60, 2012.

[77] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 615–626, 2010.

[78] Zipkin: A Distributed Tracing System. Retrieved April 2022 from `http://zipkin.io/`.

A Supplemental Experiment Results

A.1 End-to-end Application Overheads

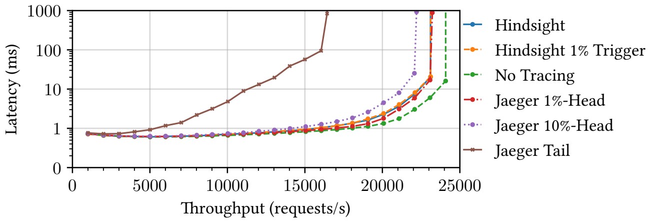


Fig. 7: **End-to-end latency and throughput** for a 2-service MicroBricks topology configured with various tracers, demonstrating minimal application impact for Hindsight despite tracing 100% of requests.

Fig. 7 shows a variant of the experiment described in §6.4. In the original experiment the services are configured to perform no additional compute. We additionally repeat the experiment with services configured to perform approximately 100 microseconds of matrix-multiply compute per service. We observe similar to trends to those discussed in §6.4; in particular Hindsight has a comparable latency and throughput profile to Jaeger configured with 1% head-sampling.

A.2 Head-Sampling and Tail-Sampling Overheads.

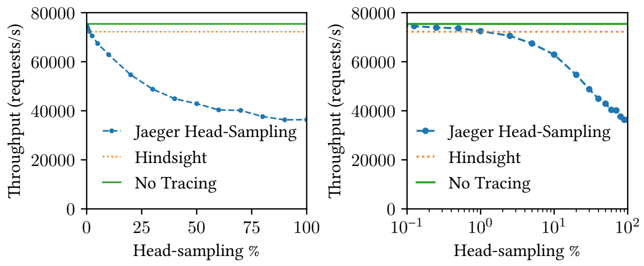


Fig. 8: **Head-sampling impact on end-to-end throughput** of a 2-service MicroBricks topology under a closed-loop workload. We vary the head-sampling percentage configured for OpenTelemetry Jaeger. The right figure plots the same data with a log-scale x-axis to highlight overheads at low head-sampling percentages. Tail-sampling is equivalent to 100% Head-sampling

We further measure the application-level impact of different head-sampling regimes. We run the application described in Fig. 8 and submit a closed-loop workload to saturate the system. We measure the application-level throughput achieved with by OpenTelemetry and Jaeger configured with different head-sampling percentages. We compare the throughput to that of Hindsight and with No Tracing. The results illustrate that the OpenTelemetry Jaeger overheads at typical low sampling percentages (<1%) is negligible, but the client library performance deteriorates at higher tracing percentages. 100% head-sampling is equivalent to tail-sampling.

A.3 Client throughput

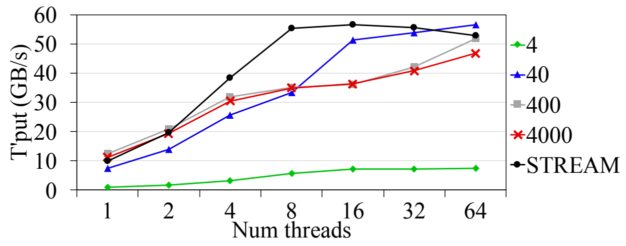


Fig. 9: **Client Throughput** achieved by a microbenchmark that varies the number of threads and the size of payload to `tracepoint` calls. Even modest payloads (40 bytes) can saturate memory bandwidth.

The purpose of this experiment is to evaluate the `tracepoint` write throughput that client applications can achieve, based on the payload size to `tracepoint` calls. The experiment demonstrates the peak attainable data ingestion throughput for different numbers of threads and different payload sizes. Larger payloads can attain higher throughput, but even with small payloads (40 bytes), we can saturate memory bandwidth.

This experiment configures Hindsight to use 32 kB buffers and a 1 GB buffer pool. We run a client application comprising between 1 and 64 threads. Each thread continually writes traces in a loop to Hindsight. Writing a trace entails calling `begin`, 100 `tracepoints`, then `end`. We repeat the experiment for different numbers of threads and varying the size of `tracepoint` calls from 4 bytes to 4000 bytes, resulting in traces between 400 and 400,000 bytes in size.

Fig. 9 plots the throughput achieved in GB/s. Small payloads of 4 bytes fail to fully saturate memory bandwidth, achieving only 887 MB/s with one thread and peaking at 7.55 GB/s with 64 threads. By contrast, even a modest increase in payload size to 40 bytes is enough to nearly saturate memory bandwidth; with 400 byte payloads, we achieve throughput of 12.5 GB/s on a single core. We include in Fig. 9 measurements of peak memory bandwidth from the STREAM benchmark [47].

Hindsight achieves high throughput, despite each trace acquiring and writing a new buffer at an arbitrary non-sequential offset in the buffer pool. This occurs because Hindsight's client library coordinates buffers only during `begin` and `end` (at the start and end of each trace respectively) and stores the buffer pointer thread-local in the interim. Calls to `tracepoint` are then little more than a memory copy to the thread-local buffer acquired by `begin`.

A.4 Control-Data Trade-offs

Hindsight's design emphasizes a control-data split, to enable applications to write trace data at large volume while reducing the amount of indexing work agents must perform. The main factor influencing this trade-off is Hindsight's buffer size. With large buffers, agents index fewer buffers and thus

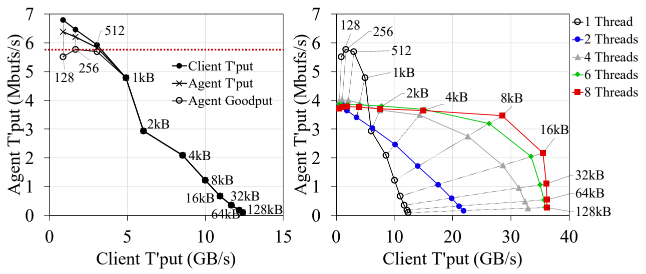


Fig. 10: **Buffer size trade-off.** Each data point is annotated with the Hindsight buffer size. Small buffers require more indexing work from agents, while large buffers are less memory efficient by exacerbating internal fragmentation.

perform less work; however it may exacerbate internal fragmentation when traces only partially fill buffers. Conversely, small buffers are more space-efficient, but require more indexing work from agents. We evaluate this trade-off by measuring client-side and agent-side throughputs, while varying Hindsight's internal buffer size from very small (128 B) to very large buffers (128 kB).

We run the benchmark application with one thread, 100 kB traces, and a payload of 1 kB per `tracepoint` call (Hindsight fragments payloads across multiple buffers when necessary). Fig. 10 (left) plots the client-side throughput of generating data (x-axis) and the agent-side throughput of indexing buffers (y-axis). We annotate data points with the corresponding buffer size used. Large buffer sizes (128 kB) can support peak client data throughput (12.1 GB/s) while requiring little of the agent. Conversely, tiny buffer sizes (128 B) stress the agent buffer throughput since we more frequently cycle buffers through the queues. Fig. 10 (left) plots three lines and indicates two important phenomena. The client throughput line plots the rate at which the client writes buffers, whereas the agent throughput line plots the rate at which the agent cycles buffers; the delta in-between are 'null buffers', written by the client because the available queue is empty, *i.e.* the agent cannot keep up. Writing to null buffers means lost trace data; the third line, agent goodput, only counts buffers of coherent traces, *i.e.* excluding all buffers for traces that lost data. We observe that the goodput with 128 B buffers is lower than with 256 B buffers due to greater loss. In general, with $\geq$1 kB buffers, the agent is able to consistently keep up without losing data.

Fig. 10 repeats this experiment with varying numbers of threads, and plots client-side data throughput and agent-side buffer goodput. Buffer sizes of 16 kB and higher are sufficient for reaching peak write throughput while remaining comfortably within agent throughput limits; by default, we select 32 kB for Hindsight.