10.48047/jocaaa.2026.35.03.25

Attributing the Unseen: High-Precision CPU Profiling via Trace-Join Analytics

Varun Raj
Independent Researcher, USA

Abstract

One of the major challenges of modern distributed systems is to understand CPU consumption patterns. Existing profiling tools cannot indicate which experiments or traffic patterns caused the CPU to be busy. This creates challenges for developers in diagnosing performance issues. The proposed system uses unique sequence tagging to correlate profiling with attribution. Each profile is tagged with the id of the associated request, which is persisted through the profile's execution, and is joined at the batch analytics stage with the request metadata, including the experiment identifiers, the latency classifications and the traffic slice identification. Developers can query the dataset for profiles matching their attribution settings, then visualize the result in a flame graph. The flame graph is a method of visualizing profiles in an easy to understand manner, displaying what paths through code are consuming too much CPU in the context of an experiment. By enabling attribution-based profiling, moved performance debugging efforts from relying on infrastructure engineers and experts to be self-serve by product developers. As a result, performance regressions can be detected earlier in the development cycle and their occurrence prevented, saving computational resources and protecting system stability in production environments serving global traffic. The same attribution can be applied on other dimensions, such as memory or network profiling.

Keywords: CPU Profiling, Distributed Systems, Performance Attribution, Trace Analytics, Flame Graphs

10.48047/jocaaa.2026.35.03.25

1. Introduction

In large distributed systems, aggregate metrics such as total CPU usage often hide small overhead introduced by a particular experiment, a slice of traffic, or a deployment. Customary profiling of computational resources provides a snapshot view over the entire system, without semantic understanding of code paths. For profiling systems to be helpful to developers, they have to answer certain attribution questions, for example, why is the CPU busy? The question of who owns that consumption becomes more complex when the system grows or when there are more than one development team involved.

The problem is exacerbated when an organization has thousands of A/B tests and feature flags running at the same time, such that regressions in performance can go undetected for a long time. Context-aware profiling has the potential to reduce debugging time and utilize fewer resources for experiments needing to be run on a larger scale. Techniques such as weighted sampling may help in this [1]. Modern telemetry systems must strike a balance between enabling complete observability and the cost of storing, processing, and analyzing the telemetry [2].

The tracer system we describe allows us to collect random CPU samples at runtime and to create useful diagnostic data from these samples to attribute computational costs to the experiment and the latency tier. We describe our proposed solution in terms of a trace-join analytics framework that bridges execution-time profiling to post-completion context. Through the use of per-sequence tagging and batch joins, developers have a mechanism for observing the effect of localized changes on performance at a global level, resolving the gap between when profiling data is available and when contextual information for making this attribution is available.

2. The Limitations of Decoupled Sampling

The main issue with attributing CPU cost is temporal: the total latency of a request is only known when the request is complete, but CPU profiles are usually sampled periodically during program execution. Customary sampling tools measure what the CPU is doing at any given millisecond, and have no sense of whether that work is a fast query or a tail query. That temporal decoupling results in a major gap in information, complicating the performance assessment.

Current methods of profiling distributed systems with statistical sampling operate by sampling periodically at interrupts and saving a stack trace. While low impact, they lose the meaningful context necessary for attribution, particularly in modern microservices architectures where calls and execution flow between several services. Zero-code instrumentation systems show that thorough tracing in service meshes is possible. Leading systems increase over 95% coverage of application-level transactions without changing the application code [3]. They incur less than 1% CPU overhead even on high-throughput services. By contrast, instrumentation with 3-5% overhead is typical of most approaches, which when scaled to the user volume expected for production use, is prohibitive. But state-of-the-art tracing systems can obtain 98.7% accuracy in reconstructing distributed traces across microservice boundaries, and 85 to 90% for manual instrumentation through code insertion.

Attribution is more difficult in the cloud as a shared infrastructure and multi-tenancy further complicate the landscape of applications. For example, performance tracing for microservices in the cloud entails searching through a highly connected dependency graph to extract information because a single client request may result in hundreds of internal service calls. An analysis of production microservices in large-scale systems shows many inter-service dependencies; another analysis of 260 microservices in

10.48047/jocaaa.2026.35.03.25

production gives insights into performance [4]. Thorough monitoring systems, analyzing more than 48 million performance failures, characterize failures but identifying the service or code path that caused a performance degradant remains challenging in the absence of failure attribution mechanisms. The decoupling in time between the collection of performance data and its associated context can complicate root cause analysis.

Another issue for profiling multi-tenant systems, such as a cluster where many services share the compute resources, is attribution of CPU usage to specific services. Performance overhead from continuous profiling also needs to be measured and kept low in order not to interfere with production workloads. However, because many production systems sample data at levels acceptable for high service rates, there are also blind spots in observability where periodic performance problems will not be detected. Because they are decoupled, customary sampling loses the ability to trace back from aggregate metrics to individual requests or experimental designs. The network infrastructure is a common source of performance degradation. In one study, 47.3% of production performance problems were caused by the network infrastructure [4]. In particular, among the network problems, the virtual network components had the highest failure rate of 30.8%. The performance bottlenecks are mainly located in the application layer (32.7%), followed by computing infrastructure (12.7%) and external traffic spikes due to attack traffic and other causes (7.3%).

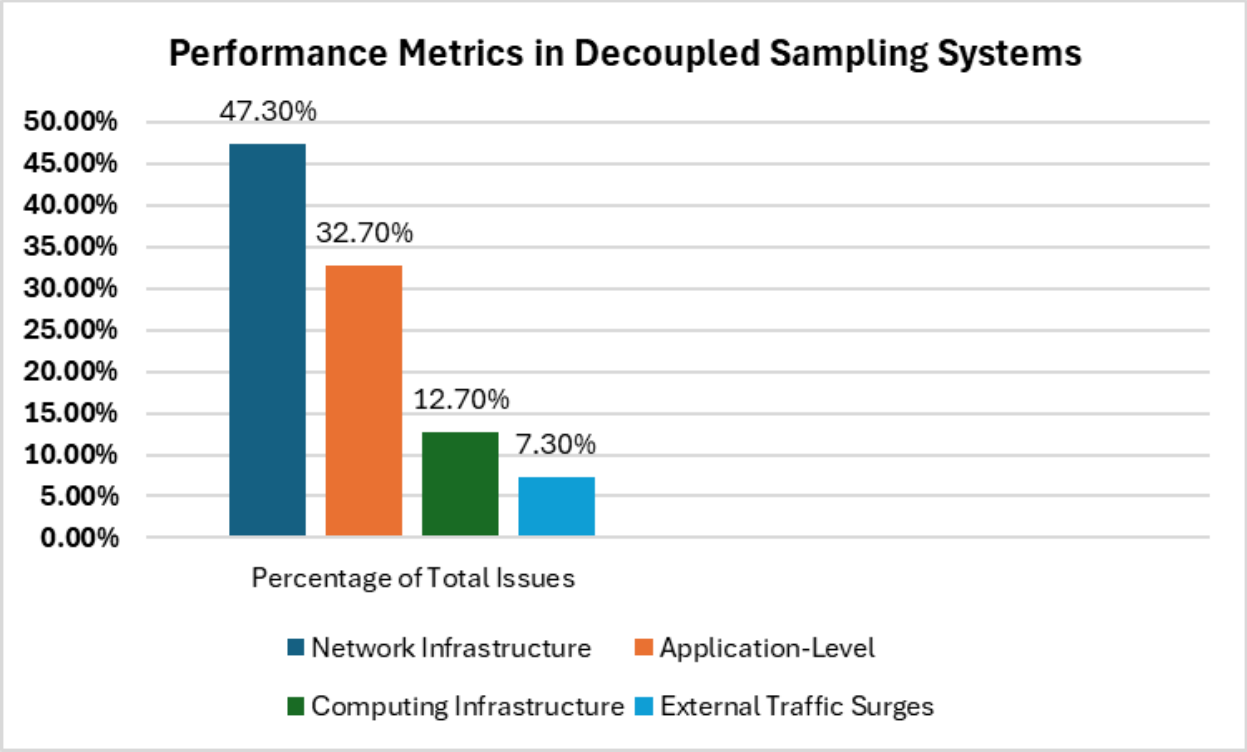

Fig. 1: Performance Metrics in Decoupled Sampling Systems [3, 4]

3. Proposed Solution: The Unique Sequence Tagging Model

To allow the attribution of execution, a tagging mechanism associates a sequence number (unique to the request) with all profiling data the CPU sends in a profile sample. The sequence number is part of the

10.48047/jocaaa.2026.35.03.25

profiling header structure. A new sequence number is given to each request. This ensures that each stack trace can be correlated to the request for which it was collected. It separates the concerns of collecting the data from the concerns of interpreting it.

The implementation of sequence number generation must take into account its performance impact, ideally making the cost of generating the sequence number negligible to the point that it could be generated on every request without introducing measurable latency. Using monotonically increasing identifiers with host-specific prefixes gives good uniqueness guarantees without considerable cost. The implementation uses atomic counter operations along with instance identifiers to generate globally unique sequence numbers, which remain stable even when threads are moved across different processors, allowing reliable correlation in highly concurrent environments.

This tagging mechanism uses an existing profiling infrastructure. It extends standardized profiling headers with a sequence number field. When the profiling subsystem takes a CPU sample, it reads the sequence number from thread-local storage, rather than going through complex mechanisms to propagate context. This allows our sequence numbers to be tied directly to the requests, even if work is done in a different thread. Studies on tail latency management have shown that request tagging is effective for fine-grained QoS without introducing a meaningful overhead to the critical path [5]. This allows us to delegate the complexity away from the serving path, preserving the performance guarantees of customary sampling based profiling, while providing rich attribution functionalities.

The system architecture separates the hot path (serving requests) from the cold path (attribution processing), and only the sequence number associations need to be kept during the request execution. However, all complicated join operations, metadata lookups, and index updates are performed asynchronously after the request is completed so that attribution does not impact user-facing latency or system throughput. The reduction of critical path operations has been discussed as a method of improving performance in high-throughput distributed systems [6].

System Component	Implementation Approach	Key Characteristics	Design Purpose
Sequence Number Generation	Atomic counter operations with instance identifiers	Lightweight, globally unique	Minimal latency impact on every request
Identifier Persistence	Monotonically increasing with host-specific prefixes	Stable across thread migrations	Uniqueness guarantees in concurrent environments
Integration Method	Extension of standard profiling headers	Sequence number field addition	Seamless integration with existing infrastructure
Context Propagation	Thread-local storage	No complex propagation mechanisms	Maintains association across distributed work
Architecture Design	Hot path vs. cold path decoupling	Attribution processing occurs asynchronously	Zero impact on user-facing latency
Critical Path	Minimal tagging	Complex operations	Maintains traditional

10.48047/jocaaa.2026.35.03.25

Operations	overhead only	deferred to offline processing	profiling performance

Table 2: Unique Sequence Tagging System Components [5, 6]

10.48047/jocaaa.2026.35.03.25

4. Batch Join Analytics and Metadata Enrichment

This framework is particularly useful in post-processing in large batch join jobs, where distributed batch jobs use the unique sequence number as the join key to join production logs to the CPU profile. Production logs include context information that is only available at the completion of a request, such as the total latency bucket, the entire set of active experiment identifiers, the traffic slice, and other request properties. The join operation transforms the raw profiling samples into improved performance records that can answer queries.

Batch Processing Pipeline: Consumes profiling data and request logs stored in distributed storage systems. The separation of data collection and context enrichment improves system reliability and improves analysis options. Instead, profiling samples and request metadata are streamed periodically to the data lake infrastructure, where they are consumed by batch analytics. This streaming architecture enables near real-time performance analysis, while retaining the fault tolerance of batch architectures. Evaluation frameworks have to take care that the measurements are statistically sound, since the execution time can vary considerably across runs [7].

The join operation is technically challenging at scale because profiling systems can generate millions of samples per second. The implementation is based on distributed hash join algorithms. The profiling samples and request logs are divided into partitions by their sequence numbers, and a local join is performed for each partition. It then re-tags profiling samples based on the metadata in their request log entry. The modern implementations of RPC in datacenter environments show that sub-microsecond tail latencies can be achieved for RPC requests with efficient protocol design, even under constrained load conditions. They have also provided a template to reduce the overheads in the telemetry pipeline [8]. The distributed hash join can be used in parallel in a cluster of nodes.

Every profile sample, with its complete set of metadata, including its experiment IDs, latency bins, service versions, and regions, is stored in a searchable database. To store and retrieve the data efficiently, the database is partitioned in multiple dimensions, allowing arbitrary query patterns to be run while optimizing query execution time. To ease this, developers can query profiles against attribution dimensions using a query interface that uses Boolean expressions, e.g. "show me all CPU profiles for experiment X where latency exceeded threshold Y in geographic region Z".

10.48047/jocaaa.2026.35.03.25

Pipeline Stage	Technical Implementation	Operational Characteristics	Analytical Capability
Data Collection	Continuous streaming to data lake	Profiling samples and request metadata	Near-real-time with batch fault tolerance
Join Algorithm	Distributed hash join	Partitioned by sequence number	Parallel processing across cluster nodes
Join Key	Unique sequence number	Links CPU profiles with production logs	Correlates execution with completion context
Metadata Enrichment	Local join per partition	Adds latency, experiments, traffic slice data	Comprehensive attribution information
Indexing Strategy	Multi-dimensional partitioning	Optimized for query performance	Efficient retrieval by attribution criteria
Query Interface	Boolean combination support	Experiment, latency, geography filters	Complex analytical queries across dimensions
Context Addition	Post-completion enrichment	Total latency, experiment IDs, traffic slice	Available only after request completion

Table 3: Batch Join Analytics Pipeline Architecture [7, 8]

5. Visualization via Aggregated Flame Graphs

Once this profiling data is properly improved, indexed and made accessible through the visualization layer, each individual profile may be visualized together with the others as a flame graph on a single page. The x-axis is the profile population, the y-axis is the stack depth within the profile. The boxes correspond to functions in the call stack and their widths are proportional to the CPU time the function consumed. This allows for more intuitive identification of resource allocations in a given attributed request context.

Aggregation combines thousands of profiling samples that match a single attribution rule into a single visualization. When a developer provides the identifier of their experiment, all profiles that match that identifier are returned and improved. It then creates a single flame graph of the total CPU consumption. Since any sample taken from a single profile only shows a small part of the program's behavior, when several samples with the same attribution attributes are aggregated, the flame graph can display statistically important code paths in the program. This telemetry aggregation process allows for performance analysis and debugging of distributed systems.

A flame graph provides a model that is easier to understand than a call graph and provides intuitive mapping to the call stack structure. For developers, flame graphs make it easy to identify CPU hotspots by representing their cost as a horizontal width that graphically corresponds to the amount of time. This allows developers to focus their optimization efforts on the right functions. Fine-grained dependency analysis is able to identify the portions of the critical path that contribute most to request latency [10]. Performance optimizations can reduce page load time by an average of 34% across different web application workloads and can be as high as 59% in terms of elapsed time for complex applications that have deep dependency graphs. For a diverse set of application scenarios, time-to-first-byte improved by an average of 23% after the system identified and parallelized those resource operations that are

10.48047/jocaaa.2026.35.03.25

independent: the median case improved by 2.1x. Strikingly, when the system deeply optimized those resource operations that are dependent on other resource operations, the 95th percentile improved by 3.4x. The visualization system has interactive features, allowing the developer to zoom-in on performance data for a specific stack frame by clicking on it or by filtering for other attribution dimensions. The system allows developers to compare two experiments to identify their differences and visualize them using differential flame graphs. Differential views give direct answers to the questions of what changed and by how much. In comparison mode, the stack frames are sorted by increasing or decreasing CPU consumption and regressions pop out clearly.

The flame graph visualizes the profiles aggregated from all a developer's profiling runs of a specific experiment. In the resulting diagram, developers can quickly find where code is using too many resources. This may be due to serialization overhead, logging, or sometimes hidden performance costs, such as cache behavior. It is particularly useful for examining performance regressions that only appear under certain conditions or on certain traffic slices. By creating the attribution information, developers are able to see not only that performance got worse, but which experiment or traffic pattern caused the regression and which code paths were affected.

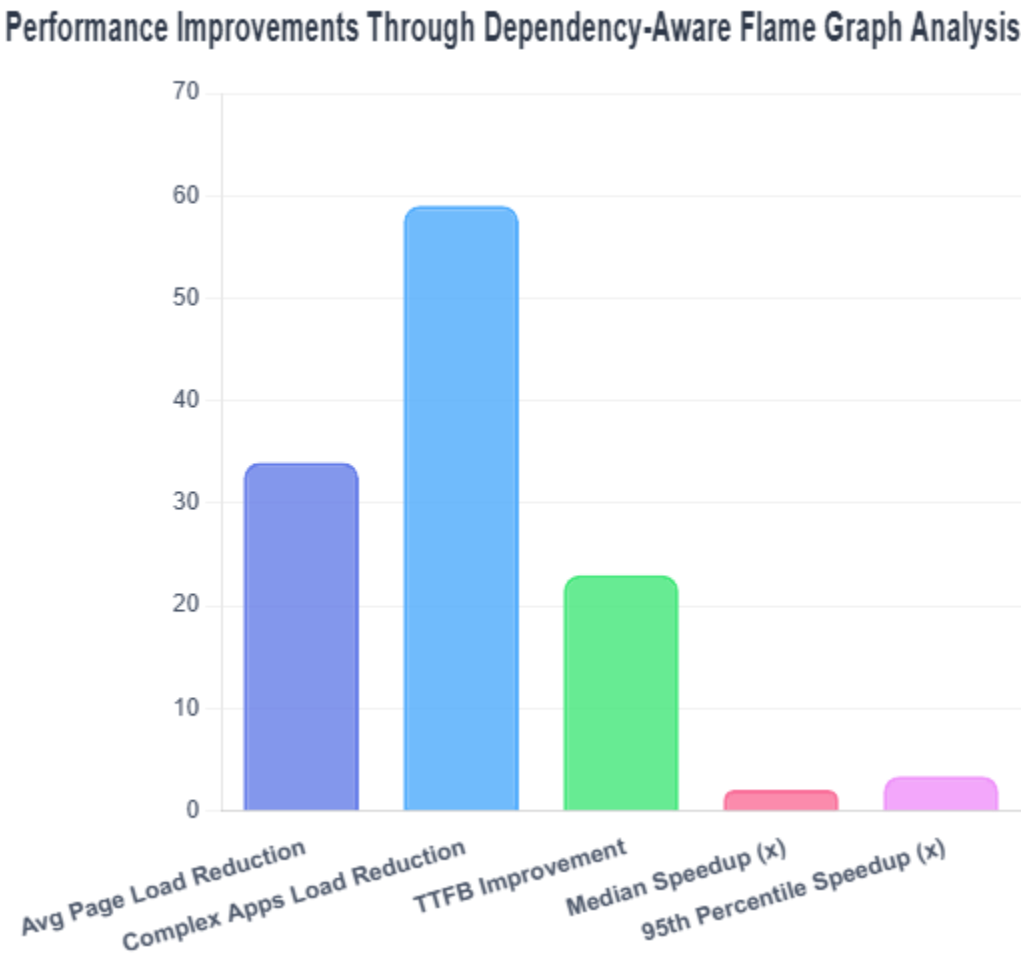

Fig. 2: Performance Optimization via Flame Graph Analysis [9, 10]

10.48047/jocaaa.2026.35.03.25

10.48047/jocaaa.2026.35.03.25

Conclusion

Attribution-enabled profiling fundamentally changes the performance engineering of large-scale distributed systems by providing developers with knowledge of how their code affects resources. This is achieved by tying together sequence tagging with sampling of the program's execution and attaching that data to its context after the fact. Batch join analytics allows raw profiling data to be transformed into performance history for queries. Engineers can filter profiles by experiment identifiers, latency tiers, and traffic patterns. Flame graphs help make sense of complex performance data. No need to rely on infrastructure engineers for performance debugging. Self-service capabilities lead to a high code quality and improved development velocity. Performance regressions are caught earlier, and the stability of the system increases by finding inefficient code paths before production traffic passes through them. Also, resource usage can often be improved through optimizations. The allocation model is not limited to CPU profiling and can include modeling other resources, such as memory and network I/O. The importance of attribution increases with the complexity of the distributed system. Seeing the results of their changes inspires developers to focus on performance, and the system has generally not added important infrastructure costs or computational overhead to the multiple experimental launches it has supported.

10.48047/jocaaa.2026.35.03.25

References

1. Anna Ben-Hamou, et al., "Weighted sampling without replacement," arXiv, 2017. Available: https://imstat.org/wp-content/uploads/import/BJPS359.pdf
2. Shir Landau-Feibish, et al., "Compact Data Structures for Network Telemetry," ACM Computing Surveys, 2025. Available: https://dl.acm.org/doi/10.1145/3716819
3. Junxian Shen, et al., "Network-Centric Distributed TracingwithDeepFlow:Troubleshooting Your MicroservicesinZeroCode," Yunshan Networks, 2023. Available: https://mmlab.snu.ac.kr/wp-content/uploads/2025/06/5BSIGCOMM27235D20Network-Centric20Distributed20Tracing20with20DeepFlow20Troubleshooting20Your20Microservices20in20Zero20Code-20250609-073054.pdf
4. Yu Gan, et al., "Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices," ACM Digital Library, 2019. Available: https://dl.acm.org/doi/10.1145/3297858.3304004
5. Timothy Zhu, et al., "PriorityMeister: Tail Latency QoS for Shared Networked Storage," in ACM Digital Library, 2014. Available: https://www.pdl.cmu.edu/PDL-FTP/CloudComputing/pmeister-SoCC14.pdf
6. Brendan Gregg, "Systems performance: Enterprise and the cloud," 2nd Edn., Pearson Education, 2021. Available: https://ptgmedia.pearsoncmg.com/images/9780136820154/samplepages/9780136820154_Sample.pdf
7. Charlie Curtsinger and Emery D. Berger, "STABILIZER: Statistically Sound Performance Evaluation," ASPLOS'13, 2013. Available: https://people.cs.umass.edu/~emery/pubs/stabilizer-asplos13.pdf
8. Anuj Kalia, et al., "Datacenter RPCs can be General and Fast," USENIX, 2019. Available: https://www.usenix.org/system/files/nsdi19-kalia.pdf
9. Francisco Romero, "INFaaS: Automated Model-less Inference Serving," USENIX, 2021. Available: https://www.usenix.org/system/files/atc21-romero.pdf
10. Ravi Netravali, et al., "Polaris: faster page loads using fine-grained dependency tracking," ACM Digital Library, 2016. Available: https://dl.acm.org/doi/10.5555/2930611.2930620