Mach: Firefighting Time-Critical Issues in Complex Systems Using High-Frequency Telemetry

Franco Solleza
Brown University
franco_solleza@brown.edu

Shihang Li
University of Washington
shli@cs.washington.edu

William Sun
Brown University
william_sun@brown.edu

Richard Tang
Brown University
richard_tang@brown.edu

Malte Schwarzkopf
Brown University
malte@cs.brown.edu

Nesime Tatbul
Intel Labs and MIT
tatbul@csail.mit.edu

Andrew Crotty
Northwestern University
andrew.crotty@northwestern.edu

David Cohen
Intel
david.e.cohen@intel.com

Stan Zdonik
Brown University
sbz@cs.brown.edu

ABSTRACT

To understand the complex interactions in modern software, engineers often rely on high-frequency telemetry (HFT) data generated via tools like eBPF. However, today's database systems are too slow for HFT's rate and volume and cannot process HFT within the limited resources available on individual host machines.

Mach is a new storage engine for collecting and querying HFT. Key to Mach is the Temporal Skip Log (TSL)—a lightweight, write-optimized, log-based data structure specialized for HFT. Mach supports high ingest rates and makes data immediately queryable while operating within a limited on-host resource envelope.

Our demo shows how Mach helps engineers collect and query HFT in near real-time when diagnosing performance problems. In contrast, current systems and data reduction techniques fail to keep up. While a widely used time series database (InfluxDB) drops much of the HFT, the audience will see how Mach loses no data and allows them to interactively explore HFT from application and kernel events as they arrive.

PVLDB Reference Format:
Franco Solleza, Shihang Li, William Sun, Richard Tang, Malte Schwarzkopf, Nesime Tatbul, Andrew Crotty, David Cohen, and Stan Zdonik. Mach: Firefighting Time-Critical Issues in Complex Systems Using High-Frequency Telemetry. PVLDB, 17(12): 4425 - 4428, 2024.
doi:10.14778/3685800.3685891

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/fsolleza/mach-vldb.

1 INTRODUCTION

When managing system deployments, engineers frequently need to "firefight" in deployment scenarios like system outages, bug reports, and performance regressions. Because deployments are complex, many scenarios involve subtle interactions between components across different layers in the stack. For example, this demo covers one such issue where processes interfere due to OS scheduling decisions. During these firefighting scenarios, engineers go through a data-centric, iterative, human-in-the-loop process to reason about the system's state. They rely on their knowledge of the system infrastructure to formulate hypotheses and then verify them by correlating available data in real-time while iteratively drilling down to the issue's root cause.

In real-world settings, engineers typically face two main challenges. First, monitoring, logging, and tracing data—collectively referred to as "telemetry"—available from telemetry collection tools are often insufficient to characterize complex interactions. Typical telemetry collection tools rely on coarse-grained sampling or aggregation techniques to keep up with the vast amount of telemetry that the system generates. Although aggregate and sampled data provide a high-level overview of system state, they are less useful for diagnosing specific issues. Second, to address the lack of fine-grained data, engineers usually begin collecting *high-frequency telemetry* (HFT) using tools like `perf`, `tcpdump`, or `bpftrace` after learning of an issue. In modern systems, HFT is ubiquitous: an in-memory key-value store can generate millions of operations per second; multi-threaded applications cause millions of systems calls per second; and fine-grained hardware event sampling (e.g., branch mispredictions) is common in tools like `perf` and VTune.

In this demo, we show that even a basic key-value store application already generates as many as 1M events per second, with the Linux kernel adding another 150K scheduler events per second. By design, current HFT collection tools specialize to capture and summarize HFT from a single, specific source. Some tools build in analyses to keep up with HFT: most `bpftrace` tools output histograms of the needed telemetry, discarding data afterward. Other tools take a record-then-analyze approach: an engineer might run `perf record` for some time and then use `perf report` to analyze the captured data. However, this approach leaves the engineer in the dark about whether they actually captured any relevant data from a potentially intermittent event.

Thus, using the specialized, siloed HFT tooling available to them today, engineers cannot easily identify complex correlations across

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685891

components in real-time. Instead, they rely on fuzzy mental mapping to perform these time-based correlations. Wouldn't it be nice to store HFT from various sources in a single location that an engineer could then query to quickly diagnose the issue?

Mach [8] is a storage system for fine-grained HFT collected from across the entire system stack. It targets time-critical deployment scenarios where current systems fall short, keeping up with HFT generating millions of events per second and responding to queries about newly arrived data at interactive speeds. Mach empowers engineers to freely add arbitrary HFT sources, read HFT in near real-time when firefighting, and write custom analyses on top of Mach's API primitives. Using Mach makes it simple to temporally correlate HFT across layers of the stack, accelerating the human-in-the-loop, iterative hypothesis-testing process.

This demo shows Mach in action through a realistic firefighting scenario. In our scenario, an engineer needs to urgently understand the root cause of an intermittent performance regression in a key-value store application caused by subtle resource contention with a CPU-intensive machine learning (ML) workload. To drill down to the root cause, the engineer collects and queries HFT from the application and the kernel. As Mach is designed to ingest and query HFT under limited resources in deployment environments, the engineer can interact with HFT in near real-time, guiding them to the root cause of a problem without having to rely on data aggregation or sampling that may mask the underlying issue.

2 RELATED WORK

Time series databases (TSDBs) like InfluxDB [2] target general time-oriented data processing tasks and usually assume a separate deployment with dedicated resources, rather than operating on-host within a limited resource footprint.

Ingest-optimized systems like FasterLog [1] focus on the actual ingestion and storage of data, assuming that analytical queries occur offline. To support fast ingest, they avoid organizing the ingested data for fast querying. Mach, on the other hand, enables real-time querying via a lightweight indexing scheme.

Several near real-time systems support specific, specialized use cases. IoTDB is designed for ingesting numeric sensor measurements from many sources [10]. FishStore scales out indexing predicates on JSON data via multi-threading [4]. Its design is ill-suited for low-resource settings like on-host telemetry processing. In the HFT use case, predicate indexing is an unnecessary cost: engineers add exactly the HFT they wish to observe. Other systems leverage telemetry to address database monitoring use cases: Sentinel [6] extracts information from logging libraries to help users reason about system behavior, without storing these logs for further drill-downs; and Dendrite [7] extracts metrics from logs, `libc`, and the OS to help define heuristics for self-driving database systems.

Unlike these systems, Mach focuses on ingesting large volumes of ad hoc HFT to be queried quickly and with high resource-efficiency.

3 MACH

Deployments of Mach link it as a library within a *monitoring daemon* (Figure 1). The monitoring daemon is a telemetry endpoint like OpenTelemetry Collector [3] that receives data from user-space

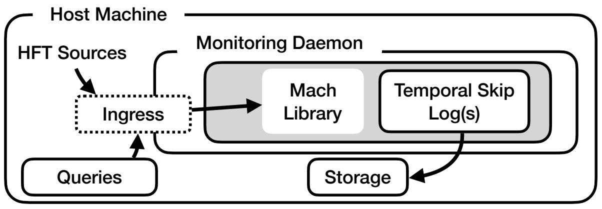

Figure 1: Mach is a library within a monitoring daemon running on a host. HFT sources send data to the monitoring daemon, which invokes Mach to store data. Querying clients use Mach's API to request data for a time range.

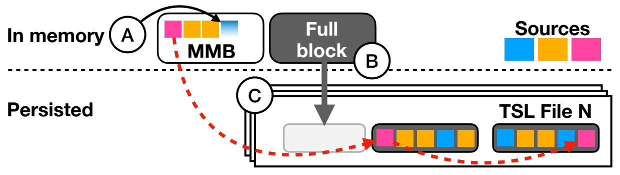

Figure 2: At Ⓐ, a TSL writes a record into a main memory block (MMB), which contains records from multiple sources. A storage thread persists a full MMB into a fixed-sized TSL file (Ⓑ). Each record contains an address of its source's prior record, so queries can skip records from other sources (Ⓒ).

applications instrumented by developers, kernel probes (e.g., a process collecting eBPF events), or hardware events (e.g., via `perf`). The daemon calls Mach's API to write HFT records into Mach. Mach writes these records into the *Temporal Skip Log* (TSL; §3.2), which organizes and persists data locally on the host or remotely in off-host storage (e.g., HDFS, Kafka). A user then *queries* Mach for records in a time range across multiple sources using Mach's `align` query primitive.

3.1 Mach's API

Mach exposes two key API calls: `push` adds a new record, and `align` queries one or more sources by time range.

`Mach::`**`push`**`(source, timestamp, bytes)` pushes a record from `source` into Mach. `timestamp` indicates the record's timestamp and `bytes` correspond to the data pushed. Arrival timestamps must be monotonically increasing for append-only writes. The monitoring daemon adds sources dynamically: it may choose to push application request latency first, and start pushing system call events later. To create a new source, the caller pushes the source's first record.

`Mach::`**`align`**`([sources], time_range)` returns an iterator over records from the set of `sources` within the `time_range`. It aligns the records by timestamp, iterating over them in descending order. The telemetry collector uses this iterator to perform custom correlations and analyses. `align` differs from temporal joins [5] in TSDBs, which match tuples in the same *temporal interval* from different temporal relations. In contrast, Mach's `align` is more akin to taking the union of two temporal relations and sorting the result by *timestamp*, interleaving data from different sources into a single timeline. Mach's `align`—as opposed to a join operator—makes it possible to optimize Mach's internals for HFT, although temporal joins can still be implemented atop the `align` primitive.

3.2 Temporal Skip Log (TSL)

The TSL is an append-only log that organizes time series from multiple sources (Figure 2). Records added to a source receive unique *addresses* in the TSL and link to older records via their addresses, creating a reverse chronological view.

The TSL is arranged as a sequence of fixed-size *blocks*. It holds two blocks of recent data in memory (MMBs) while the remaining blocks reside sequentially in *index files* in persistent storage. The address of a newly inserted record is its position in the current block, plus the combined size of the previous blocks. The TSL can therefore efficiently identify the block for any record using only the record's address. Mach maintains one or more TSLs and manages their addresses internally. The client application interacts with the data in the TSLs using Mach's `align` and `push` APIs.

Efficient Writes. To operate within resource limits, the TSL needs to maintain a fixed memory footprint and perform constant work for each record pushed. To achieve this, the TSL uses fixed-size MMBs and amortizes the cost of flushes to persistent storage.

On construction, the TSL allocates two empty MMBs, selects one as the current MMB, and starts a background thread that flushes full MMBs to persistent storage. When a client pushes a record, Mach appends it to the TSL's current MMB after a 32-byte header (Ⓐ in Figure 2). This operation is efficient, since it simply copies the record into the MMB.

If there is insufficient room in the current MMB, the TSL passes the current MMB to the background flushing thread and writes to the second MMB. The background thread flushes full MMBs in parallel (Ⓑ in Figure 2).

For CPU-bound workloads, the background thread will finish flushing a MMB before the ingest thread fills the other MMB, and ingestion proceeds normally. If the workload is I/O-bound, the ingest thread blocks waiting for one of the two MMBs to be persisted. In other words, the TSL applies backpressure when I/O falls behind, which avoids Mach wasting cycles on retries. This design also simplifies coordination between the pushing and flushing threads, requiring only a simple update of a shared pointer.

Efficient Reads. Internally, Mach maintains the address of the latest record for each source. It uses the TSL to iterate over a source's records in reverse insertion order when reading. This provides two key advantages. First, it prioritizes access to the most recently inserted records, enabling low-latency querying. Second, it allows Mach to skip records more flexibly and at a finer granularity than with data partitioning or zone maps.

To iterate over records for a source, Mach returns the latest record and then iteratively reads the previous record's address to traverse the TSL (illustrated in Ⓒ in Figure 2). The client library stops iteration using a higher-order criterion (like a minimum timestamp) or when the source has no more records.

Mach interleaves records from multiple sources in the same TSL block. Knowing the previous address allows the iterator to skip irrelevant records and blocks, which speeds up queries and reduces CPU footprint (inspiring the Temporal *Skip* Log name). This design yields benefits in Mach's target workloads that mix data from high-rate and low-rate sources [9].

For any record address a, the TSL must support reading from both persisted and in-memory blocks. Because index file sizes are fixed (size M), the TSL determines whether a resides in memory or in an index file by comparing a/M with the total index file count. If the record is in a file, it can be read at offset $a \bmod M$. Otherwise, `read` attempts to read this record in a MMB.

Read-Write Coordination. Reading from MMBs requires careful coordination because it may race with MMB reuse that occurs when the TSL flushes MMBs and returns them to the ingest thread. To safely read a record, the TSL performs a lightweight MMB snapshot. The TSL maintains C, a version counter incremented during reuse, and o, the last inserted record's offset in the head MMB. The query thread reads the version counter C's value, then makes a copy of the MMB's contents up until offset o plus the size of the record. The reader reads C again after copying the data and compares the new value with the value loaded at the beginning of the `read`. If the values differ, a concurrent MMB reuse occurred. MMB reuse implies the record in question was persisted, so the query proceeds to read from persistent storage.

4 DEMO DESCRIPTION

The demo scenario mimics a realistic deployment running a latency-critical key-value store application and a compute-intensive machine learning (ML) workload on the same machine. An engineer receives a notification that some clients are experiencing intermittent drops in query throughput measured in queries per second (QPS). The demo shows that quickly ingesting and querying HFT is critical in diagnosing these complex scenarios and how Mach plays a key role in the firefighting process. Since existing telemetry collection tools and TSDBs either mask, hide, or fail to capture critical information for debugging, the audience sees how an engineer using Mach can effectively diagnose these problems in real-time.

Audience members participate as if they are the engineer in the firefighting scenario, viewing the collected HFT data via a web-based UI served from a Rust backend that queries: *(i)* InfluxDB as a representative baseline; and *(ii)* Mach. Since summary statistics are ubiquitous and most systems easily support them, engineers will often look first at a QPS summary visualization similar to ① in Figure 3. The blue line indicates QPS over the past 30 seconds. We annotated Figure 3 with points of interest (black arrows) that would prompt further investigation from an engineer. In the aggregate case ①, however, it is quite difficult to tell these dips apart from regular fluctuations in system performance. Therefore, the engineer needs more fine-grained HFT data to diagnose the problem.

The demo then walks through the engineer's steps to identifying the root cause of these periodic throughput drops. The next step involves collecting additional per-query application events (e.g., latency, query runtime) and aggregated summary statistics, which the audience can enable using the controls in ②. This data is high-volume (about 1M events per second) and must be collected continuously, since the engineer does not know the next time that the issue will occur (e.g., within a few seconds, a few minutes, or a few hours). Collecting HFT and scanning the visualizations for patterns allows the engineer to iteratively formulate a hypothesis about the source of the intermittent decrease in QPS and validate this hypothesis by aggregating the HFT in different ways. In our scenario, the engineer eventually determines that the issue is localized to two CPU cores (green and red lines, other cores in gray).

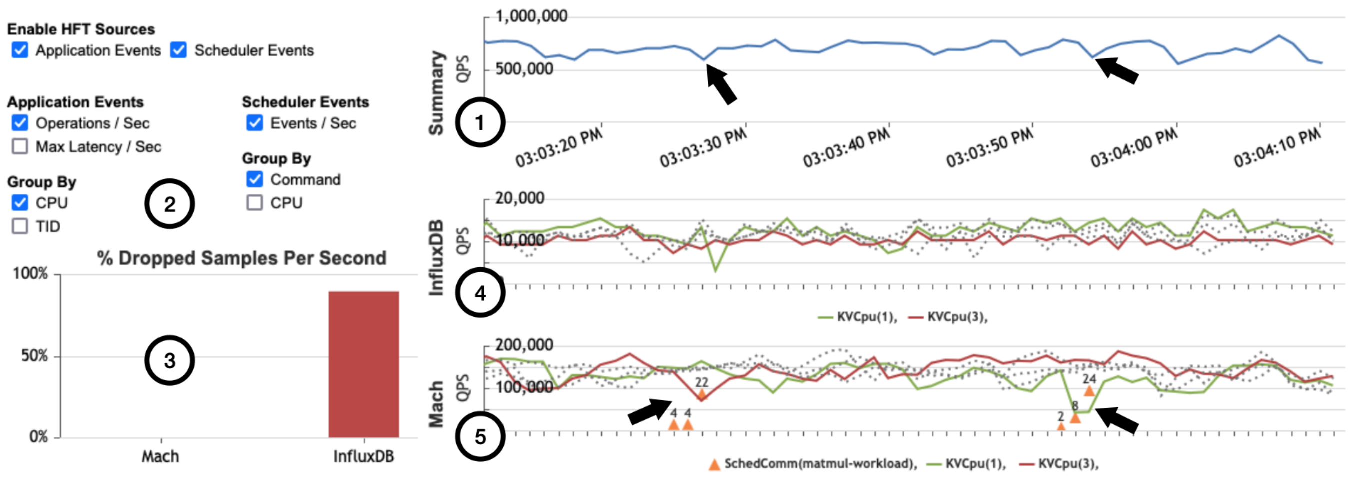

Figure 3: In ①, summary statistics fail to clearly show the performance regression. An engineer can add HFT sources via ②. When enabled, ③ shows how much data is dropped: InfluxDB drops 90% of the data while Mach keeps up. As a result, InfluxDB's data does not have enough detail to characterize the performance regressions (④). Using Mach (⑤), HFT demonstrates the issue: it occurs in two CPU cores (green and red lines) due to the same interfering application (orange triangles). InfluxDB misses these rare events as fewer than 20 events in about 1M records cause the performance regressions.

The audience will also see how InfluxDB cannot keep up with HFT data, potentially leading to erroneous conclusions. The drop percentage visualization (③ in Figure 3) shows that InfluxDB drops nearly all of the HFT data it receives, while Mach drops none. Hence, InfluxDB's remaining data points are difficult to discern from noise ④. At best, the engineer is left with no clear understanding; at worst, it might lead to the assumption that the issue is "just noise" in the system. The complete picture that Mach provides clearly shows separate drops in QPS in individual CPUs ⑤, annotated with black arrows that correspond to the dips from the original summary visualization ①. This fine-grained breakdown from Mach highlights two key points: (1) the initial aggregated visualization partially masked the issue because it counted QPS over all CPU cores; and (2) the clients only experience the issue when the OS schedules the ML workload on the same core as the key-value store.

With this more detailed view, the engineer can correctly deduce that a CPU-intensive task is periodically scheduled on the same core, and audience members can again use the controls ② to add scheduler events from in-kernel instrumentation to verify this hypothesis. Since systems like InfluxDB cannot keep up with the high volume of HFT, engineers typically need to use another suite of tools (e.g., `perf`, BPF, `tcpdump`) for collecting kernel-level data, but splitting HFT collection across many different tools makes it difficult to correlate data from disparate sources to diagnose issues in real-time. Specifically, the engineer in our scenario would like to correlate scheduler events (about 150K events per second) with the previously observed CPU behavior. Mach facilitates this type of analysis by keeping up with *both* data sources and allowing the engineer to query them from within a single system.

The Mach visualization ⑤ in Figure 3 confirms the engineer's hypothesis: scheduler events for a process named `matmul-workload` correspond to drops in QPS on specific CPU cores. Out of all the collected HFT data (about 1.1M events), fewer than 20 such events occurred during the relevant time period. These "needle-in-the-haystack" issues are nearly impossible to identify with traditional tools because rare events are lost by storage engines that rely on sampling, pre-aggregation, or dropping data. As shown in ④, InfluxDB fails to help the engineer diagnose the root cause because it misses *all* of the rare events.

The engineer can now take actionable steps to address the issue. In the short term, they can pin the key-value store application to other CPU cores to avoid this contention. Longer term, they can reach out to the team managing the interfering process to coordinate better resource utilization of the shared machine.

ACKNOWLEDGMENTS

This work was supported by a gift from Intel and by a Microsoft Grant for Customer Experience Innovation.

REFERENCES

[1] [n.d.]. *FasterLog*. https://microsoft.github.io/FASTER/docs/fasterlog-basics/
[2] [n.d.]. *InfluxDB*. https://www.influxdata.com/
[3] [n.d.]. *OpenTelemetry Collector*. https://opentelemetry.io/docs/collector/
[4] Badrish Chandramouli, Dong Xie, Yinan Li, and Donald Kossmann. 2019. FishStore: Fast Ingestion and Indexing of Raw Data. *PVLDB* 12, 12 (2019), 1922–1925.
[5] Dengfeng Gao, Christian S Jensen, Richard T Snodgrass, and Michael D Soo. 2005. Join Operations in Temporal Databases. *VLDB J.* 14 (2005), 2–29.
[6] Brad Glasbergen, Michael Abebe, Khuzaima Daudjee, Daniel Vogel, and Jian Zhao. 2020. Sentinel: Understanding Data Systems. In *SIGMOD*. 2729–2732.
[7] Brad Glasbergen, Fangyu Wu, and Khuzaima Daudjee. 2021. Dendrite: Bolt-on Adaptivity for Data Systems. In *SIGMOD*. 2726–2730.
[8] Franco Solleza, Andrew Crotty, Suman Karumuri, Nesime Tatbul, and Stan Zdonik. 2022. Mach: A Pluggable Metrics Storage Engine for the Age of Observability. In *CIDR*.
[9] Goutham V. 2017. *How and Why Prometheus' New Storage Engine Pushes the Limits of Time Series Databases.* https://youtu.be/C4YV-9CrawA?si=S6poj7qBUu2LS3UR Talk at DockerCon EU 2017.
[10] Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin Mcgrail, Peng Wang, Diaohan Luo, Jun Yuan, Jianmin Wang, and Jiaguang Sun. 2020. Apache IoTDB: Time-series Database for Internet of Things. *PVLDB* 13, 12 (2020), 2901–2904.