The Time Travel Index: Supporting Efficient Historical Queries over the Mach Storage Engine

William Sun
Brown University

Note: this research is done in collaboration with Franco Solleza, Richard Tang, and Malte Schwarzkopf, and while this paper focuses on my own contributions, it relates to and discusses work done by these others.

1 Introduction

When managing large-scale deployments, developers often have to move quickly to handle various bugs and outages that may arise. In such scenarios, developers rely heavily on a variety of data to diagnose and resolve the problem, using this data to glean information about the state of the system and reacting appropriately. Tools such as OpenTelemetry [3] and Prometheus [4] are designed to provide a way to access application-level telemetry data, but engineers often additionally need access to high-fidelity telemetry (HFT), produced by tools such as eBPF or perf. However, even for a single application, HFT sources can generate millions of samples, and today’s state-of-the-art storage systems are not equipped to ingest this amount of data and serve queries for it in real time, potentially leaving engineers with insufficient data to identify the source of the phenomenon that they are investigating.

Mach [6] is a storage engine that addresses these challenges and is designed to ingest HFT and serve queries in real time. Mach is able to keep up with HFT both on the ingest and the query side, in large part due to its underlying data structure, the Temporal Skip Index (TSI). However, to keep Mach and the TSI lightweight enough to support HFT, they only natively support linear traversal through the stored data, with the starting point being the most recent entry stored in the TSI. This means that with Mach, a developer cannot efficiently perform historical queries – queries for data some distance in time away from the most recent sample.

To demonstrate this, we look at the RocksDB workload described in the Mach paper. In this scenario, an engineer is investigating slowdowns in a RocksDB-based application, first looking at routinely collected telemetry in Phase A, then investigating system call latencies in Phase B, and finally looking at get requests, pread64 call, and page-cache evictions in Phase C. In the original paper, Mach queries the last 10 seconds of data starting from the present. Engineers may additionally want to compare the system calls from the last 10 seconds to system calls that occurred before the slowdowns happened. In that case, they would want to query specifically for 10 seconds of data starting, for example, 1 minute ago - without needing the data more recent than that.

However, there is no way to directly access any sample other than the most recent one in Mach, and all other samples must be reached via iteration from the most recent sample. As such, to reach data from 1 minute ago, Mach must iterate through all the data from the last minute as well. As shown in Figure 1, this means that even when querying the same amount of data, historical queries can be significantly more expensive to perform.

As an alternative to Mach, there do exist storage systems that provide support for efficient historical queries, such as storage systems that implement B-trees or similar structures that provide logarithmic time access to arbitrary accesses to stored data. However, the tradeoff is that these all require heavyweight indexes that sacrifice ingest speed – making them ill-suited for handling HFT data.

In this paper, we propose the Time Travel Index (TTI), a
lightweight index over Mach. It preserves Mach’s efficient ingest and linear query properties while also providing support for efficient historical queries with low memory and latency overhead. We demonstrate that the TTI is still able to keep up with HFT workloads while serving efficient historical queries. We also demonstrate that this index provides benefits over Mach, which is unable to efficiently serve historical queries.

The TTI achieves efficient time travel querying by splitting each user-provided source into subsources, which split the original data into small continuous subsegments. The TTI feeds each subsoure into Mach as a separate source, and connects adjacent subsources by a new linking API added to Mach. Doing this allows the TTI to directly access subsources corresponding to data sometime in the past without needing to traverse the data in all the subsources more recent than it. The linking API additionally provides functionality to allow for a broad range of other expressive indices to be created over Mach.

The TTI also maintains low overhead by only keeping track of subsources in the recent past, discarding metadata about older subsources as new subsources are created. In addition, we add new operations to Mach’s API so that Mach can discard older subsources from its metadata, making these subsources only accessible via traversal across linked subsources but allowing Mach to incur low overhead from the creation of new sources. This allows for the TTI to have a fixed and adjustable memory overhead when compared to Mach, while still allowing older data to be accessible.

This paper makes the following contributions:

1. The Time Travel Index (TTI), a lightweight index over Mach that provides fast historical queries on recent data with low overhead.
2. Additions to the Mach API that allow for the TTI to perform well, as well as enabling more expressive indices to be added to Mach in the future.

2 Background

High-fidelity telemetry (HFT) consists of large amounts of non-sampled raw data collected from frequently occurring events such as system calls. For example, in the RocksDB workload, in Phase B the engineer collects data from all system calls, and then in Phase C narrows this down to get requests, pread64 call, and page-cache evictions. HFT is often necessary to diagnose the state of a system — referring back to the RocksDB workload, the engineer in that scenario seeks to align the get requests, pread64 call, and page-cache evictions in Phase C to investigate how they correlate to each other, meaning the raw data is required. Tools that generate HFT include eBPF and perf, and while these tools can collect and provide HFT in real time, they maintain high performance by not storing the data and sometimes providing aggregates instead of raw data.

Figure 2: Mach contains address maps that store the most recent address for every source, and shares these addresses with readers via a shared address map. Mach allows for separation of data into partitions via internally creating multiple TSI, and these allow for localization of data based on a-priori knowledge of queries. The Mach Reader executes snapshots on the relevant partitions through shared references to the TSI’s (TSI-R).

```cpp
Mach::push(source, partition, timestamp, bytes)
Mach::align([(source, partition)], time_range)
```

Figure 3: Mach’s API: push writes samples into Mach, while align reads one or more sources within a given time range.

2.1 Mach Overview

Mach [6] is a storage system designed to ingest HFT without dropping data, and simultaneously serving queries in real time. Figure 2 shows the overall architecture of Mach. As shown in Figure 3, Mach’s API exposes two operations: push, which adds new data into Mach, and align, which performs a time-range query over multiple sources.

```cpp
Mach::push(source, partition, timestamp, bytes) performs a push into Mach. Mach retrieves from its address maps the most recent address of an element in that source-partition (or creates a new entry in the address map if this is the first element of that source-partition). It then passes this into the TSI corresponding to the given partition, so that the TSI can link this new data item with the last one of this source. Then, Mach updates the address maps with the address of this new item. Mach contains a fixed number of partitions, which each correspond to a TSI internally, and the caller might choose to separate their data into separate partitions if they expect a new source to generate a lot of data or to speed up queries over existing data.

Mach::align([(source, partition)], time_range) returns an iterator over the source-partitions provided by the call to the API. It creates a snapshot of the data at the time that Mach receives the query, and aligns each relevant record such that iteration occurs over timestamps in
Mach supports HFT workloads via its Temporal Skip Index (TSI). The TSI is an append-only data structure that stores data in fixed-size blocks. Each piece of data is identified via a unique address and stores a reference to the address of an existing item (in Mach’s case, the previously most recent item in the same source). The TSI stores blocks containing recent data in memory, moving older ones to persistent storage in index files which each contain a fixed number of data blocks. This structure allows for even data entries in persistent storage to be efficiently accessed, as the address of the entry can be easily translated into an offset into an index file. However, this linearly linked structure of the TSI is also what causes Mach to perform poorly in a historical query scenario – as Mach chooses to store only the address of only the most recent element in a source-partition, in order to access data back in time, the TSI must linearly traverse the data in a source via the links until it reaches the first relevant element.

### 2.2 FasterLog

As an alternative to Mach, FasterLog [1] is a log-based storage system built for high ingest rate workloads. However, due to the log storage structure, similar to Mach, data needs to be iterated through linearly from the end of the log. This means it is similarly difficult to access older data, and this issue is compounded by the fact that FasterLog does not natively support features such as sources or partitions, nor does it natively store timestamps for each record. Thus, to get FasterLog to provide similar functionality to Mach, additional serialization work needs to occur from the query worker before passing it into the storage system, and similar deserialization needs to occur on the read worker’s end. These challenges are also present in many other log-based storage systems, making them hard to use in a scenario that requires fast historical queries.

### 2.3 Other Approaches

Structures such as B-trees or LSM trees can provide more efficient time travel queries. These structures use heavyweight tree-based indices to accelerate read performance and are used by time-series databases such as RocksDB [5] and LMDB [2]. However, the heavyweight nature of these indices means that these structures do not support high-rate ingest, which means that while they would provide efficient time travel queries, they are ill-suited for an HFT workload.

### 3 The Time Travel Index (TTI)

The TTI builds off of Mach, maintaining Mach’s ability to keep up with HFT while additionally providing functionality for efficient access to historical queries of recent data. We design the TTI to provide fast historical queries for recent data, while preserving Mach’s high ingest rate and not incurring significant extra resource overhead. To achieve this, we modify the Mach API to provide 2 new operations: a linking operation, which links two sources together, and a close operation, which removes a source from Mach’s metadata, making it only accessible via traversal of links. Using these modified APIs, the TTI splits user-provided sources into subsegments known as subsources, which each contain a short time range’s worth of data and are linked to subsources containing the adjacent time ranges’ data. This allows the TTI to perform fast historical queries by beginning iteration not at the most recent record in a source-partition but rather the subsource corresponding to the time range the first relevant record is in. To minimize resource overhead, the TTI only keeps track of recent subsources for each source, enabling for fast historical queries of recent data while preventing the memory required to maintain the TTI from growing indefinitely.

#### 3.1 Mach API Changes

To support the TTI, we add the following operations to the API of Mach: a link operation, which links a source to the most recent element of another source, and a close operation, which removes a source from Mach’s address map.

**Mach::link(source, partition, timestamp, (linked_source, linked_partition))** creates a link between two existing sources, pushing the address of the linked source into the linking source. It does this by looking up the linked source in Mach’s address map, and then pushing into the relevant TSI an entry that contains a flag indicating that this entry is a link, as well as the address corresponding to the most recent item in the linked source. This means that a link can only be created with a source that is still in Mach’s address map, i.e. that hasn’t been closed.

**Mach::close** removes a source from the address map that translates between source IDs and the equivalent address in the TSI. This does not remove data from the TSI, so any data stored in that source still exists in the TSI – this means that if a still-accessible source linked to a closed source, that source’s data could still be accessed. However, that source is no longer accessible directly via Mach’s align API.

In addition, we modified Mach’s align API functionality to allow for the traversal of links. The iterator returned by Mach’s align API no longer returns just a data point; if the iterator encounters a link, the iterator will instead return a new snapshot which represents the snapshot if the linking source is replaced with the linked source (i.e., traversal down the link) and all other sources remain the same. If a user of Mach wishes not to traverse down the link, they can simply iterate to the next item using the original iterator, and iteration can proceed as normal.

#### 3.2 TTI Architecture

The TTI leverages these new Mach API operations to provide efficient traversal back in time. The structure of the TTI is
Figure 4: The TTI’s architecture builds off of Mach. It mirrors Mach’s structure, exposing a writer, which handles ingesting HFT data, and a reader, which serves queries. The TTI also stores a map which tracks the subsources and their corresponding true source and time range.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME_PER_SUBSOURCE</td>
<td>Defines the length of data a subsource contains.</td>
</tr>
<tr>
<td>MAXIMUM_SUBSOURCES</td>
<td>Defines how many subsources the TTI holds in metadata.</td>
</tr>
</tbody>
</table>

Figure 5: The TTI contains parameters to configure how much data a subsource contains and how many subsources are stored in memory. These are configurable for each source. Shown in Figure 4. The structure of the TTI is similar to Mach, exposing a writer, which receives data and pushes it into Mach, and a reader, which receives queries for stored data. The TTI stores an additional piece of metadata, the shared subsources map, which is a map that maps subsources to their corresponding true source and time range. This is accessed by both the writer, so it knows what subsource to write into, and the reader, so it knows what subsource to start the returned iterator at.

3.2.1 Configurable Parameters

Figure 5 describes the configurable parameters of the TTI. The TTI allows you to configure SECONDS_PER_SUBSOURCE, which defines how long of an interval of data a subsource contains, and MAXIMUM_SUBSOURCES, which defines how many of the most recent subsources the TTI contains in its metadata. With these, users of the TTI can tune the tradeoff between memory overhead and efficiency of historical queries. If subsources are made to be longer, then historical queries are expected to take longer, as a user is expected to need to iterate longer within a subsource until they reach relevant data. However, memory is saved by making the amount of metadata for a particular time range smaller. Similarly, more subsources in metadata means a longer period for which the TTI can directly access the subsource with the first relevant record in it, but more memory is needed to store more subsources. We refer to the time range defined by the time range per subsource multiplied by the maximum number of subsources as the "recent past", and data within the recent past as "recent data".

3.2.2 Subsources

Figure 6 shows the structure of subsources as they are stored in the TTI. The TTI creates subsources for each logical source the user provides to the TTI, which split the original data into smaller continuous chunks of data that corresponds to a time range. The parameter SECONDS_PER_SUBSOURCE defines the length of data stored in each subsource, and is set by default to one second’s worth of data. Thus, as an example, a single logical source would contain a subsource for the first second’s worth of its data, another subsource for the next second’s worth of its data, and so on. The TTI creates a new subsource at each of these time intervals, and links each subsource to the subsource containing the adjacent prior time range of data using Mach::link, which means that once the end of one subsource is reached, Mach can directly iterate to the next subsource.

With this, the TTI can create Mach iterators that start at the most recent record of each subsource, rather than only being able to create an iterator starting from the most recent record of the whole source. This means that for historical queries, an iterator can be created for the subsource that contains the first relevant element, eliminating the need for a user to traverse the entire source until it reaches the first relevant element, which is true of native Mach.
3.2.3 TTI API

The TTI’s API is shown in Figure 7. It exposes the same API as Mach, but with functionality modifications to make it more suited for historical queries:

\[
\text{TTI::push(source, partition, timestamp, bytes)}
\]

\[
\text{TTI::align([\{source, partition\}], time_range, time_travel_len)}
\]

Figure 7: TTI’s API: push writes samples into Mach, while align reads one or more sources within a given time range and given a certain amount of lookback.

3.2.4 Low Resource Overhead Tradeoffs

To maintain the efficiency of the TTI::push operation, the TTI limits the operations it needs to perform on an individual push or read, instead updating all metadata upon reaching the point at which a new subsource needs to be created. When the most recent subsource has not yet reached its maximum length, the TTI simply pushes the data into this subsource; only when this most recent subsource reaches its maximum length does the TTI create a new subsource and links it to the previously most recent subsource. Similarly, the iteration API is a light wrapper over Mach’s align iterator API, simply returning the value the Mach iterator would have returned and traversing all links if any link is present. This allows the TTI to preserve Mach’s high ingest rate and ability to query while ingesting HFT.

In addition, the TTI continuously creates new subsources as it receives data, and since the TTI keeps track of subsources and their corresponding true source and time range via the subsources map, the TTI’s map could grow arbitrarily large. To combat this and limit memory overhead, we choose to limit the number of subsources the TTI keeps in its subsources map. This does not limit accessibility to all of the data, as all of the subsources are still linked together and thus all data is accessible via traversal. However, any subsource not present in the TTI subsource map cannot have an iterator start at that subsource, which means that the range of queries that are sped up the most by the TTI is limited only to recent data. For any lookback lengths longer than the total amount of time the subsources we have stored make up, the TTI begins its iteration at the oldest saved subsource, and iterates backwards until it reaches the relevant data; this is not as efficient as historical queries for recent data, but this iteration skips through all of our still saved subsources, making it still more efficient than native Mach.

However, since what the TTI considers a "subsource" corresponds to a regular source in Mach, each subsource requires Mach to add an entry to its address map. This means that with no modification, the TTI’s setup would also result in Mach’s address map growing arbitrarily large even with just one source generating data into the TTI. To address this, the TTI leverages the new Mach::close API. We can observe that once a subsource has collected all of its data the TTI
will never add more data to it, and the TTI cannot create an iterator on a subsoure that is evicted from the TTI subsources map. This means that for any subsoure not in the TTI map, Mach will never need to look up what the address of the most recent entry of that subsoure is, since the TTI cannot call either Mach::push or Mach::align on it again. Thus, the TTI also evicts that subsoure from Mach’s address map to save memory. This fixes the memory overhead of Mach’s memory map.

4 Evaluation

In our evaluation, we seek to answer the following questions about the TTI:
1. Does the TTI keep up with ingest in an HFT setting? (Section 4.1)
2. How does the TTI impact the ingest performance of Mach? (Sections 4.1, 4.2)
3. Does the TTI provide a speed-up to historical queries compared to native Mach? (Section 4.3)

To this end, we evaluate the TTI under the RocksDB workload, the motivating example in the original Mach paper and a scenario that derives from real-world observability use cases, as well as testing it under a synthetic benchmark. We then compare these results to a baseline of the original Mach implementation. We evaluate all of the systems under the same parameters as the original Mach paper.

4.1 End-To-End Performance

First, we measure the ingest throughput achieved for the TTI and Mach on the original RocksDB workload as proposed in the Mach paper as an end-to-end evaluation of the TTI. We set the parameters of Mach to be identical to the RocksDB evaluation in the original Mach paper. For the TTI, we configure it have subsources of length 2 seconds with the maximum number of subsources being 32. This makes the range of "recent data" reasonably large while also being small enough that each phase of the workload is longer than the "recent data" range and subsources begin to be evicted.

Figure 9: The TTI manages to keep up with the RocksDB workload with identical throughput to Mach in all phases and without dropping items.

Figure 9 shows the results. Matching the results of the original Mach paper, these results show Mach manages to keep up with the workload throughout every phase. Crucially, however, the TTI produces identical throughputs to Mach across all phases of this workload and drops no data in any phase. These results indicate that the TTI is still lightweight enough to handle ingest of HFT workloads.

4.2 Ingest Performance

Next, we measure the maximum ingest throughput achieved for the TTI and Mach on a synthetic ingest benchmark. The benchmark used mirrors the Data Structure Ingest Scaling benchmark used in the original Mach paper. This workload writes samples into both Mach and the TTI in a closed loop, and measures the ingest throughput of each system in MiB and samples per second. We additionally vary the sample size from 8 bytes to 1KiB to test how each system’s performance compares under varied sample sizes. The parameters for both systems remain the same as in the RocksDB end-to-end evaluation.

Figure 10 shows the results. Matching the results in the Mach paper, I/O throughput (in MiB/second) increases with sample size for both Mach and the TTI, while the throughput in samples/second decreases with sample size. Comparing the performance of Mach and the TTI, we see that at smaller sample sizes, the TTI has approximately 0.8x the ingest rate of Mach. This is due to the per-sample overhead of needing to look up the source in the TTI subsource map to find the most recent subsoure to push the sample into. However, as the sample size increases, the TTI’s performance eventually reaches Mach’s performance, as at a sample size of 64 bytes the TTI achieves roughly 0.9x Mach’s performance and at 256 bytes and 1024 bytes the TTI achieves identical performance to Mach. This demonstrates that the TTI is lightweight and
does not heavily impact the ability of Mach to ingest HFT.

4.3 Query Performance

Finally, we evaluate the query performance for the TTI and Mach on a modified version of the RocksDB workload where we perform historical queries instead of queries from the most recent sample point. We specifically focus on Phase B of the RocksDB workload, as this phase contains the most intensive queries and is the phase where the queries take the longest for Mach in the original Mach paper.

We set the query length of all queries to be 10 seconds, and vary the lookback time from 0 seconds to 20 seconds. The parameters for both systems in this experiment are identical to the previous experiments, except that we modify the TTI’s maximum number of subsources to be 5. This makes the "recent data" time range to be 10 seconds, as for any lookback time that lands in the first 5 subsources (which contain 10 seconds worth of data) we can jump directly to the subsource that contains the first relevant sample is in, whereas all lookback times larger than that must begin their iteration from the oldest saved subsource and traverse linearly through the data until it reaches the first relevant sample. This experimental setup allows us to see how the TTI and Mach perform on both historical queries on recent data and historical queries on older data.

Figure 11 shows the results of this evaluation. As expected from Mach, the amount of time it takes to query the data increases linearly as the lookback time increases. This is because in Mach, we must start at the most recent data point for the relevant source-partition, and iterate from that data point through every sample in that source-partition until it has iterated past the lookback time period and reaches the first sample that is in the query range.

In contrast, the TTI maintains constant-time queries with respect to the lookback time within the range of recent data (in this example, 10 seconds), which provides a marked query latency improvement from Mach, at maximum reaching 1.9x improvement in this experiment. Increasing the maximum number of subsources would increase this improvement factor even further, as the length of lookback for which the TTI provides a constant time lookback would increase.

For lookback times that exceed this 10 second "recent data" period, we see that the TTI’s query latency begins to scale linearly with respect to the lookback time. This is expected, as the TTI can no longer directly jump to the subsource that contains the first relevant element, since that subsource is no longer in the TTI’s subsource map. However, since we can still skip to the oldest subsource and begin iteration from there, we see a constant query latency improvement of around 9 seconds for lookback times longer than 10 seconds, showing that the TTI still provides some speed up for historical queries of older data when compared to an unmodified Mach system.

5 Discussion and Future Work

We show in this paper that the Time Travel Index (TTI) provides efficient historic queries on recent data without sacrificing the ability to ingest and serve queries on HFT in real time. We see that the TTI, like Mach, is able to keep up with intense HFT workloads where other baseline storage systems can not, and that the TTI provides significantly improved query times in a historic query scenario compared to Mach and other baselines.

However, there remain improvements that can be made to the TTI. For example, it may be necessary for engineers to look back a more significant distance in the past, and currently the TTI provides very little query latency improvement for these queries. To combat this, we could modify the TTI to keep some older subsources, decaying the number of older subsources the TTI keeps by an exponential factor so that the range of time the TTI can provide a noticeable query latency improvement for a significant amount without incurring significant memory overhead.

In addition, as mentioned in Section 1, the newly provided APIs of Mach::link and Mach::close can be used to build other indices, and building such indices is a potential area of future work. For example, if an engineer needs to access the system calls that occur during an application-level event, we can link the system calls and application-level event sources together such that to access the relevant system call data, we only need to traverse the much sparser application-level event data until we reach the right link. An index that provides this functionality could improve the speed of queries on HFT by linking them to related infrequent events, and is an area worth exploring in the future.
References


