General Baggage Model for End-to-End Tracing and Its Application on Critical Path Analysis

A PROJECT REPORT
BY
HONGKAI SUN
TO
THE DEPARTMENT OF COMPUTER SCIENCE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE
IN THE SUBJECT OF
COMPUTER SCIENCE

BROWN UNIVERSITY
PROVIDENCE, RHODE ISLAND
MAY 2016

Project advisor: Rodrigo Fonseca Hongkai Sun

General Baggage Model for End-to-End Tracing and Its Application on Critical Path Analysis

Abstract

Many monitoring and diagnosis systems have been proposed based on causal tracing for end-to-end executions in distributed systems as more and more modern computer applications become distributed systems. These applications are based on metadata propagation along the request's executions, which has been closely tied to the application logic, the metadata formats, and the developer APIs. Such kind of coupling makes it impossible to reuse the metadata propagation code added by the developers to their systems and duplicates the efforts. Such duplicate efforts have been identified as the main barrier to entry for these tracing applications.

In this report, we propose a general metadata container called the Baggage Model, which enables us to reuse the metadata propagation. Baggage provides necessary flexibility, extensibility, and isolation to allow different tracing applications to share the same instrumentation, which should significantly reduce the duplicate efforts for the developers. Three tracing applications are modified by us to take advantage of the general baggage model. We have also briefly analyzed the performance of our baggage implementation.

Based on the baggage model, we propose a new tracing application called CPath for critical path analysis. This tool measures the overall latency, the critical execution path, and the slack of the request's executions to help developers to figure out the slow-down factors. We also propose the idea of hypothetical speedups to simulate the optimization plans with the real request executions to see how much improvement on the overall latency the plans can work out, which helps the performance engineers to set up feasible and meaningful peformance goals for the development teams.

We also talk about the lessons we have learnt and future work with both baggage model and CPath.

Contents

Acknowledgments

I would like to thank my advisor Rodrigo Fonseca for the awesome advising on this project and the generous help in getting started in this field, and Jonathan Mace for his excellent ideas on this topic and the extensive helpful suggestions on this report. I also want to thank my office-mates as well as all my friends for such great companions during my years at Brown.

1 Introduction

Many modern computer applications are distributed systems, comprising multiple application layers spread across many machines. Examples include the increasingly popular domains of data analytics, microservices, web services, and more. Monitoring and troubleshooting distributed systems is a fundamental challenge faced by operators and developers, because unlike in standalone systems, problems in distributed systems often span multiple machines and application layers. The tools commonly used today to diagnose problems, such as logs, counters, and metrics, cannot coherently reason about end-to-end executions across a distributed system, because important context is lost whenever an execution crosses software components or machine boundaries.

Consequently, a variety of monitoring and diagnosis systems have been proposed to reason about end-to-end executions in distributed systems. For example, X-Trace [7] is an end-to-end tracing system that produces directed acyclic execution graphs. An execution graph records events that occur during an execution, and organizes them by capturing causality between events, even across application layers and machines. X-Trace enables system operators to observe execution paths across an entire system, and has been used for both anomaly detection and diagnosing steady-state problems. Retro [10] is a resource management framework for multi-tenant systems that collects end-to-end resource consumption information. Retro enforces resource management policies even in common layers such as storage that might traditionally lack the necessary context for attributing executions to tenants. Pivot Tracing [11] is a monitoring tool that enables users to correlate statistics generated from potentially multiple points along an end-to-end execution. Prior to Pivot Tracing, it was typically difficult or impossible to correlate statistics across different applications, because applications do not share context with each other to make correlations.

Causal metadata propagation is a fundamental component used by X-Trace, Retro, Pivot Tracing, and several other systems in both research and academia [2, 4, 5, 12]. Causal metadata propagation is a white box instrumentation strategy whereby metadata is carried alongside requests as they execute. For example, Retro propagates an 8-byte *tenant ID* alongside each request as it traverses application layers and machines. Causal metadata propagation requires up-front developer effort to *instrument* systems, by changing their source code to, for example, pass metadata across thread boundaries, or include metadata within remote procedure calls (RPCs).

In theory, each system should only need to be instrumented once, because its execution boundaries will be the same regardless of the tracing application being used. However, all existing tracing applications today tightly couple their application logic, their data formats, and their developer APIs for system instrumentation. As a result, one tracing application today cannot reuse the system instrumentation of another tracing application, despite the fact that the developer efforts and instrumentation points are identical. This presents a significant barrier to adoption due to the additional developer efforts required to deploy each additional tracing application.

In this report, we address this issue with a general model for causal metadata propagation, the baggage model, which decouples the data format and the propagation rules from the end-to-end tracing applications. With the baggage model, system instrumentation is a one-time cost incurred by developers. Tracing applications are deployed as *plugins* to the baggage model, which re-use the same underlying metadata propagation. Baggage enables multiple tracing applications to coexist and share the same underlying metadata propagation and system instrumentation. New tracing applications can be deployed in the system with negligible cost, as no additional developer instrumentation effort is required.

This rest of this report proceeds as follows. In §2 we outline the challenges of instrumenting systems today and demonstrate by an example of how instrumentation is incompatible between different tracing applications. In §3 we detail our proposed general baggage model that addresses these challenges and enables tracing applications to share a common metadata propagation layer. §4 details how we convert X-Trace, Retro, and Pivot Tracing to be baggage plugins, and §5 outlines a new tracing application called CPath, which measures critical path latencies and calculates hypothetical speedups. Finally, in §6 we discuss several practical constraints that we encountered, and challenges for future work.

2 Background & Motivations

In this section, we give an overview of two distributed system debugging tools, X-Trace and Retro, which are built based on end-to-end metadata propagation. We show the similarities between them with an example, where we go through a developer's experience of using the tools.

The graph which shows the activities during an execution is called an *execution graph*. Traditionally, developers use logging to figure out the activities of programs. However, even on the same machine, the log messages from different threads would interleave when they come concurrently, which makes debugging hard, let alone the hardness of matching the logs generated on different machines. In contrast, the execution graph can intuitively show developers and operators about what the system does during the execution. X-Trace obtains the execution graph by propagating metadata to keep the causal relation between the log messages, which are called *events* in X-Trace. Therefore, the concurrent log messages can be distinguished by X-Trace and shown with their different threads. If a developer wants to use X-Trace, some modifications are necessary to be made to the application's source code.

When the system starts serving a request, the developer should call the following function provided by X-Trace to generate an 8-byte *task ID* to uniquely identify the request and store into a thread local variable, and all the events generated by this request will contain this task ID so that X-Trace can collect all the events associated to this request.

```
XTrace.startTask();
```

As most non-trivial systems are not single-threaded, a request's execution is likely to create new threads or do asynchronous operations by making RPC calls or enqueuing callbacks to a thread pool. In these cases, developers must copy over the task ID or include it in the asynchronous requests so that the new threads created have the task ID. For example, to propagate the task ID to another thread, a developer can propagate the task ID by calling

```
XTraceContext.getThreadContext();
```

to get the metadata from the thread local variable, copying or carrying it to the new thread, and loading into the new thread by calling

```
XTraceContext.setThreadContext(...);
```

The developer can log some events for debug purposes, for example

```
XTrace.log("User Logged In");
```

In the end, the developer wants to link the threads together after they join, so events should be recorded from the joining thread by calling

```
report = XTrace.createReport();
report.addParent(otherContext);
```

In another aspect, the systems shared by multiple tenants have the difficulties in performance guarantees and isolation, since the tenants are sharing the resources not only within a process but also across processes and machines. The traditional resource management mechanisms cannot work in this case. Retro is created to audit and control the resource usage along the execution of a request.

Now, if the developer wants to add Retro for multi-tenant resource management, the developer should set the tenant ID to uniquely identify the tenant by calling

```
    Retro.setTenant();
```

Similar to X-Trace, Retro must guarantee that the tenant ID is available throughout the execution of this request so that all the resource consumptions are recorded by the system based on the tenant ID. Likewise, this information should be copied to the new threads or carried with the callbacks and RPC calls. Appropriate function calls by the developer in the system are necessary.

Noticeably, X-Trace and Retro focus on different problems, where X-Trace does not care about Retro's resource attribution, while Retro does not care about X-Trace's log API. However, both systems need to make some piece of data available throughout a request's execution as the request traverses from machines to machines, processes to processes, applications to applications. They both need modifications on the application's source code in order to do such metadata propagation, and such functionality is hard to be implemented correctly, but their propagation parts are highly tied to the particular systems and cannot be reused for different systems. Namely, if a developer wants to use both X-Trace and Retro, this person would have to duplicate the effort required by adding metadata propagation.

Ideally, we would like to remove this redundancy and avoid the duplicate work by reusing the metadata propagation, since the real work this component does is to propagate information along with a request's execution. However, the metadata used in the current systems is not designed for reuse by other tracing applications. The metadata has inflexibly fixed formats, and some of the formats are fixed sized, which means new systems cannot extend the metadata. As a result, each new system requires new instrumentation with its own metadata format. In the previous work [7, 10, 11], such duplicate efforts have been identified as the main barrier to entry for these tracing applications, and a general extensible reusable design for metadata propagation is necessary for tracing applications.

3
General Baggage Model

3.1 Goals

Our goal in this section is to allow developers to instrument their system only once with metadata propagation for all the different end-to-end tracing applications. The code for metadata propagation is called *instrumentation module*, and unlike previous systems, we want a flexible extensible metadata format that can be shared. Instead of a fixed definition with, for example, a fixed length of data, we would like to have a dynamic *metadata container*.

To allow different tracing systems to utilize the same instrumentation module, the metadata container should be able to host any type of data and should isolate the metadata from different tracing systems. For example, the task ID used by X-Trace should, by no means, affect the tenant ID used by Retro. In addition, the costs of the data representation conversions for the container should be relatively

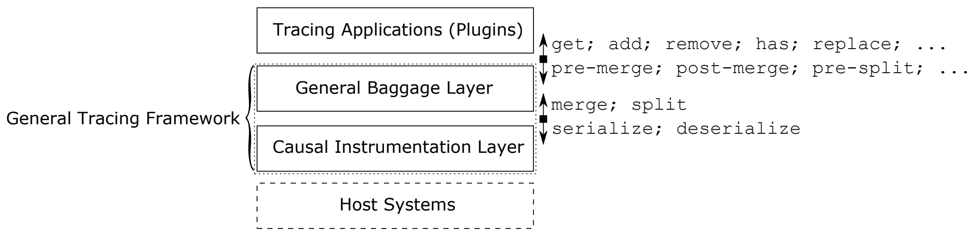
Tracing Applications (Plugins)
General Baggage Layer
Causal Instrumentation Layer
Host Systems
General Tracing Framework
get; add; remove; has; replace; ...
pre-merge; post-merge; pre-split; ...
merge; split
serialize; deserialize

Figure 3.2.1: General tracing framework is built up with layers, where baggage sits as the interface to the tracing applications.

cheap. We use the term *baggage* to refer to our proposed dynamic metadata container.

Different applications can use our baggage as they previously did with their own metadata. For example, X-Trace had a method `getTaskId` to retrieve the task ID. Instead of such direct access to the variables, X-Trace would now look up some key in the generic baggage. Similarly, instead of `XTraceContext.getThreadContext()` and `Retro.getTenantId()`, now there would only be one call: `Baggage.get()`.

3.2 Design

Based on the layered model by Fonseca [6], we propose a layered design as shown in Figure 3.2.1 where tracing applications are built atop the General Baggage Layer over the Causal Instrumentation Layer. We call these two layers *general tracing framework*, and the tracing applications that use this framework to propagate the tracing data is called *tracing plugins*. We call the systems that are being instrumented *host systems*, since they host the instrumentation module on the instrumentation layer.

3.2.1 Causal Instrumentation Layer

The instrumentation layer is the base layer of the general tracing framework to handle the logic of propagating metadata. This layer does not access the contents of the metadata, but simply specifies where the data needs to be propagated. Necessary function calls to this layer should be added to the host systems that are being instrumented.

The instrumentation layer only captures the propagation rules with the APIs provided by the baggage layer. To cross the boundary, the layer needs to use the serialization APIs (`serialize` and `deserialize`); to hand over the metadata along thread join and fork, the layer needs the baggage manipulation APIs (`merge` and `split`).

3.2.2 General Baggage Layer

The baggage layer is the interface between the causal instrumentation layer and the tracing applications. The baggage on this layer implements serializations and the baggage manipulation APIs to the lower layer, while it also exposes to the higher layer the data access APIs (`get`, `set`, etc.) and the event hooks related to baggage manipulations for different split/merge requirements by different tracing plugins. Therefore, the tracing plugins sitting on the higher layer have access to their data stored in the baggage and can register handlers on the event hooks.

An isolation of the data access between tracing plugins are necessary to avoid naming conflicts of the field keys. Although we do not fully address in this design, such isolation can also prevent a buggy tracing plugin from breaking other plugins and help on the security concerns aroused by the security auditing tracing applications built atop the causal metadata propagation.

3.2.2.1 Namespaces

Namespaces provide the logical separation of data between different tracing plugins. Each tracing plugin stores its metadata within its own namespace. For example, X-Trace would use the "`XTrace`" namespace, while Retro would use the "`Retro`" namespace. Therefore, different tracing applications can then share the same tracing infrastructure without interfering each other, where baggage is passed through the host system. Each tracing application should define their own unique namespaces and should only have access to their own namespaces.

3.2.2.2 Key-Values

In each namespace, a tracing application can define multiple slots to store the necessary information. The slots are identified by the keys, while the values are stored as sets that belong to the corresponding slots. When multiple baggage instances merge, by default, the new value set of a resulting slot is the union of the corresponding value sets from the parent baggage instances. Meanwhile, by default, a baggage instance splits into two identical baggage copies for different threads to propagate.

For example, in the case where we want to express IDs, all the IDs show up in the merged baggage for the plugin to process whenever it gets them, and if an ID does not change in branched baggage, the merged baggage will only include the same ID once to keep its size small.

3.2.2.3 Split and Merge

As a metadata container, baggage would branch and merge during instrumentation along with thread fork and join. These operations are challenging, because a general model really has no idea about what to do. For example, if we have two baggage instances $\{x = 1\}$ and $\{x = 2\}$, the possible merge output could be $\{x = 1\}$, $\{x = 2\}$, $\{x = 3\}$, or $\{x = 1, 2\}$ depending on what we want to do with it. Moreover, let's say we are counting something and get $\{x = 10\}$ at some point. If the baggage branches here what should the outcome baggage instances be? A simple duplicate might cause the count doubled to $\{x = 20\}$ at the merge point, while both sides might all need the result from previous counting.

Hence, different tracing applications have different requirements for split and merge. To be general, baggage must be able to support them all. For a particular attribute, all distinct values from the merged baggage instances should be kept as a minimum requirement. Tracing applications can implement their own rules and do their own fine tuning before and after the default split and merge provided by the general model.

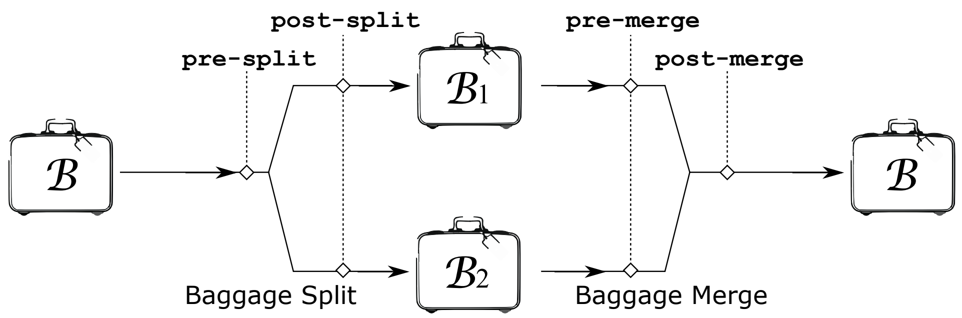

Figure 3.2.2: Events are triggered at the event hooks throughout baggage manipulations.

3.2.2.4 Event Hooks

Tracing plugins might have their own rules for baggage manipulations, especially for baggage merge. Although the default split and merge enable us to lazily resolve baggage merge, some applications might wish to eagerly operate the rules for split and merge. A tracing plugin can register event handlers to the corresponding event hooks on the baggage (Figure 3.2.2). Handlers are fed with the sub-baggage under the plugin's namespaces, and the tracing applications should not pose any assumption on the order in which the handlers attached by different plugins would run.

- `pre-split`$(\mathcal{B})$: a pre-processing on the baggage before the default splitting
- `post-split`$(\mathcal{B}_1, \mathcal{B}_2)$: a post-processing on the baggage after the default splitting
- `pre-merge`$(\mathcal{B}_1, \mathcal{B}_2)$: a pre-processing on the parent baggage instances before the default merging operation
- `post-merge`$(\mathcal{B})$: a post-processing on the baggage given by the default merging operation

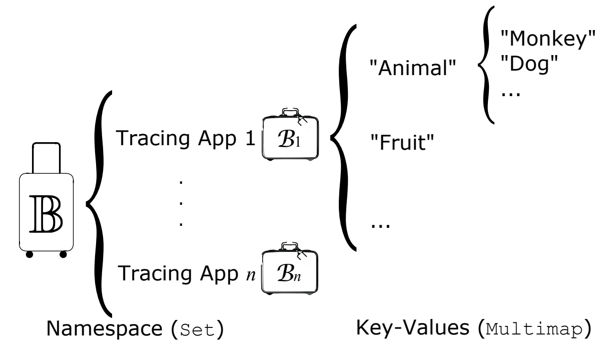

Figure 3.3.1: Baggage is implemented with multiple levels.

3.3 Implementation

As shown in Figure 3.3.1, baggage is implemented with Set and Google Guava Multimap [1], where Set is for the namespaces and Google Guava Multimap is for the key-values maps. Keys and values are represented by `ByteString`, an immutable byte-array provided by Google Guava, since common data types can be serialized to and deserialized from `ByteString` by Google Protocol Buffers [3] with a relatively low cost due to its lazy serialization feature. Meanwhile, we also use Protocol Buffers to deal with the serializations of the baggage when it goes across the network or other boundaries.

Figure 3.3.2 lists the Protocol Buffers message for the serialized representation of baggage. `BaggageMessage` is the main baggage object, and contains zero or more `NamespaceData` objects. `NamespaceData` contains the data for one namespace. It has its namespace key (*eg.* "`XTrace`"), stored as bytes, and zero or more `BagData` objects. Each `BagData` object has a key, and zero or more values. The application-level implementation takes the responsibility for the checking such as pruning `BagData` with zero values, pruning `NamespaceData` with zero bags, merging `BagData` with the same key, and merging `NamespaceData` with the same key.

Here are the APIs exposed to the instrumentation layer by the baggage:

```
package baggage;
option java_package = "edu.brown.cs.systems.baggage";
message BaggageMessage {
  /* Mapping of key to several values */
  message BagData {
    required bytes key = 1;
    repeated bytes value = 2;
  }
  /* Mapping of a namespace to a BagData */
  message NamespaceData {
    required bytes key = 1;
    repeated BagData bag = 2;
  }
  repeated NamespaceData namespace = 1;
}
```

Figure 3.3.2: The description of the Protocol Buffers Message for Baggage Serialization

- $\texttt{serialize}(\mathbb{B}) \mapsto \texttt{byteString}$
- $\texttt{deserialize}(\texttt{byteString}) \mapsto \mathbb{B}$
- $\texttt{split}(\mathbb{B}) \mapsto (\mathbb{B}_1, \mathbb{B}_2)$
- $\texttt{merge}(\mathbb{B}_1, \mathbb{B}_2) \mapsto \mathbb{B}$

Here are the APIs exposed to the plugins, which can only access their own namespaces:

- $\texttt{keys}()$
- $\texttt{get}(\texttt{key})$
- $\texttt{has}(\texttt{key}, \texttt{value})$
- $\texttt{add}(\texttt{key}, \texttt{value})$
- $\texttt{replace}(\texttt{key}, \texttt{valueSet})$
- $\texttt{remove}(\texttt{key})$

- Event Hooks: `pre-split`, `post-split`, `pre-merge`, `post-merge`

In summary, within the general framework, the lower instrumentation layer calls `merge` and `split` APIs respectively when threads join and fork, and it uses the serialization APIs provided by the baggage. A tracing application can register handlers with the baggage to implement its own metadata splitting rule and merging rule.

The performance test of our baggage implementation is shown in Table 3.3.1, and some of them have average CPU time larger than the average world time, which are the measurement errors due to the short time period between reading timestamps from the two sources.

Table 3.3.1: Performance Test of our baggage Implementation

		Count	Avg. Time (ns)	CPU Time (ns)
`add`	distinct keys; same namespace	1000000	753.43	300.04
`add`	distinct namespaces	100000	480.01	480.67
`add`	distinct values; same key	100000	1118.38	869.03
`add`	duplicated values; same key	10000000	43.76	43.82
`get`	distinct keys; same namespace	1000000	149.79	149.87
`get`	distinct namespaces	100000	228.50	228.82
`get`	non-exist keys	10000000	236.12	84.46
`get`	non-exist namespaces	10000000	34.01	34.04
`get`	single value; same key	10000000	193.52	43.15
`get`	a set of values; same key	10000000	52.07	42.70
`join`	2 disjoint keys * 1000 values	1000	223796.26	141020.36
`join`	2 disjoint namespaces * 10 keys * 100 values	10000	54993.82	55063.66
`join`	2000 * disjoint values	100	59785.33	59906.98
`remove`	distinct keys + remove namespaces	100000	551.03	551.51
`remove`	distinct keys; same namespace	1000000	272.02	272.03
`replace`	iterables	10000	1251511.80	1247822.86
`replace`	new keys	1000000	1029.65	341.32
`replace`	values	10000000	296.12	292.98
`split`	10 namespaces * 10 keys * 100 values	10000	779728.72	753578.68
`serialize`	10 namespaces * 10 keys * 100 values	10000	1131492.31	1055638.98
`deserialize`	10 namespaces * 10 keys * 100 values	10000	1017183.48	1008035.95

4 Existing Applications

In this section, we discuss about three existing tracing applications, and describe how we modified them to be tracing plugins utilizing the baggage model. For each tracing application, we give a brief description of the application as well as the metadata it uses, followed by our modifications.

4.1 X-Trace

It is hard for developers to diagnose on distributed systems due to its crossing-boundary nature. X-Trace [7] generates execution graphs based on metadata propagation. Events are created when the developers log anything or threads join. These events are collected and organized in the causal order by X-Trace for developers and operators to intuitively understand the behaviors of the applications for debugging and monitoring purposes.

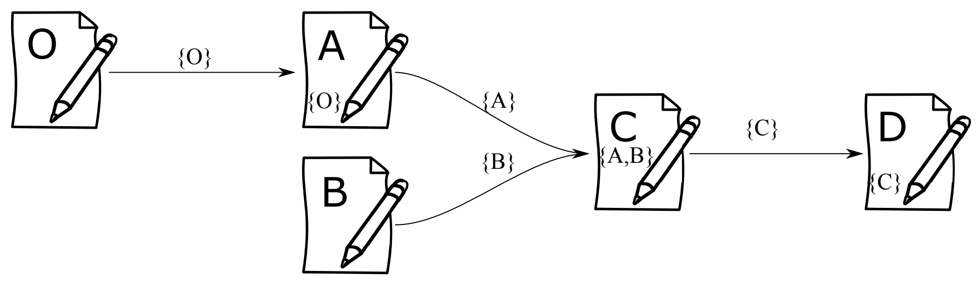

Figure 4.1.1: Report A's parent is Report O; Report C has two parents, Report A and Report B; Report D has the only parent of Report C.

4.1.1 Design

X-Trace provides an API called `logEvent`, which creates an event object and reports the event to X-Trace's reporting server. Each event has a randomly generated 8-byte ID. Each event also contains the IDs of all the immediate parent events that causally precede it. In order to know the parent events, X-Trace propagates the event ID of the most recently logged event. Events also include the task ID of the current request, which is also propagated along the request's execution. With the propagated parent event IDs, each event knows its immediate causal predecessors, which enables X-Trace to reconstruct the full execution graph of a request after receiving all of the events, as shown in Figure 4.1.1.

To summarize, each report includes the following fields:

- Task ID: a unique request identifier to distinguish it from other requests concurrently running in the guest application;
- Event ID: a unique identifier for this report;
- Parent Event IDs: identifiers for its causally parent reports;
- Timestamp; and
- Source information for the report including the hostname, the process ID, the process name, the thread ID, the thread name, the source code location, and the function call name where the report is generated.

The task ID is added to the metadata at the beginning of the execution for the task, while the parent event IDs are collected when the execution routines merge. All the other fields can either be generated or collected at the location where the event gets triggered.

X-Trace's metadata propagation fits the baggage model perfectly. In baggage, when multiple baggage instances merge, the same values in the same field are only kept as one copy, while different values in the same fields are all kept. Meanwhile, in X-Trace, task ID is not changed for the same request during the execution, so the Task ID field should be always the same single value. The event IDs from the parent baggage instances are merged automatically into a set, which contains all the parent event IDs. X-Trace only needs to read the set out, put to the Parent Event IDs field.

4.1.2 Implementation

Previously, X-Trace metadata was a fixed specification:

$$(\texttt{Flags}, \texttt{TaskID}, \texttt{ParentID}, \texttt{EventID}, \texttt{EdgeType}, \texttt{DestInfo}, \texttt{Options})$$

where only `Flags` and `EventID` are necessary to the execution graph reconstruction, and `ParentID` is the legacy field that denotes the immediate causally preceding event, which can actually be obtained during metadata propagation from `EventID` from the previous metadata.

As a result of the fixed size, the event IDs from the parent events were required to be reported at time threads join, because they could not be both carried by this metadata structure, which could only store one event ID. This added overheads.

In the new implementation with the general framework, X-Trace baggage contains task ID and event ID, and X-Trace does not register any baggage event handler. It creates a X-Trace event report with the following algorithm.

```
1: function XTRACE-REPORT($\mathcal{B}$, otherInfo)
2:     Define a report $R$
3:     if $\mathcal{B}$.TaskID is multivalued then
```

```
4:          raise TaskIdLeakage
5:      else
6:          newEventId ← EventIdGenerator()
7:          $R$.TaskID ← $\mathcal{B}$.TaskID
8:          $R$.ParentEventIDs ← $\mathcal{B}$.EventID          ▷ EventID can be multivalued after baggage merge
9:          $R$.EventID ← newEventId
10:         $R$.otherInfo ← otherInfo
11:         $\mathcal{B}$.EventID ← newEventId     ▷ Update the baggage with the new Event ID
12:         return $R$
13:     end if
14: end function
```

The new baggage does not have a fixed size, so the event IDs can be kept as multivalued and the reporting can be postponed, while due to the way we serialize the baggage, the size of metadata is increased.

4.2 Retro

Retro [10] is a multi-tenant resource management system that monitors the resource usage by tenants and molds their usage. It has an application-level instrumentation to record resource usage whenever resource API calls are made. For example, if a disk read API call is made, Retro records the resource usage and attributes it to the tenant making the disk read API call. To do this, a tenant identifier must be propagated alongside a request's execution. This propagation is done by a similar process of the metadata propagation in X-Trace.

4.2.1 Design

In Retro, an execution is associated to some tenant ID, which identifies the tenant that is responsible to these activities. This identifier is added to the metadata at the beginning of each execution and identifies the tenant at all the resource reporting points and the control points.

In baggage model, the tenant ID can be stored into a field for the propagation layer to propagate along the execution, and it should be set at the beginning of executions and removed in the end. Retro has access to it at any time during each execution, and tenant IDs should always be the same when threads merge, since by definition the executions for different tenants should not join.

4.2.2 Implementation

Retro baggage has a `TenantID` field, and it does not register any baggage event handler. When Retro retrieves Tenant ID from the baggage, it checks if the `TenantID` is single-valued. If multiple values show up in the field, executions for different tenant might join, which must be due to potential propagation layer bugs.

4.3 Pivot Tracing

Operators often need to correlate the statistics to do anomaly detections and root cause analysis, while such kind of work is hard since the data points they get from different nodes are not necessary aligned. Pivot Tracing [11] allows the operators to correlate the statistics generated at different points along an end-to-end execution based on causal metadata propagation.

4.3.1 Design

Pivot Tracing stores the intermediate results under the field for each query in the baggage. The results are grouped into tuple containers and get combined based on the query logic. At any point, the data stored in the baggage has an active instance and multiple inactive instances due to versioning, where the active one contains the result for the current branch of the execution. Their contents get merged when the execution rejoins.

4.3.2 Implementation

Pivot Tracing baggage uses the field name as the query identifier, and it has one namespace for the active instances and one for the inactive instances. Each instance is serialized and stored as a value of the corresponding field. It registers handlers to `pre-split` and `post-merge` events, where in the `pre-split` handler, the active instances are moved into the inactive namespace so that the active instance for each field is added into its inactive counterpart, and the active namespace is cleared for the new branch. For `post-merge`, the active instances from both sides are merged by merging the tuples represented by the instances.

5
Critical Path Analysis (CPath)

In this section, we describe a new tracing plugin called CPath, which is implemented based on the general tracing framework with the baggage model.

5.1 Motivation

Good performance is a major goal people are striving for. It is hard for performance engineers to figure out the slow-down factors, and it is even harder if multiple services get involved due to the large involvement of asynchrony and network crossing. The engineers need to set the goals for different teams which maintain the services in order to improve the overall product performance. However, it is likely that the goal makers are not confident if the goals they set could actually speedup the product as expected. It would be very helpful if they have a tool to tell them what modules might slow down their products and how much their optimization plans could work out.

In an execution graph, the overall latency of the execution is the longest path from the start point to the end point, which is called the *critical path*. Recall that in the concept of end-to-end tracing, metadata is propagated along the actual execution of the host system, in the same way latency gets accumulated. If we put a "timer" into the metadata to propagate along the execution while the "timer" gets paused at certain points where we are not interested in and resumed afterwards, we should be able to measure the critical latency as well as the details along the way the latency gets accumulated. These details should help discover the dominants of the overall latency.

When performance engineers have the critical path in hand, they can construct potential plans for performance optimization. Such plans can then be loaded into latency measurement to simulate the impacts on the overall latency. With the simulation results, the engineers can improve their plans to make the performance goals more meaningful and the expectations more realistic.

5.2 Design

Theoretically, a critical path is the productively[1] longest path from the start point to the end point in the execution graph of a request, and the critical path latency is the productive CPU time on this path. At any node v in this graph, we can represent the longest path from the start point by d_v, which is the critical path length of this partial execution. As proposed by Hollingsworth[8], for any node u and its preceding node set P_u, we have

$$d_u = \max_{v \in P_u} (d_v + l_{v \to u})$$

where $l_{v \to u}$ is the edge length from v to u.

So, we store the timestamp at the start of each run, and we can track the la-

[1]The edge lengths in the execution graph are the productive CPU time, while the non-productive counterpart is excluded.

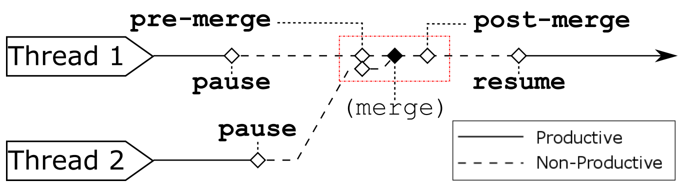

Figure 5.2.1: When a thread joins another one, each of them pauses their experiments, and they get merged and resumed.

tency by subtracting the timestamp stored in baggage with timestamp at the end point. To eliminate the non-productive CPU time [9] spent on overheads on the operation system or the tracing system, we break such latency measurement into small productive segments. The latency measurement (or the "timer") is paused before any non-productive operations to checkout the latency from the last timestamp where it starts and is resumed afterwards to timestamp the new start point. When threads join, we should take the baggage which has the largest intermediate latency, since the other branch would wait if this branch takes longer time.

The process of a thread joining another is as shown in Figure 5.2.1, where Thread 1 and Thread 2 both pause their measurements to checkout the intermediate latencies accumulated so far when they initiate a thread join operation, and the baggage instances from both sides are fed to the `pre-merge` handlers, merged by the lazy merge operation in the general model, and fed to the `post-merge` handlers, and then the measurement resumes by updating the timestamp inside the baggage. Other operations are measured in the same way.

An example measurement is shown in Figure 5.2.2, where the metadata along the highlighted path is collected in the end to show the critical path. At the merge point *d*, the lower branch is taken since its intermediate latency of 11ms is larger, while at *e*, the lower one is taken and at *f* the lower one is taken. When the baggage reaches the end point, the critical path latency measured is 28ms.

Therefore, the critical path latency is calculated while the host system is running. We also keep track with the critical path information with d_v in the baggage that is handed over throughout the system so that we can reconstruct the path.

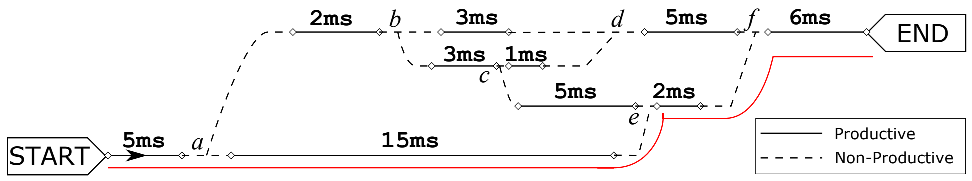

Figure 5.2.2: Each time baggage merges, the one with the largest latency from the start point is taken to be propagated. A critical path is then obtained in the baggage collected at the end point.

Although the critical path can be calculated offline by collecting all the events with tracing systems like X-Trace, an execution graph might have too many events to be scalable. In contrast, with the online algorithm described above, we only need to collect the baggage at the nodes whose partial critical paths are interesting to us. The overall critical path is retrieved from the baggage from the very last node in the execution graph, where the execution terminates.

5.2.1 Slack and Top *N* Critical Paths

The critical path shows the worst case of the performance optimization and implies the possible modules to be considered for speedup, but the developers would be also interested in the upper bound of the optimization on the critical path – namely, how much latency can be taken out before the critical path shift to a subcritical one in the former measurement. Hollingsworth [9] has proposed the metric of *slack* to characteristic the lower bound of possible improvement on a procedure. This metric is helpful for developers to figure out where which part of the program can be optimized.

Surprisingly, it is not very hard to adapt the CPath algorithm to get additional information. With CPath, we can instrument the top *N* critical paths with minor modifications, and figure out the slack in the much smaller execution graph that consists of these top critical paths.

5.2.2 Hypothetical Speedups

Meanwhile, it would be great if we could know how much an optimization plan can work out for the overall performance (saying the overall running latency). The speedup goals in the optimization plans should be verified to make sure feasible and helpful to the global speedup before they are assigned to the development teams.

Since CPath instrumentation is done in real-time when requests are being served, we can simulate the impacts on the critical path and other measurements of speeding up different parts of the execution by discounting the latency measured within the parts that are planed to speed up, which is so-called the *hypothetical speedup*.

A measurement with such kind of hypothetical speedups is called an *experiment*, while a baseline experiment is a trivial case which does not do any hypothetical speedup. An experiment comes with the designated speedup parts, the *hypothetical speedup zone*, and a latency discount factor to describe the speedup goals. Having multiple experiments set up would help observe the difference between different speedup plans in terms of the critical path latency.

5.2.3 Practical Constraints

This measurement requires necessary bookkeeping in baggage. If the host system is built atop multiple subsystems, we would need all the subsystems to have CPath running; otherwise, if a subsystem that lacks of CPath does not resume the latency measurement, the overall critical path would ignore the time spent within this subsystem.

We are also assuming the clock drift on each machine is so subtle that in the tiny time period of a measurement segment it does not affect the measurement. Note that the clocks on different machines are not necessary to be synchronized. Network crossing is non-productive, and we only take the difference of the timestamps taken on the same machines when calculating incremental productive CPU time.

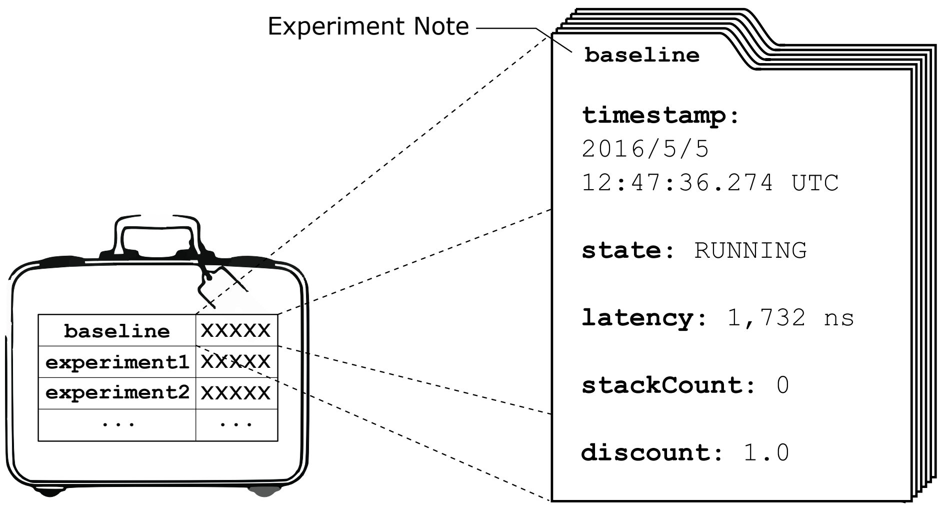

Figure 5.3.1: The baggage in CPath uses the key-values pairs for experiments, where each key refers to a critical path experiment and values are the serialized experiment metadata possibly from the threads it merged from.

5.3 Implementation

5.3.1 Baggage

For critical path instrumentation, we need the following metadata:

- Timestamp: the timestamp of the most recent baggage update, where the latency might be accumulated
- State: the state which indicates if the experiment is paused or resumed
- Latency: the accumulated latency of the critical path from the execution begin to the current point
- Stack-Count: the count of how many times the current thread has entered the speedup zone
- Discount: the discount factor which should apply within the hypothetical speedup zone

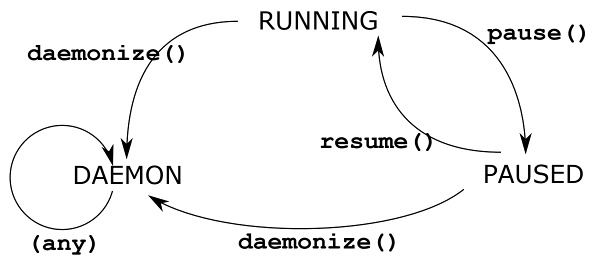

Figure 5.3.2: The experiment state pauses or resumes the latency measurement.

A critical path experiment is represented by these fields, and the structure that organizes them is called an *Experiment Note*, whose serialization is stored in the critical path namespace of the baggage under an experiment key (as shown in Figure 5.3.1). A critical path baggage can contain multiple experiments for possible speed up plans, among which a baseline experiment does the measurement without any hypothesis.

STATES The state of a critical path experiment can be RUNNING, PAUSED, or DAEMON, where DAEMON is a special pausing state introduced in subsubsection 5.3.2.4 to avoid the effects from daemon threads. Their transition diagram is shown in Figure 5.3.2.

START EXPERIMENT To define an experiment, we only need to add the necessary metadata to baggage. The experiments should be defined when a request execution starts, and the baseline experiment is defined by default. Predefined experiments can be started by static injection with AspectJ, while other experiments can be dynamically declared with Dynamic Instrumentation.

5.3.2 HANDLERS

As shown in Figure 5.2.1, the latency measurement is closely tied to thread activities, which are not supported by the baggage layer but the propagation layer. CPath injects propagation handlers with AspectJ and Dynamic Instrumentation to pause

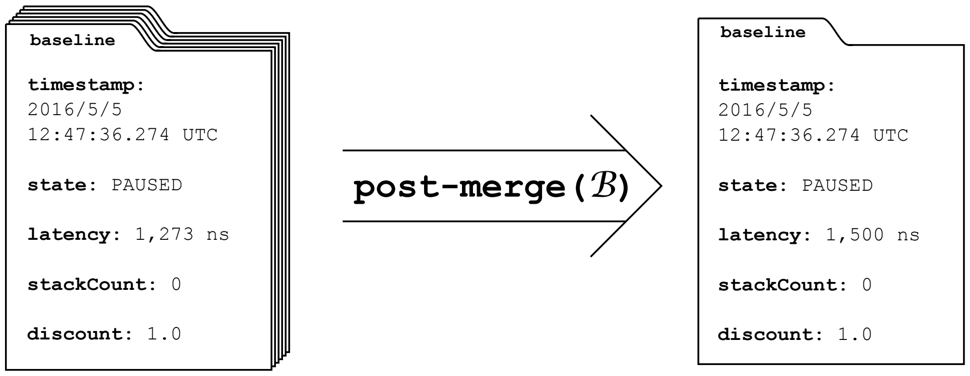

Figure 5.3.3: The experiment note with the largest latency is taken at `post-merge`.

and resume the measurement. Meanwhile, CPath implements its baggage merging rule with a `post-merge` event handler based on the general baggage model after a general baggage merge, which merges the multimaps in the baggage.

5.3.2.1 Baggage Handler

POST-MERGE Handler For each experiment, this handler collapses the experiment notes by taking the one with the largest latency after a baggage merge (Figure 5.3.3).

```
1: function POST-MERGE($\mathcal{B}$)
2:     $t \leftarrow$ CurrentTime
3:     for $B_i \in \mathcal{B}$ do                                    ▷ for each experiment $B_i$
4:         $b_{i\max} \leftarrow \text{argmax}_{b_{ij} \in B_i} (b_{ij}.\texttt{latency})$
5:         if $b_{i\max}$ is in DAEMON but $\exists b_{i,j} \in B_i$ is not DAEMON then
6:             $b_{i\max}$.state $\leftarrow$ PAUSED                     ▷ exit DAEMON state
7:         end if
8:         $B_i \leftarrow \{b_{i\max}\}$                               ▷ only keep one experiment note
9:     end for
10:    return $\mathcal{B}$
11: end function
```

5.3.2.2 Statically Injected Handlers

The following handlers can be statically injected to the metadata propagation layer with AspectJ.

Pause Each time before the host system goes for non-productive overheads (such as forking or joining threads), the critical path measurement of each experiment inside the baggage should be paused to avoid measuring non-productive CPU time, where the `latency` field should be added with the latency accumulated since last resume, and the experiment state should be transitioned to paused.

```
function PAUSE($\mathcal{B}$)
    $t \leftarrow$ CurrentTime
    for $B_i \in \mathcal{B}$ do                                  ▷ for each experiment $B_i$
        for $b_{ij} \in B_i$ do                               ▷ for each experiment note $b_{ij}$
            $\Delta t \leftarrow t - b_{ij}$.timestamp
            if $b_{ij}$.state $=$ RUNNING then
                if $b_{ij}$.stackCount $> 0$ then          ▷ in hypothetical speedup zone
                    $b_{ij}$.latency $\leftarrow b_{ij}$.latency $+ b_{ij}$.discount $\times \Delta t$
                else                              ▷ outside hypothetical speedup zone
                    $b_{ij}$.latency $\leftarrow b_{ij}$.latency $+ \Delta t$
                end if
                $b_{ij}$.timestamp $\leftarrow t$
                $b_{ij}$.state $\leftarrow$ PAUSED
            end if
        end for
    end for
    return $\mathcal{B}$
end function
```

RESUME When the host system finishes its non-productive activity, the measurement should be resumed by updating the `timestamp` field and transitioning the experiment state to RUNNING.

```
1:  function RESUME($\mathcal{B}$)
2:      $t \leftarrow$ CurrentTime
3:      for $B_i \in \mathcal{B}$ do                          ▷ for each experiment $B_i$
4:          for $b_{ij} \in B_i$ do                       ▷ for each experiment note $b_{ij}$
5:              if $b_{ij}$.state = PAUSED then
6:                  $b_{ij}$.timestamp $\leftarrow t$
7:                  $b_{ij}$.state $\leftarrow$ RUNNING
8:              end if
9:          end for
10:     end for
11:     return $\mathcal{B}$
12: end function
```

5.3.2.3 DYNAMICALLY INJECTED HANDLERS

The following handlers can be injected with Dynamic Instrumentation for the hypothetical speedups while the host system is running. They are adjustable at runtime and can be injected based on pattern matching with the source code of the host system. For the time being, we only support hypothetical speedups on single functions. The `enterFunc` can be injected to the beginning of a selected function for an experiment where the function should be hypothetically optimized. The `exitFunc` is injected to the end of the corresponding function. These functions maintain a counter that represents the number of times the selected function shows up in the current callstack.

ENTERFUNC

```
1:  function ENTERFUNC($\mathcal{B}$, ExperimentID)
2:      $\mathcal{B} \leftarrow$ PAUSE($\mathcal{B}$)                  ▷ start doing non-productive work
```

```
3:     take $B_i \in \mathcal{B}$ such that $i = \texttt{ExperimentID}$
4:     for $b_{ij} \in B_i$ do
5:         $b_{ij}.\texttt{stackCount} \leftarrow b_{ij}.\texttt{stackCount} + 1$
6:     end for
7:     $\mathcal{B} \leftarrow \textsc{resume}(\mathcal{B})$
8:     return $\mathcal{B}$
9: end function
```

ExitFunc

```
1: function exitFunc($\mathcal{B}, \texttt{ExperimentID}$)
2:     $\mathcal{B} \leftarrow \textsc{pause}(\mathcal{B})$
3:     take $B_i \in \mathcal{B}$ such that $i = \texttt{ExperimentID}$
4:     for $b_{ij} \in B_i$ do
5:         $b_{ij}.\texttt{stackCount} \leftarrow b_{ij}.\texttt{stackCount} - 1$
6:     end for
7:     $\mathcal{B} \leftarrow \textsc{resume}(\mathcal{B})$
8:     return $\mathcal{B}$
9: end function
```

5.3.2.4 Daemon Thread Declaration

A thread might run as a daemon to periodically check the states of others to wait for other threads. In this case, it does not make sense to count such waiting as productive time; otherwise, the path passing through this thread would probably become the critical path, while the daemon thread does not determine the running time in the context of performance optimization. To avoid this issue, we allow the developers to declare a DAEMON state for a thread, which lasts until the thread merges into a non-daemon thread.

Daemonize

```
1: function daemonize($\mathcal{B}$)
```

Table 5.3.1: CPath Performance

	Count	Avg. Time (ns)
`start`	1000000	446.3
`pause/resume`	1000000	876.3

```
2:      $\mathcal{B} \leftarrow \textsc{pause}(\mathcal{B})$          ▷ pause the experiment to checkout new latency
3:      for $B_i \in \mathcal{B}$ do
4:          for $b_{ij} \in B_i$ do
5:              if $b_{ij}$.state $\neq$ DAEMON then
6:                  $b_{ij}$.state $\leftarrow$ DAEMON
7:              end if
8:          end for
9:      end for
10:     return $\mathcal{B}$
11: end function
```

5.3.3 Performance Considerations

The performance of CPath operations is shown in Table 5.3.1, where the second row is the average of both `pause` and `resume`. The Experiment Note needs frequent modifications, so we implement the serialization by concatenating fields directly instead of Protocol Buffers, because Protocol Buffers seems not helpful in terms of performance in the CPath scenario according to our comparisons.

The slowness of using Protocol Buffers might be due to the heavy usage of serialization. To use lazy serialization, we should leverage the Protocol Buffers structure instead of loading all fields to our own objection and dumping them back to Protocol Buffers later. Meanwhile, the lazy serialization only helps reduce the overhead of serialization, not the one of deserialization.

5.3.4 Applications

We have instrumented HDFS, MapReduce, and Spark with CPath. However, their pipelining processing seems to prevent the parallelism that is complicated enough to use critical path analysis. In those architectures, the execution graphs are broken into stages and the slow-down factors are generally obvious to find. We believe the critical path analysis would be useful for the systems which have more complex parallelisms, such as microservices.

6 Conclusions

In this report, we have proposed a general baggage model as a general metadata container that can be used for different tracing applications. With this general model, we are able to share the metadata propagation code by implementing tracing applications as tracing plugins based on the general tracing framework. Such reuse removes redundancy and saves the developers from the tricky part of adding metadata propagation to their systems, which allows them to easily use different tracing systems. We have modified for X-Trace, Retro, and Pivot-Tracing based on our design.

With the general model, we have also proposed a new tracing application, CPath, running as a tracing plugin atop the tracing framework to measure the running latency information and verify potential optimization plans based on hypothetical speedup. The application can help developers and operators to figure out the slowdown factors and set up realistic optimization plans.

Lessons Learnt Our baggage implementation did not utilize the features such as lazy serializations provided by the third-party packages, and the baggage can have better performance with more careful implementation.

As for CPath, we found it is hard to analyze critical paths when the system being instrumented has a heavy use of pipelining (*eg.* Hadoop) and buffering, since their slow-down factors are obvious. Hadoop was not the best choice for critical path, since its execution is broken into multiple stages due to synchronizations so that the execution graph is never complex and the slow-down factors become obvious. A more suitable case could be with the microservices architecture, where requests go to different microservices with complicated parallelism. It is generally hard for performance engineers to understand what is going on exactly within such kind of architecture due to the number of microservices getting involved for a single request, and an online automated tool like CPath would definitely help.

Meanwhile, critical path should not count non-productive CPU time, and this is done by pausing and resuming measurements before and after the non-productive work. However, if a pause or resume is missing, the measurement could get extra latency in.

Future Work We would like to have much more optimized baggage with lower overhead presumably by utilizing lazy serialization and reducing the copying operations by implementing copy-on-write, and we would like to evaluate the alternatives to Google Protocol Buffers to see if we can get better performance.

For CPath, we would like to verify our tracing application with the systems which have more complicated parallelism. We also want to support more flexible definitions for the hypothetical speedup zone and figure out a way to improve the measurement accuracy.

References

[1] Guava: Google core libraries for Java.
https://github.com/google/guava.

[2] Apache HTrace – a tracing framework for use with distributed systems.
https://htrace.incubator.apache.org/.

[3] Protocol Buffers – Google's data interchange format.
https://developers.google.com/protocol-buffers.

[4] OpenZipkin – a distributed tracing system. https://zipkin.io/.

[5] Anupam Chanda, Khaled Elmeleegy, Alan L Cox, and Willy Zwaenepoel. Causeway: Operating system support for controlling and analyzing the execution of distributed programs. In *HotOS*, 2005.

[6] Rodrigo Fonseca and Jonathan Mace. We are losing track: a case for causal metadata in distributed systems. 2015.

[7] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-Trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 20–20. USENIX Association, 2007.

[8] Jeffrey K Hollingsworth. An online computation of critical path profiling. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 11–20. ACM, 1996.

[9] Jeffrey K Hollingsworth and Barton P Miller. Slack: a new performance metric for parallel programs. *University of Maryland and University of Wisconsin-Madison, Tech. Rep*, 1994.

[10] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 589–603, 2015.

[11] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 378–393. ACM, 2015.

[12] 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. URL `http://research.google.com/archive/papers/dapper-2010-1.pdf`.