Principled workflow-centric tracing of distributed systems

Raja R. Sambasivan* Ilari Shafer◇ Jonathan Mace‡ Benjamin H. Sigelman† Rodrigo Fonseca‡ Gregory R. Ganger*

*Carnegie Mellon University, ◇Microsoft, ‡Brown University, †LightStep

Abstract

Workflow-centric tracing captures the workflow of causally-related events (e.g., work done to process a request) within and among the components of a distributed system. As distributed systems grow in scale and complexity, such tracing is becoming a critical tool for understanding distributed system behavior. Yet, there is a fundamental lack of clarity about how such infrastructures should be designed to provide maximum benefit for important management tasks, such as resource accounting and diagnosis. Without research into this important issue, there is a danger that workflow-centric tracing will not reach its full potential. To help, this paper distills the design space of workflow-centric tracing and describes key design choices that can help or hinder a tracing infrastructure's utility for important tasks. Our design space and the design choices we suggest are based on our experiences developing several previous workflow-centric tracing infrastructures.

Categories and Subject Descriptors C.4 [*Performance of systems*]: Measurement techniques

1 Introduction

Modern distributed services running in cloud environments are large, complex, and depend on other similarly complex distributed services to accomplish their goals. For example, user-facing services at Google often comprise 100s to 1000s of nodes (e.g., machines) that interact with each other and with other services (e.g., a spell-checking service, a table-store [6], a distributed filesystem [21], and a lock service [6]) to service user requests. Today, even "simple" web applications contain multiple scalable and distributed tiers that interact with each other. In these environments, *machine-centric* monitoring and tracing mechanisms (e.g., performance counters [36] and `strace` [49]) are insufficient to inform important management tasks, such as diagnosis, because they cannot provide a coherent view of the work done among a distributed system's nodes and dependencies.

To address this issue, recent research has developed *workflow-centric* tracing techniques [8, 9, 18, 19, 22, 29, 33, 34, 41, 44, 45, 51], which provide the necessary coherent view.

SoCC '16 October 05-07, 2016, Santa Clara, CA, USA.
Copyright is held by the owner/author(s).
ACM ISBN 978-1-4503-4525-5/16/10.
DOI: http://dx.doi.org/10.1145/2987550.2987568

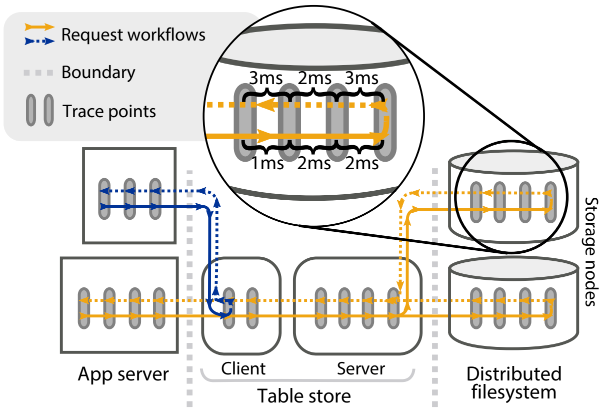

Figure 1: Workflows of two READ requests.

These techniques identify the workflow of causally-related events within and among the nodes of a distributed system and its dependencies. As an example, the workflow-centric traces in Figure 1 show the workflows of the events involved in processing two read requests in a three-tier distributed system. The first request (blue) hits in the table store's client cache, whereas the second (orange) requires a file system access. The workflow of causally-related events (e.g., a request) includes their order of execution and, optionally, their structure (i.e., concurrency and synchronization) and detailed performance information (e.g., per-function or per-trace-point latencies).

To date, workflow-centric tracing has been shown to be sufficiently efficient to be enabled continuously (e.g., Dapper incurs less than a 1% runtime overhead [45]). It has also proven useful for many important management tasks, including diagnosing anomalies and steady-state performance problems, resource-usage attribution, and dynamic monitoring (see Section 2.1). There are a growing number of industry implementations, including Apache's HTrace [4], Zipkin [56], Google's Census [22], Google's Dapper [45], LightStep [32], and others [7, 12, 13, 52]. Many of the industry implementations follow Dapper's model. Looking forward, workflow-centric tracing has the potential to become the fundamental substrate for understanding and analyzing many, if not all, aspects of distributed-system behavior.

But, despite the strong interest in workflow-centric tracing infrastructures, there is very little clarity about how they should be designed to provide maximum benefit. New research papers that advocate slightly different tracing infrastructure designs are published every few years—e.g., Pinpoint [9], Magpie [5], Pip [41], Stardust and Stardust-revised [44, 51], Mace [29], Whodunit [8], Dapper [45], X-Trace and X-Trace-revised [18, 19], Retro [33], and Pivot Tracing [34]—but there exists little insight about which designs should be preferred under different circumstances. Without research into this important question, there is a danger that future tracing implementations will not live up to expectations and that the potential of workflow-centric tracing will be squandered. This question is especially relevant today because of practitioners' emerging interest in creating a common high-level API for workflow-centric tracing within open-source software [38]. Understanding the breadth of tracing designs and why they differ can help in designing APIs that don't artificially limit workflow-centric tracing's utility for important tasks.

In this paper, we answer the following question: "What design decisions within a workflow-centric tracing infrastructure dictate its utility for important management tasks?" We do this via a systematic analysis of the key design axes of workflow-centric tracing. We distill these axes, identify commonly used options for each, and identify design points across them that will increase (or hurt) a tracing infrastructure's utility for various tasks.

Our design axes and choices for them are motivated by our experiences designing some of of the most well-known tracing infrastructures (Stardust [44, 51], X-Trace [18, 19], Dapper [45], Retro [33], and Pivot Tracing [34]) and working in a startup that instruments production code to enable workflow-centric tracing (LightStep [32]). We often draw on our experiences building and using Spectroscope [43, 44], a tool that uses workflow-centric tracing to automatically localize the root cause of performance regressions in distributed systems. Our initial design re-used a tracing infrastructure (Stardust [51]) that had previously been used for resource attribution, but it proved ineffective and expensive when used for diagnosis tasks. Our experiences revising the original tracing infrastructure helped inform several of the insights in this paper.

Overall, we find that resource attribution and performance-related management tasks benefit from different design decisions. We also find that using a tracing infrastructure best suited for one management task for another type of task will not only yield poor results, but can also result in inflated tracing overheads. Though our design axes and options are not comprehensive, they are sufficient to distinguish existing tracing infrastructures and the management tasks for which they are best suited.

In summary, we present the following contributions:

1) We distill five key design axes that dictate workflow-centric tracing's utility for important management tasks.
2) We identify potential choices for each axis based on our experiences and a systematic analysis of previous literature. Using scenarios drawn from our experiences, we describe which options are best suited for which management tasks and which will lead to poor outcomes.
3) We contrast existing tracing infrastructures' choices to our suggestions to understand reasons for any differences.

The remainder of this paper is organized as follows. Section 2 introduces workflow-centric tracing and the management tasks we consider. Sections 3–5 describe various design axes and their tradeoffs. Section 6 suggests specific design choices for the management tasks and compares our suggestions to existing infrastructures' choices. Section 7 discusses promising future research avenues. Section 8 concludes.

2 Anatomy

Figure 2 illustrates the anatomy of how workflow-centric tracing infrastructures are used for management tasks. There are two levels. At the top level, there are applications that execute management tasks by using the data exposed by the tracing infrastructure. These applications typically execute tasks out-of-band of (i.e., separately from) the tracing infrastructure. Some more recent infrastructures allow applications to statically [8] or dynamically [34] configure aspects of the tracing infrastructure at runtime. This creates the potential to execute tasks in-band (i.e., at least partially within) the infrastructure.

At the lower level, there is a workflow-centric tracing infrastructure, which exposes different types of information about the workflows it observes. It propagates metadata (e.g., an ID) with causally-related events to distinguish them from other concurrent, yet unrelated, events in the distributed system. Doing so requires mostly *white-box* distributed systems whose components can be modified to propagate metadata. Less intrusive methods exist for inferring causally-related events—e.g., correlating network messages [2, 30, 42, 46], correlating pre-existing logs [5, 27, 48, 55], or using models to identify expected causal relationships [48])—but they generate less accurate data and cannot execute management tasks in-band (see Section 2.3). As such, metadata propagation has emerged as the preferred method in many production environments [4, 7, 12, 22, 38, 45, 56] and is the focus of this paper.

Section 2.1 describes the management tasks most commonly associated with workflow-centric tracing. Sections 2.2 through 2.4 describe the components of most workflow-centric tracing architectures. We distinguish between conceptual design choices, core software components, and additional ones. The conceptual choices dictate the fundamental capabilities of a workflow-centric tracing architecture. The core software components work to implement these choices. The

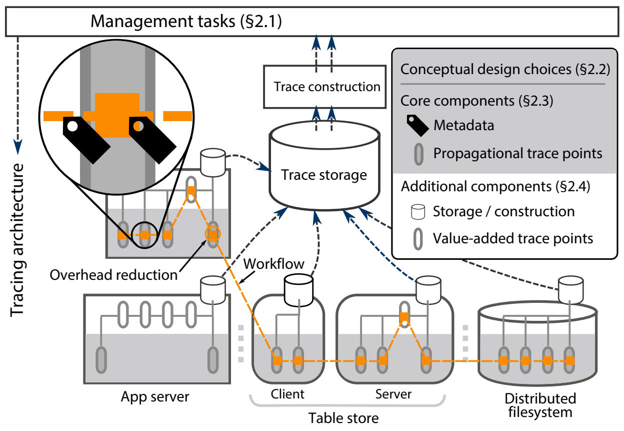

Figure 2: Anatomy of workflow-centric tracing.

additional components work to limit overhead so that tracing can be used in production and to enrich the infrastructure's utility for specific management tasks.

2.1 Management tasks

Table 1 summarizes workflow-centric tracing's key management tasks and lists tracing implementations best suited for them. Some of the listed infrastructures were initially thought to be useful for more tasks than those attributed to them in the table. For example, we initially thought that the original Stardust [51] would be useful for both resource attribution and diagnosis. Similarly, Google's Dapper has proven less useful than initially thought because it cannot detect anomalies [39]. One goal of this paper is to help future tracing developers avoid such mismatches between expectation and reality.

Identifying anomalous workflows: This task involves presenting rare workflows that are extremely different from other workflows to diagnosis teams so that they can analyze why they occur. They fall in the tail (e.g., 99.9th percentile) of some important distribution (e.g., response times). They may occur as a result of correctness problems (e.g., component timeouts or failures) or performance issues (e.g., a slow function or waiting for a slow thread).[1] They usually exhibit uncommon structures—i.e., causal order of work executed, amount of concurrency, or locations of forks and joins—latencies, or resource usages. Pinpoint [9] and Pip [41] are suited to identify anomalies. The Mace programming framework [29] embeds a workflow-centric tracing infrastructure into distributed systems that provides functionality similar to Pip.

Identifying workflows w/steady-state problems: This task involves presenting workflows that negatively affect the mean or median of some important performance distribution—e.g., the 50th or 75th percentile of request response times—to diagnosis teams so that they can understand why they occur. Unlike anomalies, they are not rare. The problems they represent manifest in workflows' structures, latencies, or resource usages. One example we have seen is a configuration change that modifies the storage nodes accessed by a large set of requests and increases their response times [44]. Dapper [45], Mace [29], Pip [41], Pinpoint [9], the revised version of Stardust (Stardust‡ [44]), and both versions of X-Trace [18, 19] are all useful for identifying steady-state problems.

Distributed profiling: This task involves identifying slow functions or nodes. Since the execution time of a function often depends on how it is invoked, tracing infrastructures explicitly designed for this purpose, such as Whodunit [8], present functions' latencies as histograms in which bins represent unique calling stacks or backtraces; workflow structures are not preserved. Tracing implementations suited for identifying anomalies or steady-state problems can also be used for profiling. We list Dapper [45] in Table 1 as an example.

[1] In most distributed systems, correctness problems are often masked by retries and fail-overs, so they initially appear to be performance problems [23, 41]. As such, we do not distinguish between the two in this paper.

Type	Management task	Implementations
Perf.	Identifying anomalous workflows	Mace [29], Pinpoint [9], Pip [41]
	Identifying workflows w/steady-state problems	Dapper [45], Mace [29], Pinpoint [9], Pip [41], Stardust‡ [44], X-Trace [19], X-Trace‡ [18]
	Distributed profiling	Dapper [45], Whodunit [8]
	Meeting SLOs	Retro [33]
	Resource attribution	Retro [33], Stardust [51], Quanto [17]
Multiple	Dynamic monitoring	Pivot Tracing [34]

Table 1: Management tasks most commonly associated with workflow-centric tracing. This table lists workflow-centric tracing's key management tasks and tracing implementations suited for them. Some implementations appear for multiple tasks. The revised versions of Stardust and X-Trace are denoted by *Stardust‡* and *X-Trace‡*.

Meeting SLOs: This task involves adjusting workflows' resource allocations to guarantee that jobs meet service-level objectives. Resource allocations are dynamically changed during runtime. Retro [33] is suited for this task.

Resource attribution: The task involves tying work done at an arbitrary component of the distributed system to the client or request that originally submitted it, perhaps for billing [53] or to guarantee fair resource usage [33]. Retro [33], the original version of Stardust [51], and Quanto [17] are suited for this task. Retro and Stardust attribute per-component resource usage (e.g., CPU time) to clients in distributed storage systems or databases. Quanto ties per-device energy usage to high-level activities (e.g., routing) in distributed-embedded systems.

Dynamic monitoring: The task involves monitoring activity (e.g., bytes read) at a distributed component only if that activity is causally-related to pre-conditions met at other components. Both the activity to monitor and the pre-conditions are dynamically chosen at runtime. For example, one might choose to monitor bytes read at a database only by users whose requests originate in China. Pivot Tracing [34] is currently the only infrastructure suited for this task. Tracing implementations that fall in this category have the potential to, but cannot necessarily, support some of the other tasks listed above (e.g., resource attribution). This is because more than what is instrumented needs to be dynamically changed to support them.

2.2 Conceptual design choices

What causal relationships should be preserved?: The most fundamental goal of a workflow-centric tracing infrastructure is to identify and preserve causal relationships. However, preserving all causal relationships can result in too much overhead, whereas preserving the wrong ones can result in a tracing infrastructure that is not useful for its intended management tasks. For example, our initial efforts in developing Spectroscope [44] were hampered because the original version of Stardust [51] preserved causal relationships that turned out not to be useful for diagnosis tasks. Section 3 describes various causality choices we have identified in the past and the management tasks for which they are suited.

What model should be used to express causal relationships?: There are two kinds: specialized and expressive models. Specialized ones can only represent a few types of relationships, but admit efficient storage, retrieval, and computation; expressive ones make the opposite tradeoff. Paths and directed trees are the most popular specialized models. The most popular expressive model is a directed acyclic graph (DAG).

Paths, used by Pinpoint [9], are sufficient to represent synchronous behavior, event-based processing, or to associate important data (e.g., a client ID) with multiple causally-related events. Directed trees are sufficient for expressing sequential, concurrent, or recursive call/reply patterns (e.g., as seen in RPCs). Concurrency (i.e., multiple events that depend on a single event) is represented by branches. They are used by the original X-Trace [19], Dapper [45], and Whodunit [8].

Trees cannot represent events that depend on multiple other events. Examples include synchronization (i.e., a single event that depends on multiple concurrent previous ones) and inter-request dependencies. Since preserving synchronization is important for diagnosis tasks (see Section 3.2), Pip [41], Pivot Tracing [34], and the revised versions of Stardust [44] and X-Trace [18] use DAGs instead of directed trees. Retro [33] and the original Stardust [51] use DAGs to preserve inter-request dependencies due to aggregation (see Section 5.1).

2.3 Core software components

Metadata: These are fields that are propagated with causally-related events to identify their workflows. They are typically carried within thread- or context-local variables. They are also carried within network messages to identify causally-related events across nodes.

To execute management tasks out-of-band, tracing infrastructures need only propagate unique IDs and logical clocks, such as single logical timestamps [31] or interval-tree clocks [3], as metadata. Such metadata is persisted to disk and used to construct traces of workflows asynchronously from the tracing infrastructure. The traces are then used to execute tasks. Single timestamps are small, but result in lost traces in the face of failures. Interval-tree clocks take up space proportional to the amount of concurrent threads in the system, but are resilient to failures. Many tracing infrastructures support only out-of-band execution [8, 9, 18, 19, 41, 44, 45, 51].

To execute tasks in-band, data relevant to them must be propagated as metadata. This may include logical clocks. In contrast to out-of-band execution, in-band execution reduces the amount of data persisted by workflow-centric tracing. It also makes it easier for management tasks to be executed online, hence resulting in fresher information being used. Several new tracing infrastructures support in-band execution [8, 22, 33, 34].

Propagational trace points: Trace points indicate events executed by individual workflows. They must be added by developers to important areas of the distributed system's software. Propagational trace points are fundamental to workflow-centric tracing as they are needed to propagate metadata across various boundaries (e.g., network) to identify workflows. For example, they are needed to insert metadata in RPCs to identify causally-related events across nodes. They are also needed to identify the start of concurrent activity (fork points in the code) and synchronization (join points).

When executing management tasks out-of-band, *trace-point records* of propagational trace points accessed by workflows are persisted to disk along with relevant metadata and used to construct traces. Trace-point records contain the trace-point name and other relevant information captured at that trace point (e.g., a timestamp, variable values, etc.) For in-band execution, propagational trace points simply transfer metadata across boundaries. Trace points, both propagational and value-added (see Section 2.4), must be added to a distributed system before the value of workflow-centric tracing can be realized. But, doing so can be challenging. Section 4.1 describes our experiences with adding propagational trace points and methods to mitigate the effort.

2.4 Additional tracing components

Value-added trace points: These trace points are optional and the choice of what trace points to include depends on the management task(s) for which the tracing infrastructure will be used. For example, distributed profiling requires value-added trace points within individual functions so that function latencies can be recorded. Trace-point records of value-added trace points are either written to disk (for out-of-band execution) or carried as metadata (for in-band execution). Section 4.2 describes our experiences with adding them.

Overhead-reduction mechanism: To make workflow-centric tracing practical, techniques must be used to reduce its overhead. For example, using overhead reduction, Dapper incurs less than a 1% overhead, allowing it to be used in production. Section 5 further describes methods to limit overhead and scenarios that will result in inflated overheads.

Storage & reconstruction component: This component is relevant only for out-of-band execution. The storage component asynchronously persists trace-point records. The trace re-construction code joins trace-point records using the metadata embedded in them.

3 Preserving causal relationships

Since the goal of workflow-centric tracing is to identify and preserve the workflow of causally-related events, the ideal tracing infrastructure would preserve all true causal relationships, and only those. For example, it would preserve the workflow of servicing individual requests and background activities, read-after-write accesses to memory, caches, files, and registers, data provenance, inter-request causal relationships due to resource contention or built-up state, and so on.

Unfortunately, it is hard to know what activities are truly causally related. So, tracing infrastructures resort to preserving

Lamport's happens-before relation (→) instead. It states that if *a* and *b* are events and *a* → *b*, then *a may have influenced b*, and thus, *b* might be causally dependent on *a* [31]. But, this relation is only an approximation of true causality: it can be both too indiscriminate and incomplete at the same time. It can be incomplete because it is impossible to know all channels of influence, which can be outside of the system [10]. It can be too indiscriminate because it captures irrelevant causality, as *may have influenced* does not mean *has influenced*.

Tracing infrastructures limit indiscriminateness by using knowledge of the system being traced and the environment to capture only the *slices* (i.e., cuts) of the general happens-before graph that are most likely to contain true causal relationships. First, most tracing infrastructures make assumptions about boundaries of influence among events. For example, by assuming a memory-protection model, the tracing infrastructure may exclude happens-before edges between activities in different processes, or even between different activities in a single-threaded event-based system (see Section 4.1 for mechanisms by which spurious edges are removed). Second, they may require developers to explicitly add trace points in areas of the distributed system's software they deem important and only track relationships between those trace points [9, 18, 19, 33, 34, 41, 44, 45, 51].

Different slices are useful for different management tasks, but preserving all of them would incur too much overhead (even the most efficient software taint-tracking mechanisms yield a 2x to 8x slowdown [28]). As such, tracing infrastructures work to preserve only the slices that are most useful for how their outputs will be used.

The rest of this section describes slices that we have found useful and describes which of them existing tracing implementations likely used. Table 2 illustrates the basic slices that are most suited for workflow-centric tracing's key management tasks. To our knowledge, none of the existing literature on workflow-centric tracing explicitly considers this critical design axis. As such, the slices we associate with existing tracing implementations that we did not develop or use is a best guess based on what we could glean from relevant literature.

3.1 Intra-request slices: basic options

One of the most fundamental decisions developers face when developing a tracing infrastructure involves choosing a slice of the happens-before graph that defines the workflow of a single request. We have observed that there are two basic options, which differ in the treatment of latent work—e.g., data left in a write-back cache that must be sent to disk eventually. Specifically, latent work can be assigned to either the workflow of the request that originally submitted it or to the workflow of the request that triggers its execution. These options, the *submitter preserving* slice and *trigger preserving* slice, have different tradeoffs and are described below.

The submitter-preserving slice: Preserving this slice means that individual workflows will show causality between the original submitter of a request and work done to process it through every component of the system. Latent work is attributed to the original submitter even if it is executed on the critical path of a different request. This slice is most useful for resource attribution, since this usage mode requires tying the work done at a component several levels deep in the system to the client, workload, or request responsible for originally submitting it. Retro [33], the original version of Stardust [51], Quanto [17], and Whodunit [8] preserve this slice of causality.

Type	Management task	Slice	Preserve structure
Perf.	Diagnosing anomalous workflows	Trigger	Y
	Diagnosing workflows w/steady-state problems	"	"
	Distributed profiling	Either	N
	Meeting SLOs	Trigger	Yes
	Resource attribution	Submitter	"
Mult.	Dynamic monitoring	Depends	Depends

Table 2: Intra-request causality slices best suited for various tasks. The slices preserved for dynamic monitoring depend on whether it will be used for performance-related tasks or resource attribution.

The two leftmost diagrams in Figure 3 show submitter-preserving workflows for two WRITE requests in a distributed storage system. Request one writes data to the system's cache and immediately replies. Sometime later, request two enters the system and must evict request one's data to place its data in the cache. To preserve submitter causality, the tracing infrastructure attributes the work done for the eviction to request one, not request two. Request two's workflow only shows the latency of the eviction. Note that the tracing infrastructure would attribute work the same way if request two were a background cleaner thread instead of a client request that causes an on-demand eviction.

The trigger-preserving slice: Preserving this slice means that individual workflows will show all work that must be performed to process a request before a response is sent to the client. Other requests or clients' latent work will be attributed to the request if it occurs on the request's critical path. Since it always shows all work done on requests' critical paths, this slice must be preserved for most performance-related tasks as it provides guidance about *why* certain requests are slow.

Switching from preserving submitter causality to preserving trigger causality was perhaps the most important change we made to the original version of Stardust [51] (useful for resource attribution) to make it useful for identifying and diagnosing problematic workflows [44]. Retro [33], Pivot Tracing [34] and X-Trace [18] preserve this slice of causality, as do many other tracing implementations implicitly [9, 29, 41, 45].

The two rightmost diagrams in Figure 3 show the same two requests as in the submitter-preserving example, with trigger causality preserved instead. In this case, the tracing

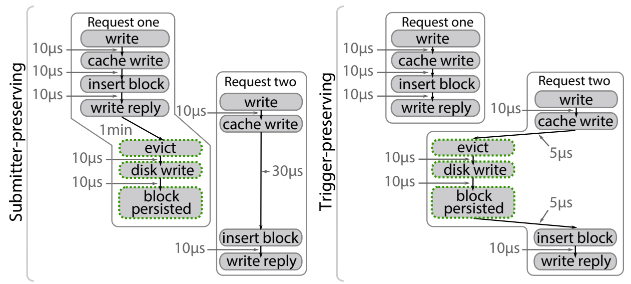

Figure 3: Differences between how latent work is attributed when preserving submitter causality vs. trigger causality.

infrastructure attributes the work done to evict request one's data to request two because it occurs in request two's critical path, helping diagnosis teams understand why request two's latency is high (i.e., that it performed an on-demand eviction).

3.2 Intra-request slices: structure

For both submitter-preserving causality and trigger-preserving causality, preserving workflow structure—concurrent behavior, forks, and joins—is optional. It must be preserved for most performance-related tasks to identify problems due to excessive parallelism, too little parallelism, and excessive waiting at synchronization points. It also enables critical paths to be easily identified in the face of concurrency. Distributed profiling is the only performance-related task that does not require preserving workflow structure as only the order of causally-related events (e.g., backtraces) need to be preserved to distinguish how functions are invoked.

The original version of X-Trace [19] used trees to model causal relationships and so could not preserve joins. The original version of Stardust [51] used DAGs, but did not instrument joins. To become more useful for diagnosis tasks, in their revised versions [18, 44], X-Trace evolved to use DAGs and both evolved to include join instrumentation APIs.

3.3 Inter-request slice options

In addition to choosing what slices to preserve to define the workflow of a single request, developers may want to preserve causal relationships between requests as well. For example, preserving *trigger and submitter causality*, would allow tracing infrastructures to answer questions, such as, "who was responsible for evicting this client's cached data?" Retro [33] preserves both of these slices because it serves two functions: guaranteeing fairness, which requires accurate resource attribution, and meeting SLOs, which requires knowing why requests are slow. By preserving *the lock-contention-preserving slice*, tracing infrastructures could identify which requests compete for generic shared resources.

4 Adding trace points

Instrumentation, in the form of trace points embedded throughout the source code for the distributed system, is a critical component of workflow-centric tracing. However, correct instrumentation is often subtle, and developers spend significant amounts their valuable time adding it before the full benefit of the tracing infrastructure can be realized. Despite the well-documented benefits of tracing, the amount of up-front effort required to instrument systems is the most significant barrier to tracing adoption today [14, 15].

Organizations whose distributed systems' components exhibit some amount of homogeneity find it easier to adopt tracing than those whose systems are very heterogeneous. Examples from systems with which we have experience and which we have observed in the literature are listed below.

First, homogeneity can be enforced through a common programming framework that is used to develop all of the distributed systems' components, such as Mace [29]. This framework can also be written (or modified) to automatically insert trace points. Second, pattern-matching can be used to automatically insert code (e.g., trace points) at arbitrary locations that match pre-defined search criteria. Similarly, code can be dynamically inserted at arbitrary locations of the distributed system. The former is provided by aspect-oriented programming [24] and the latter by Windows Hotpatching [54]. Retro [33], Pivot Tracing [34], and Facebook [16] use similar features in Java applications to add trace points with minimal developer effort. Third, common libraries, such as RPC libraries, can be instrumented once to enable tracing across all of the components of the system that share them.

At LightStep [32], we have observed that organizations' systems infrastructures may exhibit some homogeneity (most commonly in the form of libraries that are shared among a distributed systems' components). That said, extreme heterogeneity is common. For example, production systems are usually written in a variety of programming languages and share libraries only among subsets of components. This heterogeneity stems from a need to accommodate components of varying age, as well as the de-centralized nature of most large-scale organizations, which results in different teams being responsible for different components and making different design decisions for them.

Regardless of whether or not homogeneity can be leveraged, organizations' developers add trace points in similar ways. Our conversations with customers at LightStep indicate that trace points are often added on a component-by-component basis instead of in one set of sweeping changes. This is because a desire to add tracing is often initiated by individual teams that would like to understand the workflow of requests entering their components. Since workflow-centric tracing is rarely interesting when only a single component is involved, the team adding trace points involves adjacent components' developers to add trace points in those services as well. When developers are focused on end-to-end latency, they begin by instrumenting near the top of their stack (e.g., a mobile app or web client) and proceed downward along the critical path of the most important workflows. When developers are interested in the context of errors in backend services, they will start with the affected service and work up or down into adjacent services:

up when the circumstances of the error are unclear, and down when the root cause of the error is unclear.

The rest of this section describes various approaches and tradeoffs to adding trace points. Note that trace points cannot be added to *black-box* components for which source code is not available. As such, workflow-centric traces that involve block-box components can only show the clients' calls to them and replies received from them.

4.1 Trace points for metadata propagation

Propagational trace points are needed in order to propagate metadata across boundaries (e.g., process, network) to preserve the desired causality slices. They represent the bare minimum instrumentation needed for workflow-centric tracing. They are often extremely challenging to add correctly because doing so requires developers to be intimately familiar with the (possibly many) design patterns used by the distributed system being instrumented. For example, developers must be aware of which components use synchronous thread-based processing, asynchronous thread-based processing, event-based processing, or closures that run on arbitrary threads. In our experiences developing both versions of Stardust [44, 51], we found that adding these trace points always required a fair amount of trial and error and iteration. At LightStep, customers have approached us about odd-looking workflows only to find that they are a result of improper metadata propagation.

Table 3a describes tradeoffs to various approaches for adding propagational trace points. The choices that can be used and the amount of effort required to add trace points depends on the homogeneity of the system. In some homogeneous systems, metadata can be propagated for free. For example, systems written using unified programming frameworks, which constrain design patterns to ones to ones the compiler knows about, can leverage the compiler to automatically add propagational trace points [29]. However, some developer effort may still be needed to preserve the desired causality slice(s), not the ones dictated by the compiler.

Similarly, homogeneous systems written in languages that support aspect-oriented extensions [24] can use its pattern matching functionality to find code locations that use common design patterns. These can be automatically instrumented with propagational trace points before runtime [16, 33, 34]. In the Java-based systems we instrumented using Retro [33] and Pivot Tracing [34] (i.e., MapReduce, HDFS, YARN, HBase, ZooKeeper, Spark, and Tez), we only needed to modify 50–300 lines of code per system to manually add propagational trace points in areas of the system that could not be pattern matched. This was because they used uncommon patterns.

In moderately homogeneous systems, a good starting point is to manually embed trace points within commonly-used libraries (e.g., RPC libraries) if available so that the effort of adding them is incurred only once. System calls (e.g., forks, which start concurrent activity, and joins, which synchronize them) can be encapsulated by wrappers that add the necessary trace points. In extremely heterogeneous systems, purely custom (i.e., manually added) instrumentation may be required, but should be avoided if at all possible.

A promising method that avoids developer effort in finding and instrumenting complex design patterns involves learning them directly from partially-ordered workflows [12, 35]. Though this approach has the potential to completely eliminate developer effort in heterogeneous (or homogeneous) systems, the learned models learned may have a short shelf life due to the fast-changing nature of production code.

4.2 Value-added trace points

Value-added trace points decorate workflows with information relevant to various management tasks. They are optional. Examples include those embedded within individual functions to allow for fine-grained performance diagnosis or profiling and those embedded within various resource centers (e.g., CPU, disk) to enable resource attribution. Adding value-added trace points is often less challenging than adding propagational trace points. This is because adding them incorrectly does not affect the basic correctness of generated workflows (i.e., what activity is deemed causally related and workflow structure). Table 3b describes the tradeoffs among approaches to adding value-added instrumentation. Once again, usable options depend upon the homogeneity of the system.

Dynamic instrumentation, which allows trace points to be added at near arbitrary locations during runtime, is the most powerful approach, as it would enable a wide range of management tasks without the need for the tracing infrastructure to be manually modified. However, it requires homogeneous systems that are capable of this feature (e.g., those that support Hotpatching [54]). Programming frameworks can add fine-grained trace points during compile time.

In moderately homogeneous systems, the most widely ap-

Applicability	**Added via**	**Allows re-use**	**Avoids dev. effort**
Hom.	Prog. framework	✓	—
”	Auto pattern match	✓	—
Moderately Hom.	Libraries	✓	
Extremely Het.	Custom inst.		
*	Big-data analyses	—	✓

(a) Propagational trace points.

Applicability	**Added via**	**Allows re-use**	**Avoids dev. effort**
Hom.	Dynamic	✓	✓
”	Prog. framework	✓	✓
Moderately Hom.	Existing logs	✓	—
Extremely Het.	Custom inst.		

(b) Value-added trace points.

Table 3: Tradeoffs between adding propagational and value-added trace points. The mark (✓) means the corresponding column's goal is met. The mark (—) means that it is somewhat satisfied. A blank space indicates that it is not met. The mark * is a wildcard.

plicable method is to adapt existing machine-centric logging infrastructures to provide value-added trace points. However, at LightStep, we have observed that production logging infrastructures typically capture information at a coarser granularity than workflow-centric infrastructures. For example, at Google, workflow-centric traces for Bigtable compaction and streaming queries are far more detailed than the logs that are captured for them [40]. We postulate this is because separating value-added trace points by workflows increases the signal-to-noise ratio of the generated data compared to logs, allowing more trace points to be added.

5 Limiting overhead

Tracing infrastructures increase CPU, network, memory, and disk usage. CPU usage increases because metadata and trace-point records must be serialized and de-serialized (e.g., for sending metadata with RPCs or persisting records to disk) and because of the memory copies needed to propagate metadata across boundaries. Over-the-wire message sizes increase as a result of adding metadata to network messages (e.g., RPCs). Memory usage increases with metadata size. Disk usage increases if trace-point records must be persisted to storage.

Out-of-band execution and in-band execution of managements tasks affect resource usage differently and, as such, require different techniques to reduce overhead. All of them try to limit the number of trace-point records that must be considered by the tracing infrastructure (i.e., persisted to disk or propagated as metadata). While developing the revised version of Stardust [44], we learned that a very common feature in distributed systems—aggregation of work—can curtail the efficacy of some of these techniques and drastically inflate overheads when submitter causality is preserved. Aggregation is commonly used to amortize the cost of using various resources in a system by combining individual pieces of work into a larger set that can be operated on as a unit. For example, individual writes to disk are often aggregated into a larger set to amortize the cost of disk accesses. Similarly, network packets are often aggregated into a single larger packet to reduce the overhead of each network transmission.

The rest of this section describes why aggregation can stymie overhead-reduction techniques. It also describes common techniques for limiting overhead for out-of-band and in-band execution and which ones are affected by aggregation.

5.1 Aggregation & submitter causality

Figure 4 illustrates why aggregation can severely limit the ability of tracing infrastructures to reduce the number of trace points that must be considered. It shows a simple example of aggregating cached data to amortize the cost of a disk write. In this example, a number of requests have written data asynchronously to the distributed system, all of which are stored in cache as latent work. At some point in time, another request (shown as the "Trigger request") enters the system and must perform an on-demand eviction of a cached item in order to insert the new request's data (this could also be a cleaner thread). This request that triggers the eviction aggregates many other cached items and evicts them at the same time to amortize the cost of the necessary disk access.

When preserving submitter causality, all of the work done to evict the aggregated items (shown as trace points with dotted outlines) must be attributed to each of the original submitters. If the overhead-reduction mechanism has already committed to preserving at least one of those original submitters' workflows (shown with circled trace points at their top), all trace points below the aggregation point must be considered by the tracing infrastructure. Since many distributed systems contain many levels of aggregation, the effect of aggregation in limiting what trace points need to be considered can compound quickly. In many systems, aggregation will result in tracing infrastructures having to consider almost all trace points deep in the system. In contrast, trigger causality is not suspect to these effects as trace points below the aggregation point need only be considered if the overhead-reduction technique has committed to preserving the workflow that triggers the eviction.

5.2 Out-of-band execution

Tracing infrastructures that execute management tasks out-of-band primarily affect CPU, memory, and disk usage. Network overhead is typically not a concern because metadata need only increase RPC sizes by the size of a logical clock (as small as 32 or 64 bits). To a first order approximation, the overhead is a result of the work that must be done to persist trace-point records. As such, these tracing infrastructures use coherent sampling techniques to limit the number of trace-point records they must persist to disk. *Coherent* sampling means that either all or none of a workflow's trace points are persisted. For example, Dapper incurs a 1.5% throughput and 16% response time overhead when sampling all trace points. But, when sampling is used to persist just 0.01% of all trace points, the slowdown in response times is reduced to 0.20% and in throughput to 0.06% [45]. There are three options for deciding what trace points to sample: head-based sampling, tail-based sampling, and hybrid sampling.

Head-based coherent sampling: With this method, a random sampling decision is made for entire workflows at their start (i.e., when requests enter the system) and metadata is propagated along with workflows indicating whether to persist their trace points. The percentage of workflows randomly sam-

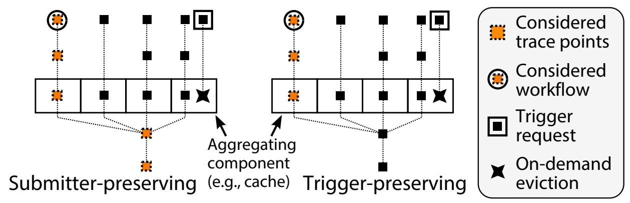

Figure 4: Trace points that must be considered as a result of preserving different causality slices.

pled is controlled by setting the *workflow-sampling percentage*. When used in tracing infrastructures that preserve trigger causality, the workflow-sampling percentage and the trace-point-sampling percentage (i.e., the percentage of trace points executed that are sampled by the tracing infrastructure) will be the same. Due to its simplicity, head-based coherent sampling is used by many existing tracing implementations [18, 44, 45].

Because of aggregation, head-based coherent sampling will result in drastically inflated overheads for tracing infrastructures that preserve submitter causality. In such scenarios, the effective trace-point sampling percentage will be much higher than the workflow-sampling percentage set by developers. This is because trace points below the aggregation point must be sampled if any workflow whose data is aggregated is sampled. For example, if head-based sampling is used to sample only 0.1% of workflows, the probability of sampling an individual trace point will also be 0.1% before any aggregations. However, after aggregating 32 items, this probability will increase to 3.2% and after two such levels of aggregation, the trace-point-sampling percentage will increase to 65%.

When developing the revised version of Stardust [44], we learned of how head-based sampling can inflate overhead the hard way. Head-based sampling was the first feature we added to the original Stardust [51], which previously did not use sampling and preserved submitter causality. But, at the time, we did not know about causality slices or how they interact with different sampling techniques. So, when we applied the sampling-enabled Stardust to our test distributed system, Ursa Minor [1], we were very confused as to why the tracing overheads did not decrease. Of course, the root cause was that Ursa Minor contained a cache near the entry point to the system, which aggregated 32 items at a time. We were using a sampling rate of 10%, meaning that 97% all trace points executed after this aggregation were always sampled.

Tail-based sampling: This method is similar to the previous one, except that the workflow-sampling decision is made at the end of workflows, instead of at their start. Doing so allows for more intelligent sampling that persists only workflows that are important to the relevant management task. Most importantly, anomalies can be explicitly preserved, whereas most of them would be lost by the indiscriminateness of head-based sampling. Tail-based sampling does not inflate the trace-point sampling percentage after aggregation events because it does not commit to a sampling decision upfront. However, it can incur high memory overheads because it must cache all trace points for concurrent workflows until they complete. Within Ursa Minor [1], we observed that the largest workflows can contained around 500 trace points and were several hundred kilobytes in size. Of course, large distributed systems can service tens to hundreds of thousands of concurrent requests.

Hybrid sampling: With this method, head-based sampling is nominally used, but records of all unsampled trace points are also cached for a pre-set (small) amount of time. This allows the infrastructure to backtrack to collect trace-point records for workflows that experience correctness anomalies, as they will appear immediately problematic. However, it is not sufficient for performance anomalies, as their response times can have a long tail.

In addition to deciding how to sample workflows, developers must decide how many of them to sample. Many infrastructures choose to randomly sample a small, set percentage—often between 0.01% and 10%—of workflows [18, 44, 45]. However, this approach will capture only a few workflows for small workloads, limiting its use for them. An alternate approach is an adaptive scheme, in which the tracing infrastructure dynamically adjusts the sampling percentage to always capture a set rate of workflows (e.g., 500 workflows/second).

5.3 In-band execution

Tracing infrastructures that execute management tasks in-band primarily increase CPU, memory, and network usage. Disk is not a concern because these infrastructures do not persist trace-point records. To a first-order approximation, overhead is a function of the size of the metadata (logical clocks and trace-point records) that must be carried as metadata. As such, developers' primary means of reducing overhead for these infrastructures involves limiting metadata size.

One common-sense method for limiting metadata sizes is to take extreme care to include as metadata only the trace-point records most relevant to the management task at hand. For example, Google's Census [22] limits both the size and number of fields that can be propagated with a request, with a worst-case upper limit of 64kB. By design, fields may be readily discarded by components in order to keep metadata within allowable size limits. Pivot Tracing [34] leverages dynamic instrumentation to include only trace-point records relevant to user-specified management tasks as metadata.

In addition to careful inclusion, tracing infrastructures that execute tasks in-band use lossy compression and partial execution to further reduce the size of metadata. Head-based sampling could also be used.

Lossy compression: This involves compressing the trace-point records carried within metadata to make them smaller with (perhaps) some loss in fidelity. For example, Whodunit compresses nested loops so that only one iteration of the outer-most loop is carried as metadata [8].

Partial execution: This involves greedily executing tasks as soon as some minimum amount of data is available. Doing so can reduce the amount of information that must be carried as metadata. For example, Pivot Tracing [34] evaluates functions, such as sum and multiply, within metadata as soon as possible in the workflow, so that only the result needs to be carried.

Regardless of which of the above techniques is used to limit overhead, infrastructures that execute tasks in-band will suffer from inflated overheads due to aggregation when preserving submitter causality. This is because trace-point records relevant to all submitters' workflows must be included as metadata after the aggregation point. One technique that can be used to limit metadata inflation for some management tasks is to map all submitters' metadata to a single "aggregated ID" and only carry this ID as metadata after aggregation. A separate mapping must be kept between the aggregated ID and submitters' original metadata. Since more trace points may be added as metadata after the aggregation event, partial executions must operate both on the external map and post-aggregation metadata.

6 Putting it all together

Based on the tradeoffs described in previous sections and our experiences, this section lists the design choices we recommend for workflow-centric tracing's key management tasks. Our choices represent the minimum necessary for a tracing infrastructure to be suited for a given task. Developers should feel free to exceed our recommendations if they see fit. Also, different choices may be needed to accommodate (or leverage) custom features of the distributed systems to which a tracing implementation is to be applied. To provide guidance for such scenarios, this section also shows previous implementations' choices and speculates on reasons for divergences.

6.1 Suggested choices

The italicized rows of Table 4 show our suggested design choices for the key management tasks. The table does not include the causal model (e.g., trees or DAGs) because it can be mostly be inferred from other design axes.

In general, we recommend that tracing infrastructures use out-of-band execution whenever tasks require full workflows to be presented. This avoids drastically inflating RPC sizes. We recommend tracing infrastructures use in-band execution for tasks that do not require presenting full workflows, as various overhead-reduction techniques can be used to keep metadata sizes small (e.g., lossy compression). Doing so means management tasks are more easily executed online (i.e., close to real time) and reduces the amount of data needed to execute management tasks. In all of the cases in which we recommend in-band execution, out-of-band execution could be used instead. Doing so would allow workflow-centric tracing to be used with distributed systems that are extremely sensitive to inflated metadata sizes.

For instrumentation, we conservatively recommend adding trace points within libraries and re-using logging infrastructures. Of the choices that mitigate developer effort, these two are likely to be the most widely applicable.

Identifying anomalous workflows: This task involves identifying rare workflows that are extremely different from others so that diagnosis teams can analyze them. Since entire workflows must be preserved, we recommend out-of-band execution. To help diagnose performance-related problems, trigger causality must be preserved as they show all work done on requests' critical paths. Workflow structure (i.e., forks, concurrency, and joins) should also be preserved to identify problems that result from excessive concurrency, insufficient concurrency, or excessive waiting for one of many concurrent operations to finish. Tail-based sampling should be used so that anomalies are not lost. In addition to the minimally-necessary choices listed above, developers may find it useful to preserve submitter causality and the contention-preserving slice. Doing so would increase overhead, but would provide insight into whether observed anomalies are a result of poor interactions with another client or request.

Identifying workflows w/steady-state problems: This task involves identifying and presenting workflows that negatively impact the mean or median of some distribution to diagnosis teams. Such problematic workflows are often performance related. Design choices for it are similar to anomaly detection, except that head-based sampling can be used, since, even with low sampling rates, it is unlikely that problems will go unnoticed. Note that if developers choose to additionally preserve submitter causality or the contention-preserving slice, tail-based sampling must be used. This is necessary to avoid inflated overheads due to aggregation and to guarantee that poorly interacting workflows will be sampled.

Distributed profiling: This task involves identifying slow functions and binning function execution time by context (i.e., by unique calling stack). Since call stacks can be carried compactly in metadata using lossy compression techniques [8], we recommend in-band execution. Either submitter or trigger causality can be preserved as both will create unique call stacks, but we recommend the latter to avoid inflating metadata sizes due to aggregation. Call stacks do not preserve workflow structure. We conservatively recommend custom instrumentation for value-added trace points because existing logs may not exhibit the per-function resolution needed for this task.

Meeting SLOs: This task involves adjusting workflows' resource allocations so that jobs meet SLOs. Trigger causality is necessary to identify work done on requests' critical paths. This provides guidance about what resource allocations to modify to increase their performance. Workflow structure should be preserved so that critical paths can be identified in the face of concurrency. We recommend in-band execution since only critical paths (not full workflows) need to be considered. Partial execution can be used to prune all concurrent paths but the critical one at join points.

Resource attribution: This task involves attributing work done at arbitrary levels of the system to the original submitter. It requires carrying client IDs or request IDs as metadata, which can be done in-band, and preserving submitter causality. Aggregation might result in extremely large metadata sizes because many IDs might need to be carried as metadata. As such, we recommend replacing individual client or request IDs with an "aggregate ID" after aggregation events. A DAG must be used as the causality model to capture aggregation events when preserving submitter causality.

Dynamic monitoring: This task involves monitoring some activity only if that activity is causally related to pre-conditions at other components. Both the activity to monitor and the pre-conditions are dynamically chosen at runtime. This task is best

	Design axes				
Use case/**name**	**Execution support**	**Causality slices**	**Structure**	**Trace points: P / V**	**Overhead reduction**
Identifying anomalous workflows	*Out-of-band*	*Trigger*	*Yes*	*Libs / Logs*	*Sampling (T)*
Pip [41]	”	”	”	” / Cust.	None
Pinpoint [9]	”	”	No	” / None	”
Mace [29]	”	”	”	PF / PF	”
Identifying workflows w/steady-state problems	*Out-of-band*	*Trigger*	*Yes*	*Libs / Logs*	*Sampling (H)*
Stardust‡ [44]	”	”	”	” / Cust.	”
X-Trace‡ [18]	”	”	”	” / ”	”
Dapper [45]	”	”	Forks / Conc.	” / ”	”
Pip [41]	”	”	Yes	” / ”	None
X-Trace [19]	”	Trigger & TCP layers	Forks / Conc.	” / ”	”
Pinpoint [9]	”	Trigger	No	” / None	”
Mace [29]	”	”	”	PF / PF	”
Distributed profiling	*In-band*	*Trigger*	*No*	*Libs / Cust.*	*Lossy comp.*
Whodunit [8]	”	Submitter	”	Taint tracking	”
Dapper [45]	Out-of-band	Trigger	Forks / Conc.	Libs / Cust.	Sampling (H)
Meeting SLOs	*In-band*	*Trigger*	*Yes*	*Libs / Cust.*	*Partial exec. (prune)*
Retro [33]	”	Trigger + Submitter	No	Auto. / Dyn.	Aggregate IDs
Resource attribution	*In-band*	*Submitter*	*No*	*Libs / Cust.*	*Aggregate IDs*
Retro [33]	”	Trigger + Submitter	”	Auto. / Dyn.	”
Quanto [17]	”	Submitter	”	Cust. / N/A	None
Stardust [51]	*Out-of-band*	”	*Forks / Conc.*	Libs / Cust.	”
Dynamic monitoring	*In-band*	*Depends*	*Depends*	*Libs / Dyn.*	*Partial exec. (e.g., sum)*
Pivot Tracing [34]	”	Trigger	Yes	Auto / ”	”

Table 4: Suggested design choices for various management tasks and choices made by existing tracing implementations. Suggested choices are shown in italics. Existing implementations' design choices are qualitatively ordered according to similarity with our suggested choices. The choices indicated for tracing infrastructures we did not develop are based on a literature survey. *Stardust*‡ and *X-Trace*‡ denote the revised versions of Stardust and X-Trace. *P* and *V* respectively denote propagational trace points and value-added ones. A ” indicates that the entry is the same as the preceding row. *PF* refers to a unified programming framework in which distributed systems' components can be written and compiled. *Auto* refers to automatically adding propagational trace points via pattern matching (e.g., as enabled by aspect-oriented programming [24]). *Dyn.* refers to dynamically inserting value-added trace points (e.g., as enabled by aspect-oriented programming or Windows Hotpatching [54]).

served by in-band execution, to execute tasks online and limit the data collected to that which is to be monitored. The choice of submitter or trigger causality depends on whether this task will be used for resource attribution or performance purposes. Dynamic instrumentation for value-added trace points allows maximum flexibility to choose arbitrary pre-conditions and activity to monitor. Existing logging infrastructures or custom instrumentation could also be used at the cost of reduced flexibility; in this case, guidance on the what instrumentation to use as pre-conditions and what to monitor could be provided as a bitmap propagated as metadata.

6.2 Existing implementations' choices

Table 4 lists how existing tracing infrastructures fit into the design axes suggested in this paper. Tracing implementations are grouped by the management task for which they are most suited (a tracing implementation may be well suited for multiple tasks). For a given management task, tracing implementations are ordered according to similarity in design choices to our suggestions. In general, implementations suited for a particular management task tend to make similar design decisions to our suggestions for that task. The rest of this section describes cases where our suggestions differ from existing implementations' choices.

Identifying anomalous workflows: We recommend preserving full workflow structure (forks, joins, and concurrency), but Pinpoint [9] and Mace [29] cannot do so because they use paths as their model for expressing causal relationships. Pinpoint does not preserve workflow structure because it is mainly concerned with correctness anomalies. We also recommend using tail-based sampling, but none of these infrastructures use any sampling techniques whatsoever. We speculate this is because they were not designed to be used in production or to support large-scale distributed systems.

Identifying workflows with steady-state problems: We suggest that workflow structure be preserved, but Dapper [45] and X-Trace [19] cannot preserve joins because they use use trees to express causal relationships. Dapper chose to use trees (a specialized model) because many of their initial use cases involved distributed systems that exhibited large amounts of concurrency with comparatively little synchronization. For broader use cases, recent work by Google and others [12, 35] focuses on learning join-point locations by comparing large volumes of traces. This allows tree-based traces to be reformatted into DAGs that show the learned join points.

Both the revised version of Stardust [44] and the revised version of X-Trace [18] were created as a result of modifying their original versions [19, 51] to be more useful for identifying workflows with steady-state-problems. Both revised versions independently converged to use the same design. We initially tried to re-use the original Stardust [51], which was designed with resource attribution in mind, but its inadequacy for diagnosis motivated the revised version. The original X-Trace was designed for diagnosis tasks, but we evolved our design choices to those listed for the revised version as a result of experiences applying X-Trace to more distributed systems [18].

Meeting SLOs: We suggest preserving trigger causality and workflow structure, and using partial execution to prune non-critical paths. Retro [33] preserves trigger and submitter causality because it can be used to both help meet SLOs and guarantee fairness. It does not preserve structure, as this was not necessary for the SLO-violation causes we considered. It uses aggregate IDs to limit overhead due to aggregation events.

Distributed profiling, resource attribution, and dynamic monitoring: Most existing implementations either meet or exceed our suggestions. We believe Whodunit [8] does not use techniques to limit overhead due to aggregation because the systems it was applied to did not have many such events. For resource attribution, we suggest in-band execution, but the original version of Stardust [51] is designed for out-of-band execution and does not sample. This mismatch occurs because Stardust was also used for generic workload modeling, which required constructing full workflows [50], and because it was not designed to support large-scale distributed systems.

We note that Pivot Tracing [34] and Retro [33] have the potential to be used for all of the tasks that do not require preserving full workflows. This is because they are used in homogeneous distributed systems that support aspect-oriented extensions, which allow many design choices to be modified easily. Specifically, they can leverage aspects' pattern matching functionality to change the causality slice that is preserved before runtime. They can also leverage aspects to dynamically change what is instrumented during runtime—e.g., resources for resource attribution or functions for distributed profiling—and what mechanism is used for overhead reduction.

In Table 4, we do not include Pivot Tracing and Retro for management tasks that would require them to be modified. Modifying Pivot Tracing and Retro to support out-of-band analyses would require additional modifications, such as inclusion of a storage & re-construction component. Similar flexibility could be provided by modifying Mace [29] and recompiling distributed systems written using it.

7 Future research avenues

We have only explored the tip of the iceberg of the ways in which tracing can inform distributed-system design and management. One promising research direction involves creating a single tracing infrastructure that can be dynamically configured to support all of the management tasks described in this paper. Pivot Tracing [34] is an initial step in this direction. This section surveys other promising research avenues.

Reducing the difficulty of instrumentation: Many real-world distributed systems are extremely heterogeneous, making instrumentation extremely challenging. To help, black-box and white-box systems must be pre-instrumented by vendors to support tracing. Standards, such as OpenTracing [38], are needed to ensure compatibility across the multiple tracing infrastructures that will undoubtedly be used in large-scale organizations. We need to explore ways to automatically convert today's prevalent machine-centric logging infrastructures into workflow-centric ones that propagate metadata.

Exploring new out-of-band analyses and dealing with scale: These avenues include exploring adaptation of constraint-based replay [11, 25] for use with workflow-centric tracing and exploring ways to semantically label, intelligently compress, automatically compare, and visualize extremely large traces. The first could help reduce the amount of trace data that needs to be collected for management tasks. The second is needed because users of tracing cannot understand large traces. Guidance can be drawn from the HPC community, which has developed sophisticated tracing and visualization tools for very homogeneous distributed systems [20, 26, 37, 47].

Pushing in-band analyses to the limit: In-band execution of tasks offers key advantages over out-of-band execution. As such, we need to fully explore the breadth of tasks that can be executed in-band, including ways to execute out-of-band tasks in-band instead without greatly inflating metadata sizes.

8 Summary

Key design decisions dictate workflow-centric tracing's utility for different management tasks. Based on our experiences developing workflow-centric tracing infrastructures, this paper identifies these design decisions and provides guidance to developers of such infrastructures.

Acknowledgements: We thank our shepherd (Alvin Cheung), Michelle Mazurek, Michael De Rosa, and Gabe Wishnie for their insights and feedback. We thank LightStep and the members and companies of the PDL Consortium (including Broadcom, Citadel, EMC, Facebook, Google, HP Labs, Hitachi, Intel, Microsoft Research, MongoDB, NetApp, Oracle, Samsung, Seagate, Tintri, Two Sigma, Uber, Veritas, and Western Digital) for their interest, insights, feedback, and support. This research was sponsored in part by two Google research awards, NSF grants #CNS-1117567 and #CNS-1452712, and by Intel via the ISTC-CC. While at Carnegie Mellon, Ilari Shafer was supported in part by a NSF Graduate Research Fellowship.

References

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. Strunk, E. Thereska, M. Wachs, and J. Wylie. Ursa Minor: versatile cluster-based storage. *FAST '05*.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. *SOSP '03*.

[3] P. S. Almeida, C. Baquero, and V. Fonte. Interval tree clocks: a logical clock for dynamic systems. *OPODIS '08*.

[4] Apache HTrace. http://htrace.incubator.apache.org/.

[5] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. *OSDI '04*.

[6] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. *OSDI '06*.

[7] Cassandra tracing. https://issues.apache.org/jira/browse/CASSANDRA-10392.

[8] A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: transactional profiling for multi-tier applications. *EuroSys '07*.

[9] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. *NSDI '04*.

[10] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. *SOSP '93*.

[11] A. Cheung, A. Solar-Lezama, and S. Madden. Partial replay of long-running applications. *FSE '11*.

[12] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: end-to-end performance analysis of large-scale internet services. *OSDI '14*.

[13] Compuware dynaTrace PurePath. http://www.compuware.com.

[14] Distributed tracing workshop meeting, October 2015. https://groups.google.com/forum/#!forum/distributed-tracing.

[15] Distributed tracing workshop meeting, July 2016. https://groups.google.com/forum/#!forum/distributed-tracing.

[16] Performance instrumentation for Android apps. https://code.facebook.com/posts/747457662026706/performance-instrumentation-for-android-apps/.

[17] R. Fonseca, P. Dutta, P. Levis, and I. Stoica. Quanto: tracking energy in networked embedded systems. *OSDI '08*.

[18] R. Fonseca, M. J. Freedman, and G. Porter. Experiences with tracing causality in networked services. *INM/WREN '10*.

[19] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: a pervasive network tracing framework. *NSDI '07*.

[20] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, **22**(6):702–719, 2010.

[21] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *SOSP '03*.

[22] Google Census. https://github.com/grpc/grpc/tree/master/src/core/ext/census.

[23] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, M. Musuvathi, Z. Zhang, and L. Zhou. Failure Recovery: when the cure is worse than the disease. *HotOS '13*.

[24] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. *AOSD '04*.

[25] J. Huang, C. Zhang, J. Dolby, J. Huang, C. Zhang, and J. Dolby. CLAP: recording local executions to reproduce concurrency failures. *PLDI '13*.

[26] K. E. Isaacs, A. Bhatle, J. Lifflander, D. Böhme, T. Gamblin, M. Schulz, B. Hamann, and P.-T. Beremer. Recovering logical structure from Charm++ event traces. *SC '15*.

[27] S. P. Kavulya, S. Daniels, K. Joshi, M. Hultunen, R. Gandhi, and P. Narasimhan. Draco: statistical diagnosis of chronic problems in distributed systems. *DSN '12*.

[28] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: practical dynamic data flow tracking for commodity systems. *VEE '12*.

[29] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. *PLDI '07*.

[30] E. Koskinen and J. Jannotti. BorderPatrol: isolating events for black-box tracing. *EuroSys '08*.

[31] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, **21**(7), 1978.

[32] LightStep. http://lightstep.com/.

[33] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. *NSDI '15*.

[34] J. Mace, R. Roelke, and R. Fonseca. Pivot Tracing: dynamic causal monitoring for distributed systems.

[35] G. Mann, M. Sandler, D. Krushevskaja, S. Guha, and E. Even-dar. Modeling the parallel execution of black-box services. *HotCloud '11*.

[36] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, **30**(7), 2004.

[37] M. Mesnier, M. Wachs, R. R. Sambasivan, J. Lopez, J. Hendricks, G. R. Ganger, and D. O'Hallaron. //Trace: Parallel trace replay with approximate causal events.

[38] OpenTracing website. http://opentracing.io/.

[39] Personal communication with Google engineers (Sambasivan), 2011.

[40] Personal conversations with Google engineers (Sigelman), 2003-2012.

[41] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. Shah, and A. Vahdat. Pip: detecting the unexpected in distributed systems. *NSDI '06*.

[42] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: black-box performance debugging for wide-area systems. *WWW '06*.

[43] R. R. Sambasivan, I. Shafer, M. L. Mazurek, and G. R. Ganger. Visualizing request-flow comparison to aid performance diagnosis in distributed systems. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Information Visualization 2013)*, **19**(12), 2013.

[44] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. *NSDI '11*.

[45] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. *Dapper, a large-scale distributed systems tracing infrastructure*. dapper-2010-1. 2010.

[46] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang. vPath: precise discovery of request processing paths from black-box observations of thread and network activities. *USENIX '09*.

[47] N. R. Tallent, J. Mellor-Crummey, M. Franco, R. Landrum, and L. Adhianto. Scalable fine-grained call path tracing. *ISC '11*.

[48] J. Tan, S. P. Kavulya, R. Gandhi, and P. Narasimhan. Visual, log-based causal tracing for performance debugging of mapreduce systems. *ICDCS '10*.

[49] The `strace` system call tracer. `http://sourceforge.net/projects/strace/`.

[50] E. Thereska and G. R. Ganger. Ironmodel: robust performance models in the wild. *SIGMETRICS '08*.

[51] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: tracking activity in a distributed storage system. *SIGMETRICS '06/Performance '06*.

[52] Tracelytics. `http://www.tracelytics.com`.

[53] M. Wachs, L. Xu, A. Kanevsky, and G. R. Ganger. Exertion-based billing for cloud storage access. *HotCloud '11*.

[54] Introduction to Hotpatching. `https://technet.microsoft.com/en-us/library/cc781109(v=ws.10).aspx`.

[55] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Detecting large-scale system problems by mining console logs. *SOSP '09*.

[56] Zipkin. `http://zipkin.io/`.