Performance Analysis of Cloud Applications

Dan Ardelean, Amer Diwan, and Chandra Erdman, *Google*

https://www.usenix.org/conference/nsdi18/presentation/ardelean

This paper is included in the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18).
April 9–11, 2018 • Renton, WA, USA
ISBN 978-1-939133-01-4

Performance analysis of cloud applications

Dan Ardelean *Google*   Amer Diwan *Google*   Chandra Erdman *Google*

Abstract

Many popular cloud applications are large-scale distributed systems with each request involving tens to thousands of RPCs and large code bases. Because of their scale, performance optimizations without actionable supporting data are likely to be ineffective: they will add complexity to an already complex system often without chance of a benefit. This paper describes the challenges in collecting actionable data for Gmail, a service with more than 1 billion active accounts.

Using production data from Gmail we show that both the load and the nature of the load changes continuously. This makes Gmail performance difficult to model with a synthetic test and difficult to analyze in production. We describe two techniques for collecting actionable data from a production system. First, coordinated bursty tracing allows us to capture bursts of events across all layers of our stack simultaneously. Second, vertical context injection enables us combine high-level events with low-level events in a holistic trace without requiring us to explicitly propagate this information across our software stack.

1 Introduction

Large cloud applications, such as Facebook and Gmail, serve over a billion active users [4, 5]. Performance of these applications is critical: their latency affects user satisfaction and engagement with the application [2, 26] and their resource usage determines the cost of running the application. This paper shows that understanding and improving their resource usage and latency is difficult because their continuously varying load continually changes the performance characteristics of these applications.

Prior work has shown that as the user base changes, the load on cloud applications (often measured as queries-per-second or QPS) also changes [28]. We show, using data from Gmail, that the biggest challenges in analyzing performance come not from changing QPS but in changing load *mixture*. Specifically, we show that the load on a cloud application is a continually varying mixture of diverse loads, some generated by users and some generated by the system itself (e.g., essential maintenance tasks). Even if we consider only load generated by users, there is significant variation in load generated by different users; e.g., some user mailboxes are four orders of magnitude larger than others and operations on larger mailboxes are fundamentally more expensive than those on smaller mailboxes.

This time-varying mixture of load on our system has two implications for performance analysis. First, to determine the effect of a code change on performance, we must collect and analyze data from many points in time and thus many different mixtures of load; any single point in time gives us data only for one mixture of load. Second, to reproduce a performance problem we may need to reproduce the combination of load that led to the performance problem in the first place. While sometimes we can do this in a synthetic test, often times we need to collect and analyze data from a system serving real users.

Doing experiments in a system serving real users is challenging for two reasons. First, since we do not control the load that real users generate, we need to do each experiment in a system serving a large (tens of millions users) random sample of users to get statistically significant results. Second, since experiments in a system serving real users is inherently risky (a mistake can negatively impact users) we use statistics whenever possible to predict the likely outcome of an experiment before actually undertaking the experiment. We show that this is not without its pitfalls: sometimes the distribution of the data (and thus the appropriate statistical method) is not obvious.

To collect rich performance data from a system serving real users we have developed two techniques.

First, *coordinated bursty tracing* collects coordinated

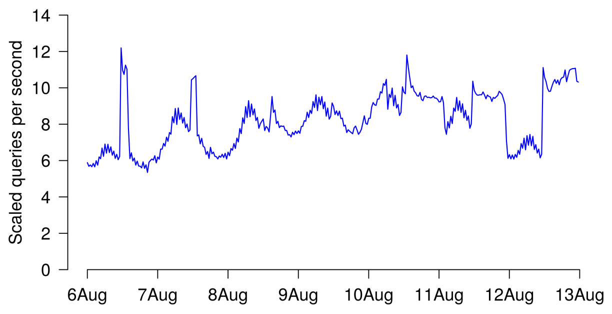


Figure 1: Continuously changing queries per second

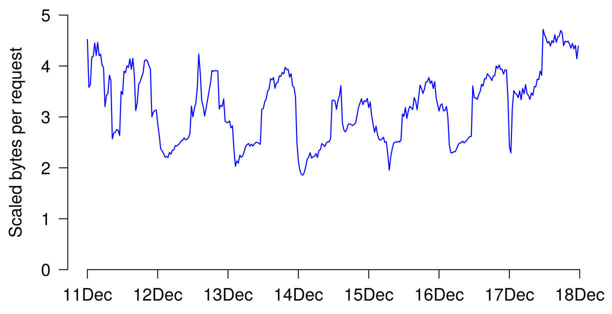


Figure 2: Continuously changing response size

bursts of traces across all software layers without requiring explicit coordination. Unlike traditional sampling or bursty approaches which rely on explicitly maintained counters [19, 6] or propagation of sampling decisions [27, 22], coordinated bursty tracing uses time to coordinate the start and end of bursts. Since all layers collect their bursts at the same time (clock drift has not been a problem in practice), we can reason across the entire stack of our application rather than just a single layer. By collecting many bursts we get a random sampling of the mix of operations which enables us to derive valid conclusions from our performance investigations.

Second, since interactions between software layers are responsible for many performance problems, we need to be able to connect trace events at one layer with events at another. *Vertical context injection* solves this problem by making a stylized sequence of innocuous system calls at each high-level event of interest. These system calls insert the system call events into the kernel trace which we can analyze to produce a trace that interleaves both high and low-level events. Unlike prior work (e.g., [27] or [15]) our approach does not require explicit propagation of a trace context through the layers of the software stack.

To illustrate the above points, this paper presents data from Gmail, a popular email application from Google. However, the authors have used the techniques for many other applications at Google, particularly Google Drive; the lessons in this paper apply equally well to those other applications.

2 Our challenge: constantly varying load

The primary challenge in performance analysis of cloud applications stems from their constantly varying load. Figure 1 shows *scaled* queries per second (QPS) across thousands of processes serving tens of millions of users over the course of a week for one deployment of Gmail. By "scaled" we mean that we have multiplied the actual numbers by a constant to protect Google's proprietary information; since we have multiplied each point by the

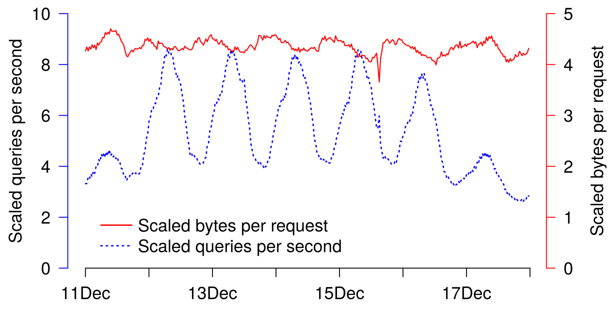


Figure 3: Continuously changing user behavior

same constant and each graph is zero based, it allows relative comparisons between points or curves on the same graph.[1] The time axes for all graphs in this paper are in US Pacific time and start on a Sunday unless the graph is for a one-off event in which case we pick the time axis most suitable for the event. We see that load on our system changes continuously: from day to day and from hour to hour by more than a factor of two.

While one expects fluctuations in QPS (e.g., there are more active users during the day than at night), one does not expect the *mix* of requests to fluctuate significantly. Figure 2 shows one characteristic of requests, the response size per request, over the course of a week. Figure 2 shows that response size per request changes over the course of the week and from hour to hour (by more than a factor of two) which indicates that the actual mix of requests to our system (and not just their count) changes continuously.

The remainder of this section explores the sources of variation in the mix of requests.

2.1 Variations in rate and mix of user visible requests (UVR)

Figure 3 gives the response size per request (upper curve) and QPS (lower curve) for "user visible requests" (UVR for short) only. By UVR we mean requests that users explicitly generate (e.g., by clicking on a message), background requests that the user's client generates (e.g., background sync by the IMAP client), and message delivery.

From Figure 3 we see that, unlike Figure 1, the QPS curve for UVR exhibits an obvious diurnal cycle; during early morning (when both North America and Europe are active) we experience the highest QPS, and the QPS gradually ramps up and down as expected. Additionally we see higher QPS on weekdays compared to weekends.

We also see that the bytes per response in Figure 3 is flatter than in Figure 2: for UVR, we see that the highest observed response size per request is approximately 20% higher than the lowest observed response size per request. In contrast, when we consider all requests (and not just UVR), the highest response size is more than 100% higher than the lowest (Figure 2).

While a 20% variation is much smaller than a 100% variation, it is still surprising: especially given that the data is aggregated over tens of millions of users, we expect that there would not be much variation in average response size over time. We have uncovered two causes for this variation.

First, the mix of mail delivery (which has a small response size) and user activity (which usually has a larger response size) varies over the course of the day which in turn affects the average response size per request. This occurs because many bulk mail senders send bursts of email at particular times of the day and those times do not necessarily correlate with activity of interactive users.

Second, the mix of user interactive requests and sync requests varies over the course of the day. For example, Figure 4 shows the scaled ratio of UVR requests from a web client to UVR requests from an IMAP client (e.g., from an iOS email application or from Microsoft Outlook). IMAP requests are mostly synchronization requests (i.e., give me all messages in this folder) while web client requests are a mix of interactive requests (i.e., user clicks on a message) and prefetch requests. Thus, IMAP requests tend to have a larger response size compared to interactive requests. We see that this ratio too exhibits a diurnal cycle indicating that over the course of the day the relative usage of different email clients changes (by more than a factor of three), and thus the response size per request also changes. This comes about

[1]All graphs in this paper use such scaling but different graphs may use different scaling to aid readability and thus absolute values are not comparable across graphs.

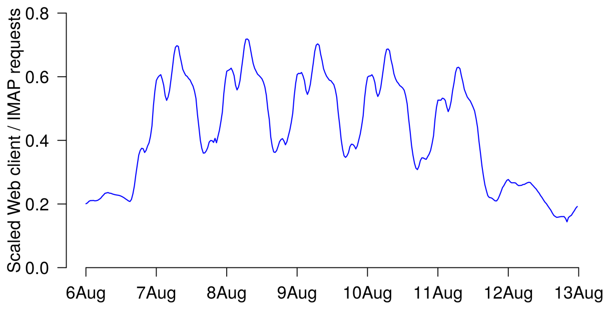


Figure 4: Scaled fraction of web client requests to IMAP requests

due to varying email client preferences; for example one user can use a dedicated Gmail application while another can use a generic IMAP-based email application on a mobile device).

In summary, even if we consider only UVR, both the queries per second and mix of requests changes hour to hour and day to day.

2.2 Variations in rate and mix of essential non-UVR work

In addition to UVR requests which directly provide service to our users, our system performs many essential tasks:

- Continuous validation of data. Because of the scale of our system, any kind of failure that *could* happen *does* actually happen. Besides software failures, we regularly encounter hardware errors (e.g., due to an undetected problem with a memory chip). To prevent these failures from causing widespread corruption of our data, we continuously check invariants of the user data. For example, if our index indicates that N messages contain a particular term, we check the messages to ensure that those and only those messages contain the term. As another example, we continuously compare replicas of each user's data to detect any divergence between the replicas.

- Software updates. Our system has many interacting components and each of them has its own update cycle. Coordinating the software updates of all components is not only impractical but undesirable: it is invaluable for each team to be able to update its components when necessary without coordination with other teams. Rather than using dynamic software updating [18] which attempts to keep each process up while updating it, we use a simpler approach: we push software updates by restarting a few processes at a time; our system automatically

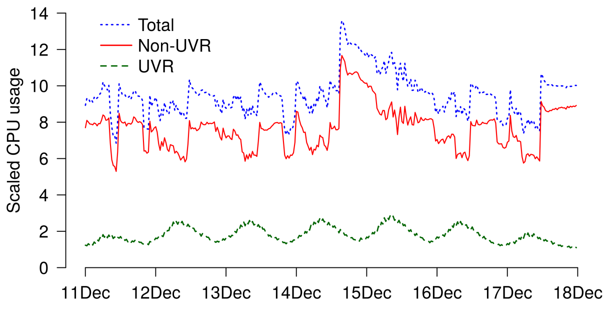


Figure 5: Scaled CPU usage of UVR and non-UVR work

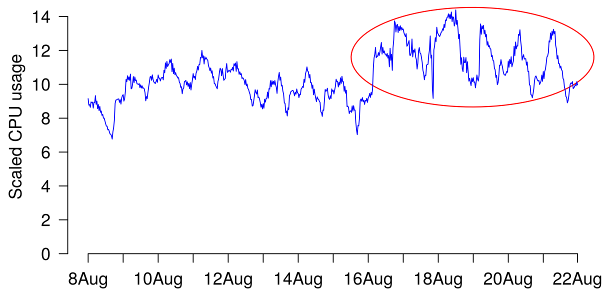


Figure 6: CPU usage goes up globally (circled) after a natural event

moves users away from servers that are being updated to other servers and thus our users get uninterrupted service.

- Repairs. Hardware and software bugs can and do happen. For example, we once had a bug in the message tokenizer that incorrectly tokenized certain email addresses in the message. This bug affected the index and thus users could not search for messages using the affected email addresses. To fix this problem, we had to reindex all affected messages using a corrected tokenizer. Given the scale of our system, such fixes can take weeks or longer to complete and while the fixes are in progress they induce their own load on the system.

- Data management. Gmail uses Bigtable as its underlying storage [11, 10]. Bigtable maintains stacks of key-value pairs and periodically compacts each stack to optimize reads and to remove overwritten values. This process is essential for the resource usage and performance of our system. These compactions occur throughout the day and are roughly proportional to the rate of updates to our system.

As with UVR work, the mix of non-UVR work also changes continuously. When possible we try to schedule non-UVR work when the system is under a low UVR load: this way we not only minimize impact on user-perceived performance but are also able to use resources reserved for UVR that would otherwise go unused (e.g., during periods of low user activity).

Figure 5 shows the scaled CPU usage of UVR work (lowest line), non-UVR work (middle line) and total scaled CPU consumed by the email backend (top line). From this we see that UVR *directly* consumes only about 20% of our CPU; indirectly UVR consumes more CPU because it also induces data management work. Thus, focusing performance analysis on just the UVR or just the non-UVR work alone is inadequate for understanding the performance of our system.

2.3 Variations due to one-off events

In addition to UVR and essential non-UVR work, our system also experiences *one-off events*. We cannot plan for one-off events: they may come from hardware or software outages or from work that must be done right away (e.g., a problem may require us to move the users served by one datacenter into other datacenters).

Figure 6 shows a 20% increase, on average, in CPU usage (circled) of our system after lightning struck Google's Belgian datacenter *four* times and potentially caused corruption in some disks [3]. Gmail dropped all the data in the affected datacenter because it could have been corrupted and automatically reconstructed data from a known uncorrupted source in other datacenters; all of this without the users experiencing any outage or issues with their email. Consequently, during the recovery period after this event, we experienced increased CPU usage in datacenters that were not *directly* affected by the event; this is because they were now serving more users than before the event and because of the work required to reconstruct another copy of the user's data.

One-off events can also interact with optimizations. For example, we wanted to evaluate a new policy for using hedged requests [13] when reading from disk. Our week-long experiment clearly showed that this change was beneficial: it reduced the number of disk operations without degrading latency.

Unfortunately, we found that this optimization interacted poorly with a datacenter maintenance event which temporarily took down a percentage of the disks for a software upgrade. Figure 7 shows the scaled 99th percentile latency of a critical email request with and without our optimization during a maintenance event. We see that during the events, our latency nearly tripled, which of course was not acceptable.

The latency degradation was caused by a bug in our heuristic for deciding whether to redirect a read to another copy of the data: our heuristic was taking too long to blacklist the downed disks and thus rather than redi-

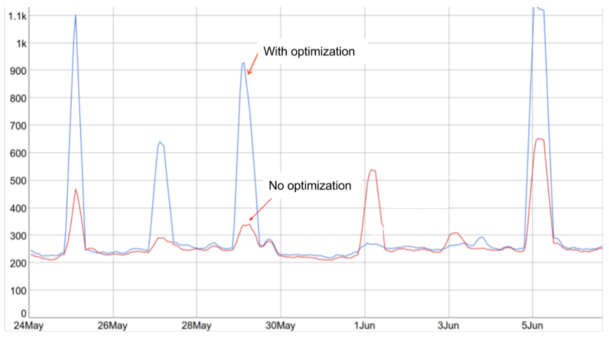


Figure 7: Scaled latency during a datacenter maintenance event

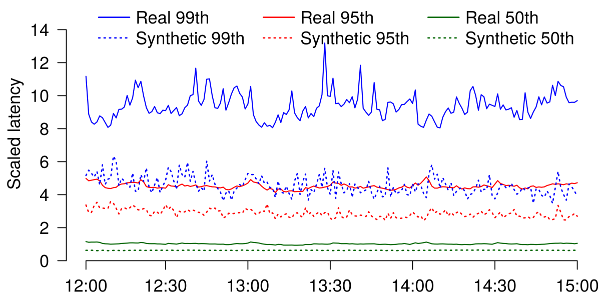


Figure 8: Latency for synthetic users versus real users at the 50th, 95th, and 99th percentiles

recting the read to another copy, it was waiting for the downed disks until the request would time out. Fixing this bug enabled our optimization to save resources without degrading latency even during maintenance events.

In summary, the resource usage during one-off events is often different from resource usage in the stable state. Since one-off events are relatively rare (affect each datacenter every few months) they are easy to miss in our experiments.

2.4 Variation in load makes it difficult to use a synthetic environment

The superimposition of UVR load, non-UVR load, and one-off events results in a complex distribution of latency and resource usage, often with a long-tail. When confronted with such long-tail performance puzzles, we first try to reproduce them with synthetic users: if we can do this successfully, it simplifies the investigation: with synthetic users we can readily add more instrumentation, change the logic of our code, or rerun operations as needed. With real users we are obviously limited because we do not want our experiments to compromise our users' data in any way and because experimenting with real users is much slower than with synthetic users: before we can release a code change to real users it needs to undergo code reviews, testing, and a gradual rollout process.

Gmail has a test environment that brings up the entire software stack and exercises it with synthetic mailboxes and load. We have modeled synthetic users as closely as we can on real users: the mailbox sizes of synthetic users have the same distribution as the mailbox sizes of real users and the requests from synthetic users is based on a model from real users (e.g., same distribution of interarrival times, same mixture of different types of requests, and similar token frequency).

Figure 8 compares the latency of a common email operation for real users to the latency for synthetic users. We collected all data from the same datacenter (different datacenters may use different hardware and thus are not always comparable). The dotted lines in Figure 8 give the latency from the synthetic load and the solid lines display the latency from the real load at various percentiles.

Despite our careful modeling, latency distribution in our test environment is different (and better) than latency distribution of real users. As discussed in Section 2.1, the continuously varying load is difficult to model in any synthetic environment and large distributed systems incorporate many optimizations based on empirical observations of the system and its users [24]; it is therefore not surprising that our test environment yields different results from real users. Even then, we find our synthetic environment to be an invaluable tool for debugging many egregious performance inefficiencies. While we cannot directly use the absolute data (e.g., latencies) we get from our synthetic environment, the relationships are often valid (e.g., if an optimization improves the latency of an operation with synthetic user it often does so for real users but the magnitude of the improvement may be different).

For most subtle changes though, we must run experiments in a live system serving real users. An alternate, less risky, option is to mirror live traffic to a test environment that incorporates the changes we wish to evaluate. While we have tried mirroring in the past, it is difficult to get right: if we wait for the mirrored operation to finish (or at least get queued to ensure ordering), we degrade the latency of user-facing operations; if we mirror operations asynchronously, our mirror diverges from our production system over time.

2.5 Effect of continuously-varying load

The continuously-varying load mixtures affect both the resource usage and latency of our application. For example, Figure 9 shows two curves: the dotted curve gives the scaled QPS to the Gmail backend and the solid line gives the 99th percentile latency of a particular operation

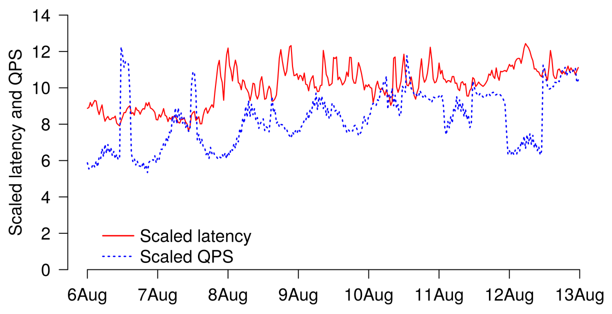


Figure 9: Changing load and changing latency

to the Gmail backend across tens of millions of users: this operation produces the list of message threads in a label (such as inbox). Since we are measuring the latency of a particular operation and our sample is large, we do not expect much variation in latency over time. Surprisingly, we see that latency varies by more than 50% over time as the load on the system changes. Furthermore the relationship between load and latency is not obvious or intuitive: e.g., the highest 99th percentile latency does not occur at the point that QPS is at its highest and sometimes load and latency appear negatively correlated.

3 Our approach

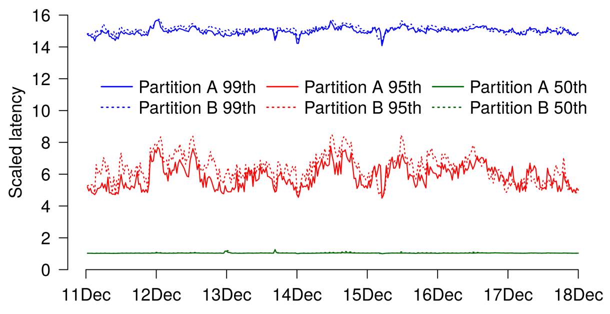
Prior work has shown that even understanding the performance of systems with stable load (e.g., [23]) is difficult; thus, understanding the performance of a system with widely varying load (Section 2) will be even harder. Indeed we have found that even simple performance questions, such as "Does this change improve or degrade latency?" are difficult to answer. In this section we describe the methodologies and tools that we have developed to help us in our performance analysis.

Since we cannot easily reproduce performance phenomena in a synthetic environment, we conduct most of our experiments with live users; Section 3.1 explains how we do this. Section 3.2 describes how we can sometimes successfully use statistics to predict the outcome of experiments and the pitfalls we encounter in doing so. Section 3.3 describes the two contexts in which we need to debug the cause of slow or resource intensive operations in a system under constantly varying load.

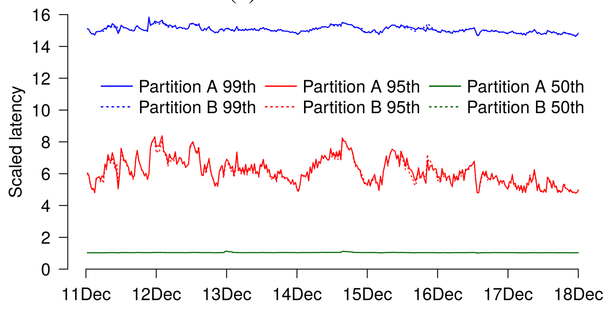
3.1 Running experiments in a live serving system

As discussed in Section 2.4, we often need to do our performance analysis on systems serving live users. This section describes and explains our approach.

To conduct controlled experiments with real users we



(a) 100K users



(b) X0M users

Figure 10: Latency comparison: (a) 100K users, (b) X0M users

partition our users (randomly) and each experiment uses one partition as the test and the other as the control. The partitions use disjoined sets of processes: i.e., in the non-failure case, a user is served completely by the processes in their partition. Large partitions enable us to employ the law of large numbers to factor out differences between two samples of users. We always pick both the test and control in the same datacenter to minimize the difference in hardware between test and control and we have test and control pairs in all datacenters.

Figure 10(a) shows scaled 50th, 95th and 99th percentile latency for two partitions, each serving 100K active users. Each partition has the same number of processes and each process serves the same number of randomly selected users. We expect the two partitions to have identical latency distributions because neither is running an experiment but we see that this is not the case. For example, the 95th percentiles of the two partitions differ by up to 50% and often differ by at least 15%. Thus, 100K randomly selected users is clearly not enough to overwhelm the variation between users and operations.

Figure 10(b) shows scaled 50th, 95th, and 99th percentile latency for two partitions, each serving tens of millions users. With the exception of a few points where the 95th percentiles diverge, we see that these partitions are largely indistinguishable from each other; the differences between the percentiles rarely exceed 1%.

Given that many fields routinely derive statistics from fewer than 100K observations, why do our samples differ by up to 50% at some points in time? The diversity of our user population is responsible for this: Gmail users exhibit a vast spread of activity and mailbox sizes with a long-tail. The 99th percentile mailbox size is four orders of magnitude larger than the median. The resource usage of many operations (such as synchronization operations and search) depend on the size of the mailbox. Thus, if one sample of 100K users ends up with even one more large user compared to another sample, it can make the first sample appear measurably worse. With larger samples we get a smaller standard deviation (by definition) in the distribution of mailbox sizes and thus it is less likely that two samples will differ significantly by chance. Intuitively, we need a greater imbalance (in absolute terms) in mailbox sizes as samples increase in size before we observe a measurable difference between the samples.

3.2 Using statistics to determine the impact of a change

Since a change may affect system performance differently under different load, we:

- Collect performance data from a sample of production. A large sample (tens of millions of randomly chosen users) makes it likely that we see all of the UVR behaviors.
- Collect performance data over an entire week and discount data from holiday seasons which traditionally have low load. By collecting data over an entire week we see a range of likely mixtures of UVR and non-UVR loads.
- Collect data from "induced" one-off events. For example, we can render one cluster as "non-serving" which forces other clusters to take on the additional load.
- Compare our test to a control at the same time and collect additional metrics to ensure that there was no other factor (e.g., a one-off event or the residual effects of a previous experiment) that renders the comparison invalid.

Once we have the data, we use statistical methods to analyze it. Unfortunately, the choice of the statistical method is not always obvious: most methods make assumptions about the data that they analyze (e.g., the distribution or independence of the data). Suppose, for example, that we want to test for statistically significant differences between latency in two partitions of randomly

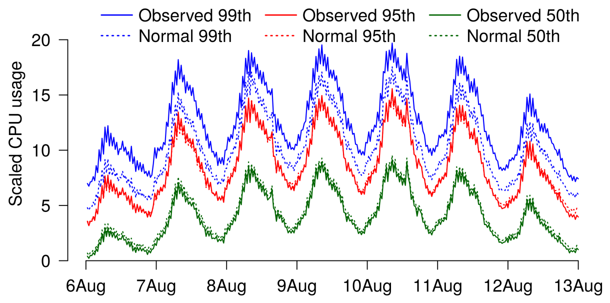


Figure 11: CPU of processes running the same binary has a near-normal distribution

selected users (as illustrated in Figure 10). Because the Kolmogorov-Smirnov (K-S) test is non-parametric (without distributional assumptions) it seems like a good candidate to use. However, due to the scale of the application, we store only aggregate latency at various percentiles rather than latency of each operation. Applying the K-S test to the percentiles (treating the repeated measurements over time as independent observations) would violate the independence assumption and inflate the power of the test. Any time we violate the assumptions of a statistical method, the method may produce misleading results (e.g., [9]).

This section gives two real examples: one where statistical modeling yields a valid outcome and one where it surprisingly does not.

3.2.1 Example 1: Data is near normal

A critical binary in our email system runs with N processes, each with C CPU cores serving U users. We wanted to deploy processes for $2U$ users but without using $2C$ CPU (our computers did not have $2C$ CPU). Since larger processes allow for more pooling of resources, we knew that the larger process would not need $2C$ CPU; but then how much CPU would it need?

Looking at the distribution of the CPU usage of the N processes, we observed that (per the central limit theorem) they exhibited a near normal distribution. Thus, at least in theory, we could calculate the properties of the larger process ($2N$ total processes) using the standard method for adding normal distributions. Concretely, when adding a normal distribution to itself, the resulting mean is two times the mean of the original distribution and the standard deviation is $\sqrt{2}$ of the standard deviation of the original distribution. Using this method we predicted the resources needed by the larger process and deployed it using the prediction.

The solid lines in Figure 11 show the observed 50th, 95th, and 99th percentiles of scaled CPU usage of the larger process using a sample of 3,000 processes. The

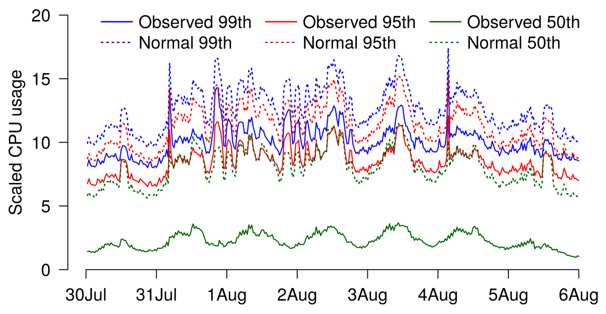


Figure 12: Time-shifted resource usage

dashed lines display the percentiles predicted by the statistical model. We see that the dashed and solid lines are close to each other (usually within 2%) at the 50th and 95th percentiles but the observed 99th percentile is always larger than the normal 99th percentile, usually by 10 to 20%. With the exception of the tail, our statistical model effectively predicted the resource needs of the larger process.

3.2.2 Example 2: Data is near normal but not independent

Our email application is made up of many different services which communicate with each other using RPC calls. Similar to the pooling in the previous section, we wanted to explore if allowing pairs of communicating processes (one from the higher-level binary and one for the lower-level binary) to share CPU would be beneficial.

The dashed ("Normal") lines in Figure 12 show the 50th, 95th, and 99th percentiles of scaled CPU usage that we expected when we shared resources between pairs of communicating processes. The solid ("Observed") lines show the actual percentiles of scaled CPU usage that we observed for the paired processes. To our surprise, the actual CPU usage is lower than the modeled CPU usage by up to 25% at the 95th and 99th percentiles and by 50% to 65% at the median. Clearly, our statistical model is incorrect.

On further investigation we determined that the CPU usage of the two communicating processes is not independent of each other: the two processes actually exhibit "time-shifted" CPU usage: when the higher-level process makes an RPC call to the lower-level process, it waits for the response and thus consumes no CPU (for that request); when the lower-level process returns a response it stops using CPU (for that request) and the higher-level process then becomes active processing the response. Thus, because these two distributions are not independent we cannot add them together.

3.3 Temporal and operation context

Section 3.2 shows how we approach establishing the impact of a change to the live system. Before we can determine what changes we need for a cloud application to become more responsive or cost effective, we need to understand why a request to a cloud application is slow or expensive. To achieve that understanding, we need to consider two contexts.

First, because cloud applications experience continuously varying load and load mixture (Section 2) we need to consider the *temporal context* of the request. By "temporal context" we mean all (possibly unrelated) concurrent activity on computers involved in serving our request throughout the time required to complete the request. Unrelated activity may degrade the performance of our request by, for example, causing cache evictions. While in our multi-tenant environment (i.e., many different applications share each computer) the operation context could include activity from processes uninvolved in serving our request, in practice we have found that this rarely happens: this is because each process runs in a container that (for the most part) isolates it from other processes running on the same computer. Thus, we are only interested in temporal context from processes that are actually involved in serving our request.

Second, a single request to a cloud application can involve many processes possibly running on different computers [21]. Furthermore, each process runs a binary with many software layers, often developed by independent teams. Thus a single user request involves remote-procedure calls between processes, functional calls between user-level software layers, and system calls between user-level software and the kernel. A request may be slow (expensive) because (i) a particular call is unreasonably slow (expensive), (ii) arguments to a particular call causes it to be slow (expensive), or (iii) the individual calls are fast but there is an unreasonably large number of them which adds up to a slow (expensive) request. Knowing the full tree of calls that make up an operation enables us to tease apart the above three cases. We call this the *operation context* because it includes the calls involved in a request along with application-level annotations on each call.

We now illustrate the value of the two contexts with a real example. While attempting to reduce the CPU reservation for the backend of our email service, we noticed that even though its average (over 1 minute) utilization was under 50%, we could not reduce its CPU reservation even slightly (e.g., by 5%) without degrading latency. The *temporal context* showed that RPCs to our service came in bursts that caused queuing even though the average utilization was low. The *full operation context* told us what operations were causing the RPCs in the bursts.

By optimizing those operations (essentially using batching to make fewer RPCs per operation) we were able to save 5% CPU for our service without degrading latency.

3.3.1 Coordinated bursty tracing

The obvious approach for using sampling yet getting the temporal context is to use bursty tracing [6, 19]). Rather than collecting a single event at each sample, bursty tracing collects a burst of events. Prior work triggers a burst based on a count (e.g., Arnold *et al.* trigger a burst on the n^{th} invocation of a method). This approach works well when we are only interested in collecting a trace at a single layer, it does not work across processes or across layers. For example, imagine we collect a burst of requests $o_1, \cdots, o_n$ arriving at a server O and each of these operations need to make an RPC to a service $\mathscr{S}$. For scalability, $\mathscr{S}$ itself is comprised of multiple servers $s_1, \cdots, s_m$ which run on many computers (i.e., it is made up of many processes each running on a different computer) and the different o_i may each go to a different server in the $s_1, \cdots, s_m$ set. Thus, a burst of requests does not give us temporal context on any of the $s_1, \cdots, s_m$ servers. As another example, the n^{th} invocation of a method will not correspond to the n^{th} invocation of a system call and thus the bursts we get at the method level and at the kernel level will not be coordinated.

We can address the above issues with bursty tracing by having each layer (including the kernel) continuously record all events in a circular buffer. When an event of interest occurs, we save the contents of the buffers on all processes that participated in the event. In practice we have found this approach to be problematic because it needs to identify all processes involved in an event and save traces from all of these processes before the process overwrites the events. Recall that the different processes often run on different computers and possibly in different datacenters.

To solve the above limitations of bursty tracing, we use *coordinated bursty tracing* which uses wall-clock time to determine the start and end of a burst. By "coordinated" we mean that all computers and all layers collect the burst at the same time and thus we can *simultaneously* collect traces from all the layers that participated in servicing a user request. Since a burst is for a contiguous interval of time, we get the temporal context. Since we are collecting bursts across computers, we can stitch together the bursts into an operation context as long as we enabled coordinated bursty tracing on all of the involved computers.

We specify our bursts using a *burst-config* which is a 64-bit unsigned integer. The burst-config dictates both the duration and the period of the burst. Most commonly it is of the form:

$(1)^m(0)^n$ (in binary)

i.e., it is m 1s followed by n 0s (in base 2). Each process and each layer performs tracing whenever:

burst-config & WallTimeMillis() == burst-config

Intuitively, each burst lasts for 2^nms and there is one burst every 2^{n+m}ms. For example if we want to trace for 4ms every 32ms we would use the burst-config of 11100 (in base 2). Unlike common mechanisms for bursty tracing, this mechanism does not require the application to maintain state (e.g., count within a burst [6, 19]) or for the different processes to communicate with each other to coordinate their bursts; instead, it can control the bursts using a cheap conditional using only *local* information (i.e., wall-clock time).

Collecting many bursts spread out over a period of time ensures that we get the complete picture: i.e., by selecting an appropriate burst-config we can get bursts spread out over time and thus over many different loads and load mixes.

There are five challenges in using coordinated bursty tracing.

First, bursty tracing assumes that clocks across different computers are aligned. Fortunately because of true time [12] clock misalignment is not a problem for us in practice.

Second, we need to identify and enable coordinated bursty tracing on *all* processes involved in our request; otherwise we will get an incomplete operation context. Because we partition our users (Section 3.1) and processes of a partition only communicate with other processes of the same partition (except in the case of failure handling) we can readily identify a set of processes for which we need to enable coordinated bursty tracing to get both the operation and the temporal context of all requests by users in the partition.

Third, since coordinated bursty tracing is time based (and not count based as in prior work on bursty tracing), a burst may start or end in the middle of a request; thus we can get incomplete data for such requests. To alleviate this, we always pick a burst period that is at least 10 times the period of the request that we are interested in. This way, while some requests will be cut off, we will get full data for the majority of requests within each burst.

Fourth, any given burst may or may not actually contain the event of interest (e.g., a slow request); we need to search through our bursts to determine which ones contain interesting events. Sometimes this is as simple as looking for requests of a particular type that take longer than some threshold. Other times, the criteria is more subtle: e.g., we may be interested in requests that make at least one RPC to service $\mathscr{S}$ and $\mathscr{S}$ returns a particular size of a response. Thus, we have built tools that use temporal logic to search over the bursts to find interesting operations [25].

Fifth, coordinated bursty tracing, and really any tracing, can itself perturb system behavior. If severe enough, the perturbation can mislead our performance analysis. The perturbation of coordinated bursty tracing depends on the traces that we are collecting; since traces for different layers have vastly different costs, we cannot quantify the perturbation of coordinated bursty tracing in a vacuum. Instead, we always confirm any findings with corroborating evidence or further experiments. In practice we have not yet encountered a situation where the perturbation due to bursty tracing was large enough to mislead us.

We use coordinated bursty tracing whenever we need to combine traces from different layers to solve a performance mystery. As a real example, cache traces at our storage layer showed that the cache was getting repeated requests for the same block in a short (milliseconds) time interval. While cache lookups are cheap, our cache stores blocks in compressed form and thus each read needs to pay the price of uncompressing the block. For efficiency reasons, the cache layer uses transient file identifiers and an offset as the key; thus logs of cache access contain only these identifiers and not the file system path of the file. Without knowing the path of the file, we did not know what data our application was repeatedly reading from the cache (the file path encodes the type of data in the file). The knowledge of the file path was in a higher level trace in the same process and the knowledge of the operation that was resulting in the reads was in another process. On enabling coordinated bursty tracing we found that most of the repeated reads were of the index block: a single index block provides mapping for many data blocks but we were repeatedly reading the index block for each data block. The problem was not obvious from the code structure: we would have to reason over a deeply nested loop structure spread out over multiple modules to find the problem via code inspection alone. The fix improved both the long-tail latency and CPU usage of our system.

3.3.2 Vertical context injection

Section 3.3.1 explores how we can collect a coordinated burst across layers. Since a burst contains traces from different layers, we need to associate events in one trace with events in another trace. More concretely, we would like to project all of the traces from a given machine so we get a holistic trace that combines knowledge from all the traces.

Simply interleaving traces based on timestamps or writing to the same log file is not enough to get a holistic trace. For example by interleaving RPC events with kernel events we will know that certain system calls, context switches, and interrupts occurred while an RPC was in progress. However, we will not know if those system calls were on behalf of the RPC or on behalf of some unrelated concurrent work.

The obvious approach to combining high- and low-level traces is to propagate the operation context through all the layers and tag all trace events with it. This approach is difficult because it requires us to propagate the context through many layers, some of which are not even within our direct control (e.g., libraries that we use or the kernel). We could use dynamic instrumentation tools, such as DTrace [1], to instrument all code including external libraries. This approach is unfortunately unsuitable because propagating the operation context through layers can require non-trivial logic; thus even if we could do this using DTrace, the code with the instrumentation would be significantly different from the code that we have reviewed and tested. Using such untested and unreviewed code for serving real user requests could potentially compromise our user's data and thus we (as a policy) never do this.

Our approach instead relies on the insight that any layer of the software stack can directly cause kernel-level events by making system calls. By making a stylized sequence of innocuous system calls, any layer can actually *inject* information into the kernel traces. [2]

For example, let's suppose we want to inject the RPC-level event, "start of an RPC," into a kernel trace. We could do this as follows:

```
syscall(getpid, kStartRpc);
syscall(getpid, RpcId);
```

The argument to the first *getpid* (*kStartRpc*) is a constant for "start of RPC." *getpid* ignores all arguments passed to it, but the kernel traces still record the value of those arguments. The argument to the second *getpid* identifies the RPC that is starting. Back-to-back *getpid* calls (with the first one having an argument of kStartRpc) are unlikely to appear naturally and thus the above sequence can reliably inject RPC start events into the kernel trace.

When we find the above sequence in the kernel trace we know that an RPC started but more importantly we know the thread on which the RPC started (the kernel traces contain the context switch events which enable us to tell which thread is running on which CPU). We can now tag all system calls on the thread with our RPC until either (i) the kernel preempts the thread, in which case the thread resumes working on our RPC when the kernel schedules it again, or (ii) the thread switches to working on a different CPU (which we again detect with a pattern of system calls).

[2] We discovered this idea in collaboration with *Dick Sites* at Google.

We use vertical context injection for other high-level events also. For example, by also injecting "just acquired a lock after waiting for a long time" into the kernel trace, we can uncover exactly what RPC was holding the lock that we were waiting on; it will be the one that invokes *sched_wakeup* to wake up the blocked *futex* call.

Implementing the above approach for injecting RPC and lock-contention information into our kernel traces took less than 100 lines of code. The analysis of the kernel traces to construct the holistic picture is of course more complex but this code is offline; it does not add any work or complexity to our live production system. In contrast, an approach that propagated context across the layers would have been far more intrusive and complex.

We use vertical context injection as a last resort: its strength is that it provides detailed information that frequently enables us to get to the bottom of whatever performance mystery we are chasing; its weakness is that these detailed traces are large and understanding them is complex.

As a concrete example, we noticed that our servers had low CPU utilization: under 50%. When we reduced the CPU per server (to increase utilization and save CPU) the latency of our service became worse. This indicated that there were micro bursts where the CPU utilization was higher than 50%; reducing the CPU per server was negatively affecting our micro bursts and thus degrading latency. We used vertical context injection along with coordinated bursty tracing to collect bursts of kernel traces. On analyzing the traces we found that reading user properties for each user request was responsible for the bursts; for each user request we need a number of properties and each property is small: e.g., one property gives the number of bytes used by the user. Rather than reading all properties at once, our system was making a separate RPC for each property which resulted in bursty behavior and short (a few milliseconds) CPU bursts. Reading all properties in a single RPC improved both the CPU and latency of our service. Vertical context injection enabled us to determine that the work in the bursts was related to servicing the RPCs involved in reading the properties.

4 Related work

We now review related work in the areas of analyzing the performance of cloud applications and performance tools.

Magpie [7, 8] collects data for all requests (thus the temporal context) and stitches together information across multiple computers to provide the operation context. Because Magpie collects data for all requests it does not scale to billions of user requests per day; Dapper and Canopy (discussed below) address this issue by using sampling.

Dapper [27] propagates trace identifiers to produce an RPC tree for an operation. We use Dapper extensively in our work because Dapper tells us exactly what RPCs execute as part of an operation and the performance data pertinent to the RPCs. By default Dapper performs random sampling: e.g., it may sample 1 in *N* traces. While this gives us the operation context, it does not give us the temporal context. Thus, we use Dapper in conjunction with coordinated bursty tracing and vertical context injection.

Canopy [22] effectively extends Dapper by providing APIs where any layer can add new events to a Dapper-like trace and users can use DSL to extend traces. Canopy's solution to combining data from multiple layers is to require each layer to use a Canopy API; while this is effective for high-level software layers, it may be unsuitable for low-level libraries and systems (e.g., OS kernel) because it creates a software dependency from critical systems software on high-level APIs.

Xtrace [15] uses auxiliary information (X-Trace Metadata) to tie together events that belong to the same operation. Thus, one could use Xtrace to tie together high-level RPC events and kernel-level events; however, unlike vertical context injection, Xtrace requires each layer to explicitly propagate the metadata across messages.

Jalaparti *et al.* [21] discuss the variations in latency that they observe in Bing along with mitigation strategies that involve trading off accuracy and resources with latency. Unlike our work, Jalaparti *et al.* do not explore the effects of time-varying load on performance.

Vertical profiling [17] recognizes the value of the operation context and subsequent work extends vertical profiling to combine traces from multiple layers using trace alignment [16]. Unlike our work, this work only considers the operation context within a single Java binary.

5 Discussion

This paper shows that for Gmail the load varies continually over time; these variations are not just due to changes in QPS; the actual nature of the load changes over time (for example, due to the the mix of IMAP traffic and web traffic). These variations create challenges for evaluating the performance impact of code changes. Concretely, (i) we cannot compare before-after performance data at different points in time and instead we run the test and control experiments simultaneously to ensure that they experience comparable loads; (ii) it is difficult to model this changing load in a synthetic environment and thus we conduct our experiments in a system serving live users; and (iii) we need large sample sizes (millions of users) to get statistically significant results.

We can take each source of variation described in this paper and devise a fix for it. For example, by isolating

IMAP and web traffic in disjoint services we circumvent the variation due to different mixes of IMAP and web traffic over the course of time. Unfortunately, such an approach does not work in general: there are many other causes of load variation besides the ones that we have described and fixes for each of the many causes becomes untenable quickly. The remainder of this section discusses some other sources of variation and in doing so generalizes the contributions of this paper.

The continuously changing software [14] causes performance to vary. A given operation to a Cloud application may involve hundreds of RPCs in tens of services; for example a median request to Bing involves 15 stages processed across 10s to 1000s of servers [21]. Different teams maintain and deploy these stages independently. To ensure the safety of the deployed code, we cannot deploy new versions of software atomically even if the number of instances is small enough to allow it (e.g., [20]). Instead we deploy it in a staged fashion: e.g., first we may deploy a software only to members of the team, then to random users at Google, then to small sets of randomly picked external users, and finally to all users. As a result, at any given time many different versions of a software may be running. Two identical operations at different points in time will often touch different versions of some of the services involved in serving the operation. Since different versions of a server may induce different loads (e.g., by making different RPCs to downstream servers or by inducing different retry behavior on the upstream servers), continuous deployment of software also continuously changes the load on our system.

Real-world events can easily change the mix or amount of load on our systems and thus cause performance to vary. For example, Google applications that are popular in academia (e.g., Drive or Classroom) see a surge of activity at back-to-school time. Some world events may be more subtle: e.g., a holiday in a place with limited or expensive internet connectivity may change the mix of operations to our system because clients in such areas often access our systems in a different mode (e.g., using offline mode or syncing mode) than clients in areas with good connectivity.

Datacenter management software can change the number of servers that are servicing user requests, move servers across physical machines, and turn up or turn down VMs; in doing so they directly vary the load on services.

In summary, there are many and widespread reasons beyond the ones explored in this paper that result in continuously varying load on cloud applications. Thus, we believe that the approaches in this paper for doing experiments in the presence of these variations and the tools that we have developed are widely applicable.

6 Conclusions

Performance analysis of cloud applications is important; with many cloud applications serving more than a billion active users, even a 1% saving in resources translates into significant cost savings. Furthermore, being able to maintain acceptable long-tail latency at such large scale requires constant investment in performance monitoring and analysis.

We show that performance analysis of cloud application is challenging and the challenges stem from constantly varying load patterns. Specifically, we show that superimposition of requests from users, different email clients, essential background work, and one-off events results in a continuously changing load that is hard to model and analyze. This paper describes how we meet this challenge. Specifically, our approach has the following components:

- We conduct performance analysis in a live application setting with each experiment setting involving millions of real users. Smaller samples are not enough to capture the diversity of our users and their usage patterns.
- We do longitudinal studies over a week or more to capture the varying load mixes. Additionally, we primarily compare data (between the test and control setups) at the same point in time so that they are serving a similar mix of load.
- We use statistics to analyze data and to predict the outcome of risky experiments and we corroborate the results with other data.
- We use coordinated bursty tracing to capture bursts of traces across computers and layers so we get both the temporal and the operation context needed for debugging long-tail performance issues.
- We project higher-level events into kernel events so that we can get a vertical trace which allows us to attribute all events (low or high level) to the high-level operation (e.g., RPC or user operation).

We have used the above strategies for analyzing and optimizing various large applications at Google including Gmail (more than 1 billion users) and Drive (hundreds of millions users).

7 Acknowledgments

Thanks to Brian Cooper, Rick Hank, Dick Sites, Garrick Toubassi, John Wilkes, and Matei Zaharia for feedback on the paper and for countless discussions on performance.

References

[1] About dtrace. http://dtrace.org/blogs/about.

[2] Google research blog: Speed matters. http://googleresearch.blogspot.com/2009/06/speed-matters.html, June 2009.

[3] Google loses data as lightning strikes. BBC News, Aug. 2015. http://www.bbc.com/news/technology-33989384.

[4] Facebook stats. http://newsroom.fb.com/company-info/, 2016.

[5] Gmail now has more than 1B monthly active users. https://techcrunch.com/2016/02/01/gmail-now-has-more-than-1b-monthly-active-users/, Feb. 2016.

[6] ARNOLD, M., HIND, M., AND RYDER, B. G. Online feedback-directed optimization of java. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2002), OOPSLA '02, ACM, pp. 111–129.

[7] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 18–18.

[8] BARHAM, P., ISAACS, R., MORTIER, R., AND NARAYANAN, D. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9* (Berkeley, CA, USA, 2003), HOTOS'03, USENIX Association, pp. 15–15.

[9] BLACKBURN, S. M., DIWAN, A., HAUSWIRTH, M., SWEENEY, P. F., AMARAL, J. N., BRECHT, T., BULEJ, L., CLICK, C., EECKHOUT, L., FISCHMEISTER, S., FRAMPTON, D., HENDREN, L. J., HIND, M., HOSKING, A. L., JONES, R. E., KALIBERA, T., KEYNES, N., NYSTROM, N., AND ZELLER, A. The truth, the whole truth, and nothing but the truth: A pragmatic guide to assessing empirical evaluations. *ACM Trans. Program. Lang. Syst. 38*, 4 (Oct. 2016), 15:1–15:20.

[10] Apache cassandra. http://cassandra.apache.org/.

[11] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26.

[12] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 251–264.

[13] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56* (2013), 74–80.

[14] FEITELSON, D., FRACHTENBERG, E., AND BECK, K. Development and deployment at facebook. *IEEE Internet Computing 17*, 4 (July 2013), 8–17.

[15] FONSECA, R., PORTER, G., KATZ, R. H., AND SHENKER, S. X-trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)* (Cambridge, MA, 2007), USENIX Association.

[16] HAUSWIRTH, M., DIWAN, A., SWEENEY, P. F., AND MOZER, M. C. Automating vertical profiling. *SIGPLAN Not. 40*, 10 (Oct. 2005), 281–296.

[17] HAUSWIRTH, M., SWEENEY, P. F., DIWAN, A., AND HIND, M. Vertical profiling: Understanding the behavior of object-priented applications. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2004), OOPSLA '04, ACM, pp. 251–269.

[18] HICKS, M., AND NETTLES, S. Dynamic software updating. *ACM Trans. Program. Lang. Syst. 27*, 6 (Nov. 2005), 1049–1096.

[19] HIRZEL, M., AND CHILIMBI, T. Bursty tracing: A framework for low-overhead temporal profiling. In *In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization* (2001), pp. 117–126.

[20] ISARD, M. Autopilot: Automatic data center management. Tech. rep., April 2007.

[21] JALAPARTI, V., BODIK, P., KANDULA, S., MENACHE, I., RYBALKIN, M., AND YAN, C. Speeding up distributed request-response workflows. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 219–230.

[22] KALDOR, J., MACE, J., BEJDA, M., GAO, E., KUROPATWA, W., O'NEILL, J., ONG, K. W., SCHALLER, B., SHAN, P., VISCOMI, B., VENKATARAMAN, V., VEERARAGHAVAN, K., AND SONG, Y. J. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 34–50.

[23] LI, J., SHARMA, N. K., PORTS, D. R. K., AND GRIBBLE, S. D. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 9:1–9:14.

[24] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi'13, USENIX Association, pp. 385–398.

[25] RYCKBOSCH, F., AND DIWAN, A. Analyzing performance traces using temporal formulas. *Softw., Pract. Exper. 44*, 7 (2014), 777–792.

[26] SCHURMAN, E., AND BRUTLAG, J. Performance related changes and their user impact. http://conferences.oreilly.com/velocity/velocity2009/-public/schedule/detail/8523, June 2009. Additional Bytes, and HTTP Chunking in Web Search.

[27] SIGELMAN, B. H., BARROSO, L. A., BURROWS, M., STEPHENSON, P., PLAKAL, M., BEAVER, D., JASPAN, S., AND SHANBHAG, C. Dapper, a large-scale distributed systems tracing infrastructure.

[28] VEERARAGHAVAN, K., MEZA, J., CHOU, D., KIM, W., MARGULIS, S., MICHELSON, S., NISHTALA, R., OBENSHAIN, D., PERELMAN, D., AND SONG, Y. J. Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 635–651.