Weighted Sampling of Execution Traces: Capturing More Needles and Less Hay

Pedro Las-Casas
UFMG
Belo Horizonte, Brazil
pedro.lascasas@dcc.ufmg.br

Jonathan Mace
MPI-SWS
Saarbrücken, Germany
jcmace@mpi-sws.org

Dorgival Guedes
UFMG
Belo Horizonte, Brazil
dorgival@dcc.ufmg.br

Rodrigo Fonseca
Brown University
Providence, RI
rfonseca@cs.brown.edu

Abstract

End-to-end tracing has emerged recently as a valuable tool to improve the dependability of distributed systems, by performing dynamic verification and diagnosing correctness and performance problems. Contrary to logging, end-to-end traces enable coherent sampling of the entire execution of specific requests, and this is exploited by many deployments to reduce the overhead and storage requirements of tracing. This sampling, however, is usually done uniformly at random, which dedicates a large fraction of the sampling budget to common, 'normal' executions, while missing infrequent, but sometimes important, erroneous or anomalous executions. In this paper we define the representative trace sampling problem, and present a new approach, based on clustering of execution graphs, that is able to bias the sampling of requests to maximize the diversity of execution traces stored towards infrequent patterns. In a preliminary, but encouraging work, we show how our approach chooses to persist representative and diverse executions, even when anomalous ones are very infrequent.

CCS Concepts

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Software testing and debugging**; • **Information systems** → *Clustering*;

Keywords

distributed tracing, weighted sampling

ACM Reference Format:
Pedro Las-Casas, Jonathan Mace, Dorgival Guedes, and Rodrigo Fonseca. 2018. Weighted Sampling of Execution Traces: Capturing More Needles and Less Hay. In *ACM Symposium on Cloud Computing (SoCC '18), October 11–13, 2018, Carlsbad, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3267809.3267841

1 Introduction

End-to-end tracing has emerged in the past decade as a valuable tool to diagnose correctness and performance problems in distributed systems [8, 18, 20, 24, 25]; model workloads, resource usage, and timings [2, 4, 14, 24, 25]; and to detect anomalous requests at runtime [2, 5, 17, 20]. It has also been gaining widespread use in industry, with ongoing standardization efforts [16, 27] and use in many companies and frameworks [9, 24, 28]. In the increasingly distributed environments of today, the coherent view of executions provided by end-to-end tracing, spanning the boundaries of components, layers, machines, and even administrative domains, enables developers and operators to reason about, and understand, how their systems run, and why they fail.

One important aspect of end-to-end tracing is that it allows for coherent sampling of execution events, *i.e.*, the capture, across all components, of all events related to a particular execution or request, which can't be done with traditional, component-centric logging [15]. Given the scale of instrumented distributed systems, this becomes especially valuable. For example, both Google [24], and, more recently, Facebook [10] mention request-based sampling as a fundamental aspect of their tracing systems. The W3C's recent specification for a trace context HTTP header [27], which follows the Dapper span model, has 'sampled' as one of three top-level components. These systems vary in the complexity of their sampling policies, including a specified fraction of requests to be sampled (often 10^{-5} in Dapper [24], in 2010), or a rate limit on generated samples. After some filtering and rate limiting, however, all current systems resort to uniform sampling to choose which executions to keep.

The problem with uniform sampling is that, given a fixed budget for storing request traces, most of the budget will be dedicated to a few classes of 'normal' traces, and almost no resources will be dedicated to recording infrequent, anomalous requests. In particular, if a problematic type of execution only happens with very low frequency, with uniform sampling the global sampling frequency has to be increased to capture enough of these executions. As a simple example, Figure 1 plots the distribution of HTTP status codes in events in a trace of a large ride sharing provider using microservices (note the logarithmic scale). More than 85% of the events have status 200 OK, and the four most popular statuses

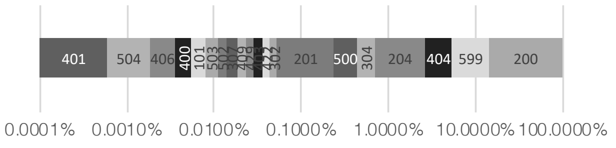


Figure 1: Distribution of HTTP status codes of a microservices trace from a large ride sharing provider. X axis is scaled logarithmically.

account for over 99.2% of the events. There are however 17 other status codes that together account for only 0.7% of all events!

In this paper, we introduce the representative trace sampling problem, and provide initial solutions based on clustering that work both online and offline. In short, our goal is to make the probability of persisting an execution trace higher the more unique that trace is. Differently from classic statistical techniques of stratified sampling [26] or importance sampling [23], in our scenario we cannot assume that we know the types of traces that we will collect a priori, and much less their frequency distribution. While we do not claim to completely solve the problem, we demonstrate promising early results in controlled scenarios, and in an initial analysis of production traces.

2 Background

We assume that we have a distributed system (*e.g.* a cluster running Spark over HDFS, or a set of loosely coupled microservices) and that the system is instrumented with end-to-end tracing [8, 16, 19, 24]. The instrumentation produces traces for executions of the system.

A common format for these traces is a tree of *spans*, as defined by Dapper [24], and used by most open source tracing frameworks of today. A span is a portion of an execution of a system, usually in a single component, with a defined start and end. Spans can initiate other spans (*e.g.*, the server side of an RPC call), and a child span keeps a reference to the parent span. Spans can have internal annotations, and, in some frameworks, can also link back to parent nodes when done. X-Trace uses a more general model, consisting of a directed acyclic graph of events, where edges represent causality [8]. In this work, we consider a trace to be a DAG of events, and can easily convert a tree of spans with annotations to this format.

Usually not all executions of the system are logged, and traces can be sampled. Sampling techniques include *head-based coherent sampling* and *tail-based sampling*. In head-based sampling, the sampling decision is made at the initial event and metadata is propagated through the execution, indicating if the trace points should generate trace events. As mentioned in §1, even though there are some variations, head-based sampling chooses which traces to generate and persist uniformly at random.

Alternatively, in tail-based methods, the sampling decision is made at the end of an execution. While more expensive a priori – as more trace data is generated and kept in memory for a period of time – it is possible to use more intelligent sampling and to persist only traces that are "interesting". Even in cases where all executions are traced (*e.g.*, [13]), selecting interesting or representative traces is important, to reduce ongoing storage costs, or to select what traces to show to operators or developers with a finite cognitive budget.

3 Problem Definition

Given the limitations of uniform sampling, here we define more precisely the problem we are trying to solve:

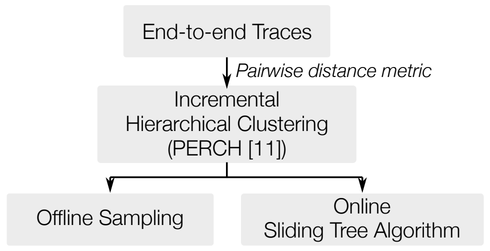
Representative trace sampling problem: *Given a set of execution traces and a sampling budget s, select a subset of the traces that is maximally diverse (or minimally redundant).*

Let us first look at an idealized case in which N total traces can be classified into a well-defined set of K clusters, with $|C_i|, 1 \leq i \leq K$, as the number of traces in each cluster. The sampling budget is $sN = S$ traces, *i.e.* s is the fraction of the N traces we can store. We also assume that s is enough to capture at least one of each type of trace, *i.e.*, $sN \geq K$. Intuitively, we would like to have an equal representation of each cluster in our sample, except when there are not enough traces of a given cluster. A max-min fairness allocation of sampled traces per cluster provides precisely this [3]. With this allocation, for each cluster, one will have $S_i = \min(|C_i|, \sigma)$, where $\sigma \geq S/K$ is such that $\sum S_i = S$. If $|C_i| \geq S/K \; \forall i$, then $\sigma = S/K$, *i.e.*, when there are enough traces in each cluster, all clusters get exactly the same "fair" share of S. It turns out that the max-min fair allocation is also the allocation with maximum entropy, as shown by Coluccia *et al.* [6], and this is in line with the problem definition.

Unfortunately, in a realistic setting the context is not so clear, because we cannot assume that this clustering exists: executions differ for many reasons such as timing, and execution path, and systems have multiple components that are loosely coupled, and which change independently. Even if it does, we do not know what the clustering is, or how many clusters there are, nor the probability distribution over the clusters.

We assume in this work, instead, that we can compute a distance metric among any two traces, and use this to approximate a sampling strategy that maximizes the diversity of the resulting sample. The intuition is that a trace that is very similar to many other traces adds little information, and would decrease the entropy of the sample. Traces that are more unique, on the other hand, would increase the resulting entropy.

Further challenges in solving this problem in a practical setting are that this has to be done in an online fashion, as new traces arrive, at scale, and in comparison to the traces that we have already stored (as opposed to all other traces seen).


End-to-end Traces
Pairwise distance metric
Incremental Hierarchical Clustering (PERCH [11])
Offline Sampling
Online Sliding Tree Algorithm
Figure 2: Overview of our sampling approach.

4 An approximate solution

In this section we describe our approach, using hierarchical clustering, to derive a sampling strategy that approximates a solution to the representative trace sampling problem above. Figure 2 outlines our approach. We will first describe the clustering and the sampling algorithms, and defer the description of how we process the traces, and the definition of a distance metric between two traces to Section 4.4. For now, it is sufficient to assume we have such a distance metric.

4.1 Hierarchical Clustering

The first step in our sampling strategy is to organize the traces in a hierarchical clustering binary tree, also called a dendrogram. While there are several approaches to hierarchical clustering, we adopt the PERCH algorithm (Purity Enhancing Rotations for Cluster Hierarchies) [11]. Perch is a recently proposed non-greedy, incremental algorithm for hierarchical clustering that scales to both massive N points and K clusters. This algorithm constructs a tree over data points that are incrementally inserted to the leaves, using rotation operations that maintains its dendrogram *purity*. Assuming the existence of an underlying clustering (separable data), purity roughly measures how close a dendrogram tree is to this ground truth clustering, by comparing, for each pair of nodes in the underlying cluster, how close they are in the tree. We refer to the original paper [11] for the precise definition. When purity decreases, or when the tree becomes unbalanced, the algorithm will perform recursive rotations to restore these properties. The tree structure enables an efficient search that scales over large datasets and is often logarithmic.

In PERCH, each new trace is inserted next to its nearest neighbor, found using A* search. In order to reduce the computational cost of the algorithm, the authors use approximations based on bounding boxes. Each internal node of the tree maintains a bounding box that has all its leaves. The heuristic used in the A* search algorithm is the minimum distance to each node based on the bounding box. Thus, instead of comparing to all leaves of an internal node, they simply compare the new point to the bounding box, reducing the computational cost. In order to reduce even more the cost of the nearest neighbor search, it is possible to use the search algorithm with predefined beam width, that is, the algorithm only allows a limited number of nodes to be in the frontier at any time. That solution will not guarantee that the best solution will be found, but it reduces the cost of this operation.

Because of its node rotations, PERCH trees tend to be balanced, and this is helpful in our sampling strategy.

4.2 Offline Sampling

Given the binary tree generated by the hierarchical clustering, our offline sampling strategy is very simple, and given in Algorithm 1: to produce a sample, start at the root of the tree and for each branch choose one side with 50% chance. Repeat this without replacement until the desired number of samples is reached. All traces are in the leaves of the tree. For each individual trace, at depth d, the probability of being sampled is 2^{-d}.

Algorithm 1 Dendrogram Sampling algorithm

```
        Input: tree and n, number of traces to be sampled.
              Output: $s_{trace}$, set of sampled traces.
 1: $s_{trace} \leftarrow set()$
 2: $c_{trace} \leftarrow 0$
 3: while $c_{trace} < n$ do
 4:     $node \leftarrow tree.root()$
 5:     while node is not leaf do
 6:         $node \leftarrow choose(node.children)$        ▷ Select a child randomly.
 7:     if node not in $s_{trace}$ then
 8:         $s_{trace}.add(node)$
 9:         $c_{trace} \leftarrow c_{trace} + 1$
10: return $s_{trace}$
```

Under this strategy, in an ideal case with K clusters, where κ is an integer and $K = 2^{\kappa}$, and in which the first κ levels of the tree are balanced, each cluster will have the same probability $2^{-\kappa}$ of having elements sampled, independent of the number of nodes in each cluster. If a cluster then has $C_i = 2^{\gamma}$ traces, and again the subtree is balanced, each node will be at depth $\kappa + \gamma$, and will have a probability $2^{-\kappa} \times 2^{-\gamma}$ of being sampled, which equals $2^{-\kappa}$ if multiplied by $C_i = 2^{\gamma}$.

In practice, neither the number of clusters nor the number of nodes per cluster will be powers of 2, and the tree will not be perfectly balanced. This is a tradeoff in a binary tree that will make the sampling deviate from the ideal case. However, we found that this strategy, due to the properties of the clustering algorithm, seems to work well in practice. PERCH has two desirable properties: dendrogram purity, which strives to place similar nodes close in the tree, and the balancing, which tends to make subtrees balanced, provided this does not violate purity as well. Similar and frequent executions will likely be placed in the same branch, making it deeper than the branches containing infrequent ones. Rare traces, different from other traces, will tend to be in separate branches, with less nodes and, thus, more shallow. These will tend to be sampled with higher probability.

4.3 Online Sampling: Sliding Tree

While we use the offline algorithm to evaluate the quality of the clustering, it is not very practical to have all traces and then sample them. In this section, we describe an online sampling approach, which we call the Sliding Tree algorithm. The name comes from an analogy with a sliding window, as we use the tree to store a fixed number of traces.

With each new trace, the algorithm (listed in Algorithm 2) first checks to see if the tree has reached its maximum size. If so, it deletes an element that is among the ones with the lowest probability of being sampled. To do this, the algorithm starts at the root and follows the branch with the largest number of leaves, until it reaches a leaf. Ties are broken randomly. After this element is removed, the algorithm inserts the new element in its regular place, *i.e.*, adjacent to its closest node already in the tree.

Algorithm 2 Sliding Tree Algorithm

```
                 Input: tree, n, trace
                    Output: tree
        ▷ Every node in the tree keeps information about its leaves.
1: if $|tree| == n$ then
2:     delete-unlikely-node(tree)
3: Insert new trace (tree, trace)              ▷ Algorithm from [11].
```

Algorithm 3 Delete Unlikely Node

```
1: procedure DELETE-UNLIKELY-NODE(tree)
2:     $node \leftarrow tree.root()$
3:     while node is not leaf do
4:         $node \leftarrow max(child.leaves$ for child in $node.children)$
5:     delete node
6:     update leaves property for all ancestors
```

The use of this strategy makes it difficult to remove rare traces from the tree since they are likely to have high sampling probability.

In contrast, frequent executions do not flood the tree, since when a new trace of this type is to be inserted, it simply replaces another one of this same type. This way, we can balance the tree with the different trace types, as will be shown in Section 5. This algorithm also decouples the insertion cost from the volume of traces, as the cost of insertion depends only on the (fixed) size of the tree, and not on the total number of traces. Most of the cost of storing traces is also done outside of the critical path of the traced application, causing only a small overhead to a production system.

4.4 Distance between traces

Our approach is extensible to different types of traces, provided we can compute a meaningful distance between two traces. In this paper, we use X-Trace event-based traces and OpenTracing traces that we convert to the same event format. X-Trace traces are represented as event DAG's. Given a graph, we summarize it as an array and use the Euclidean distance as a metric to compare traces. We propose three different approaches to represent the execution graphs as arrays, but only present results in this paper for the simplest, node-based one below.

A graph $G = (V, E, \ell)$ consists of a set of vertices $V = v_0, v_1, ..., v_n$, a set of directed edges $E \subset V \times V$, and a labeling function $\ell : V \rightarrow \Sigma$ that assigns labels from an alphabet Σ to vertices. When G represents an execution graph, V is constructed from the events of the execution, E from the causal edges, and ℓ is constructed from the user-specified labels that the execution graph records for each event. That is, for any nodes $v_0 \in G_0$ and $v_1 \in G_1$, $\ell(v_0) = \ell(v_1)$ implies that v_0 and v_1 are different occurrences of the same event. Commonly this means that the events were generated by the same line of code, though specifying event labels is a design decision at the time of instrumentation.

From these graphs we generate a node-based array, where each distinct node of the graph is represented as an entry in the array, and its value indicates the number of times this event occurred in the trace. We use the Euclidean distance as the metric between two vectors. We also experimented with other vector representations, namely using edge names (as the concatenation of the names of the two adjacent nodes), and a generalization based on the Weisfeiler-Lehman isomorphism test [21], but we leave further investigation of these other metrics as future work.

5 Evaluation

We evaluate our solution to the representative sampling problem in three steps and two datasets. First, we evaluate the clustering algorithm (and our distance metric among traces) using the offline algorithm on a set of controlled, small scale experiments in which we artificially create diverse executions of known types and frequencies. Second, we evaluate the online tree sliding algorithm on the same datasets. Lastly, we present an initial evaluation of the offline algorithm on a set of real production traces from a large ride sharing provider. Note that in the controlled experiments, while we created a known number of classes of executions (*e.g.*, executions with a failure), we only use this knowledge for the evaluation, the algorithm does not use any a priori information about the nature of the executions.

5.1 Controlled Experiments

We instrumented Spark, YARN and HDFS with the X-Trace framework and deployed it on a small, 11-node cluster. We use X-Trace based on automatic instrumentation of the systems, that is, the events to be recorded in the traces are created based on log4j/commons logging calls. While this is a low effort way of instrumenting these systems, the generated traces are large, and filled with events that have little discriminatory power for the graphs. Thus, we summarize the graphs aiming to remove those events.

We do this by eliminating nodes in the graph that are abstracted by redundant edges. An edge in G is said to be redundant if it can be removed without changing the reachability of the graph. Thus, an edge (i, j) is redundant if there is a path from event i to event j without the need to use edge (i, j). We can remove the nodes in the longer path. To find redundant edges, we identify the transitive reduction of the graph [7]. The original graphs have an average of 16,502 nodes and 32,967 edges, while the summarized ones average 7,536 nodes and 8,753 edges.

Using this instrumentation, we ran two different experiments in order to understand and evaluate how our sampling behaves.

Memory Fault We simulated a system fault during the execution by limiting the total memory for one of the VM's in the cluster. While a normal VM has 8 GB memory, we limited one with only 512 MB. We then ran a Spark WordCount job with a 30 GB input file. During the execution, the low memory VM would fail and all its task would have to be resubmitted to the other nodes.

VM Fail In these experiments, we simulate the failure of one of the VMs during Spark job execution. We randomly selected one of the machines and stopped it. This way, all tasks being performed on this machine had to be reallocated to another VM.

5.2 Clustering Evaluation

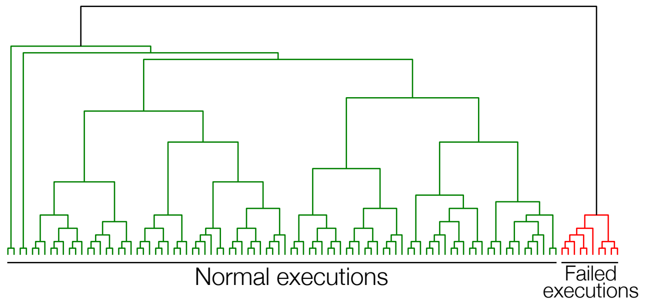


Figure 3: Hierarchical tree with 90 normal traces and 10 failed executions.

To evaluate the quality of the sampling, we analyzed the two experiments described above, varying the distribution of the different execution trace classes. For comparison purposes, we also present the results obtained by sampling uniformly at random.

In the memory failure case, we started the experiment with 99 normal execution traces and only one failed execution (1%). Then we increase the proportion until the failed executions reach 20% of the total. Figure 3 shows the tree obtained when applying the hierarchical clustering to a scenario with 10% traces with failures. In this case, the clustering is perfect, with all the executions with failure in the same subtree, split from the others at the first level. In this particular tree, we sample from each type of execution with the same probability, until we run out of executions with failure.

Figure 4 shows the result obtained when considering a 15% sampling ratio (we select 15 out of 100 executions), for both our sampling (weighted), and uniform sampling ('random'). For each scenario, we performed 100 different experiments, randomly varying the traces used, and the order they are inserted in the tree. As we vary the fraction of executions with a failed VM, the Y-axis shows the fraction of the sampled traces that contained an execution with a failure. According to the max-min allocation (cf. §2), the expected fraction of traces with a failure for the sampled traces is $\min(\frac{|C_i|}{S}, 1/K)$, indicated in the plot as the 'expected' line.

It is worth noticing that, although in production distributed systems the sampling percentage is much lower (Dapper uses less than 1%, for example), we used 15% due to the small number of executions we have. As can be seen, random sampling is proportional to the distribution of traces, that is, failed executions are rarely selected by the sampling. In contrast, our sampling sampled both execution classes with equal probability.

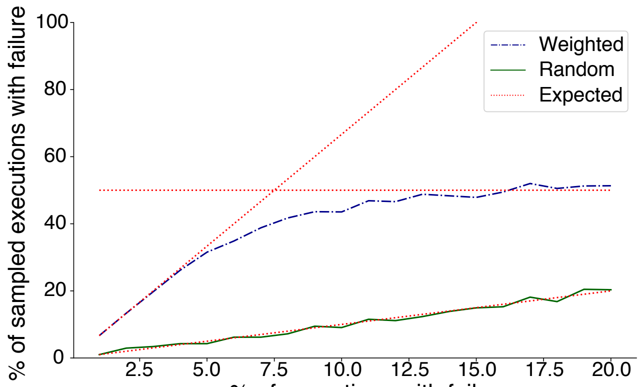


Figure 4: Percentage of failed traces (low memory) in the selected sample.

For the VM Fail experiment, we also varied the distribution of normal and failed executions. However, in this case, the failed class has 4 subclasses, one for each different VM stopped during the execution. Thus, we would expect normal traces representing 20% of the sampled traces and the failed ones, 80%. As can be seen in Figure 5, the failed traces achieve around 85% of the sampled traces when they represent 20% of all performed executions.

5.3 Sliding Tree Evaluation

To evaluate the online sliding tree algorithm, we defined a sliding tree containing 50 leaves (traces). For the Memory Fault case, we inserted 100 normal traces and 30 faulty ones. We performed 100 different experiments varying the order of insertion, and in each of them, the order of insertion of traces was randomly defined, and plotted the average number of traces for each type over time. Figure 6 shows that, until reaching 50 traces inserted, the tree continues to grow, being composed, mostly, by normal traces. However, from this moment, for each new trace, a normal execution is removed from the tree, since it is more frequent, until it converges

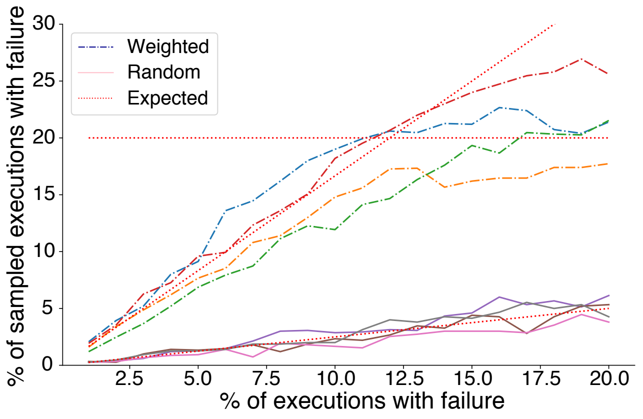


Figure 5: Percentage of failed traces (VM Fail) in the selected sample.

and the number of normal executions and failures present the same proportion.

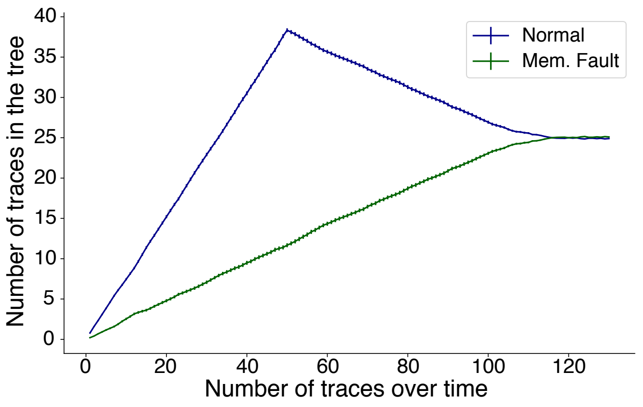


Figure 6: Online evaluation for the Memory Fault experiment.

5.4 Production Traces

Our experiments in a controlled environment showed the potential of our sampling approach. We also applied our method in real production data from a large ride sharing provider. Their infrastructure is based on microservices, using an OpenTracing-based tracing environment. For this preliminary work, we tested our representative sampling approach using 1,000 different traces. Since we do not have the ground truth for these traces, we measure the quality of our sampling in a different way, by using the sampled traces to reconstruct the original trace (as in a lossy compression scheme). More specifically, we can use a subset of the original traces to represent the whole set by replacing each missing original trace with a reference to the closest trace in the sample. Then, to measure the quality of the approximation, for each trace in the original set, we determine the distance (using our trace distance metric) to the closest trace in the sampled set. We report the sum of this distance for all traces in the original set. A better sample will have a smaller overall distance. Figure 7 shows the results for our approach and for random sampling, averaged over 10 runs for each sampling ratio.

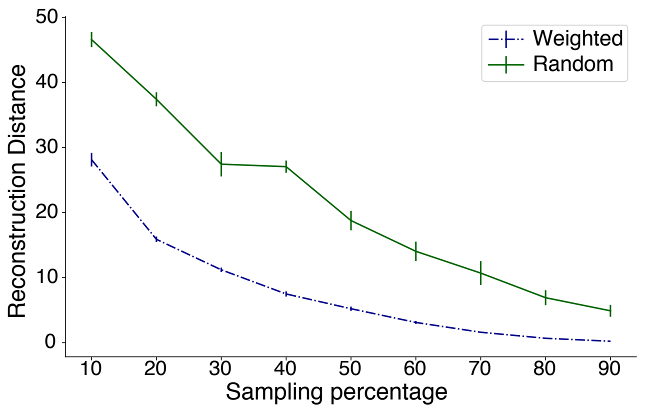


Figure 7: Distance between a reconstruction of all traces from a sample and the original set of traces.

The distance between the selected samples using our approach is significantly lower than the samples selected through the random sampling, meaning that the chosen set is a better approximation of the original set of traces, *i.e.*, they are more representative of the original set. We plan to further investigate the quality of our sampling with other methods as future work.

6 Related Work

Causal end-to-end tracing has emerged as a valuable tool for improving the dependability of distributed systems by recording, diagnosing and analyzing their execution across components, recording their causality according to Lamport's happens before relation [12]. A number of such frameworks exist both in academia [2, 4, 8, 18, 25] and in industry [1, 13, 16, 24, 28], and have been used for a variety of purposes, including diagnosing anomalous requests [2, 5, 17, 20]; diagnosing steady-state problems problems that manifest across many requests [8, 18, 20, 24, 25]; identifying slow components and functions [4, 14, 24]; and modelling workloads and resource usage [2, 14, 25].

Spectroscope [20] diagnoses performance changes by comparing sets of before- and after- traces, based on the request structures and/or timings. Their request sampling is made randomly when the request enters the system and, by default, captures 10% of all traces. Dapper uses a random sampling with less than 0.1% rate. Every day, their production clusters generate more than 1 TB of sampled trace data. Both Spectroscope and Dapper suggest storing traces for two weeks for post-hoc analyses before discarding them [19]. Since just one out of thousands of requests provides sufficient information for many common uses of the tracing data [24], we could use our representative sampling approach to select the most interesting traces and instead of discarding them, storing a small fraction in a long-term storage.

7 Discussion and Future Work

End-to-end tracing frameworks applied to modern distributed system generates high volumes of very rich data. The representative trace sampling problem we introduce in this paper can improve the usefulness of this data by reducing the amount of data while still preserving interesting and diverse traces, even if very infrequent. We present an initial solution that can work both offline and online in an efficient way. While our experiments are preliminary – performed at small scale in controlled experiment, and with a small amount of traces from a large company – the results encourage further investigation.

We aim to test our scheme with more real-world traces and to explore the costs and performance overhead of our approach. Since our solution is tail-based, the tracing metadata has to be propagated across all invocations. Thus, we intend to investigate the use of the prefix of the traces as input for the sampling approach.

We also aim to further explore different distance metrics, such as ones that take into account latency and importance of nodes, and the richer structure of the graphs, and how they impact zero-day bugs. Interesting options include considering the edges as components, or using the Weisfeiler-Lehman framework for efficient subgraph isomorphism [22].

In a broader context, our approach can be applied to online clustering scenarios where the goal is to keep as diverse a set of samples as possible, given a fixed budget.

References

[1] Apache HTrace [n. d.]. Apache HTrace. http://htrace.incubator.apache.org/. Accessed March 2018.

[2] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. 2003. Magpie: Online Modelling and Performance-aware Systems. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9 (HOTOS'03)*. USENIX Association, Berkeley, CA, USA, 15–15. http://dl.acm.org/citation.cfm?id=1251054.1251069

[3] Dimitri Bertsekas and Robert Gallager. 1992. *Data Networks (2nd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[4] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. 2007. Whodunit: Transactional Profiling for Multi-tier Applications. *SIGOPS Oper. Syst. Rev.* 41, 3 (March 2007), 17–30. https://doi.org/10.1145/1272998.1273001

[5] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. 2004. Path-based Failure and Evolution Management. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1 (NSDI'04)*. USENIX Association, Berkeley, CA, USA, 23–23. http://dl.acm.org/citation.cfm?id=1251175.1251198

[6] Angelo Coluccia, Alessandro D'Alconzo, and Fabio Ricciato. 2012. On the optimality of max–min fairness in resource allocation. *Annals of Telecommunications - Annales des Télécommunications* 67, 1 (Feb 2012), 15–26.

[7] Rodrigo Fonseca. 2008. *Improving Visibility of Distributed Systems through Execution Tracing*. Ph.D. Dissertation. EECS Department, University of California, Berkeley.

[8] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-Trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation (NSDI'07)*. USENIX Association, Berkeley, CA, USA, 20–20. http://dl.acm.org/citation.cfm?id=1973430.1973450

[9] Jaeger Tracing [n. d.]. Jaeger Tracing. https://www.jaegertracing.io/. Accessed May 2018.

[10] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Vekataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *26th ACM Symposium on Operating Systems Principles (SOSP '17)*.

[11] Ari Kobren, Nicholas Monath, Akshay Krishnamurthy, and Andrew McCallum. 2017. A Hierarchical Algorithm for Extreme Clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17)*. ACM, New York, NY, USA, 255–264. https://doi.org/10.1145/3097983.3098079

[12] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[13] lightstep [n. d.]. Lightstep. http://lightstep.com. [Online; accessed January 2017].

[14] Gideon Mann, Mark Sandler, Darja Krushevskaja, Sudipto Guha, and Eyal Even-Dar. 2011. Modeling the Parallel Execution of Black-box Services. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'11)*. USENIX Association, Berkeley, CA, USA, 20–20. http://dl.acm.org/citation.cfm?id=2170444.2170464

[15] Adam Oliner, Archana Ganapathi, and Wei Xu. 2012. Advances and Challenges in Log Analysis. *Commun. ACM* 55, 2 (Feb. 2012), 55–61. https://doi.org/10.1145/2076450.2076466

[16] opentracing [n. d.]. OpenTracing. https://goo.gl/aFgklW. [Online; accessed January 2017].

[17] Krzysztof Ostrowski, Gideon Mann, and Mark Sandler. 2011. Diagnosing latency in multi-tier black-box services. *LADIS* (2011).

[18] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. 2006. Pip: Detecting the Unexpected in Distributed Systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3 (NSDI'06)*. USENIX Association, Berkeley, CA, USA, 9–9. http://dl.acm.org/citation.cfm?id=1267680.1267689

[19] Raja R. Sambasivan, Rodrigo Fonseca, Ilari Shafer, and Gregory R. Ganger. 2014. *So, you want to trace your distributed system? Key design insights from years of practical experience.* Technical Report. Parallel Data Laboratory, Carnegie Mellon University.

[20] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. 2011. Diagnosing Performance Changes by Comparing Request Flows. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011*. Boston, Massachusetts, USA.

[21] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. 2011. Weisfeiler-Lehman Graph Kernels. *J. Mach. Learn. Res.* 12 (Nov. 2011), 2539–2561. http://dl.acm.org/citation.cfm?id=1953048.2078187

[22] Nino Shervashidze, SVN Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten M Borgwardt. 2009. Efficient graphlet kernels for large graph comparison.. In *AISTATS*, Vol. 5. 488–495.

[23] D. Siegmund. 1976. Importance Sampling in the Monte Carlo Study of Sequential Tests. *The Annals of Statistics* 4, 4 (1976), 673–684. http://www.jstor.org/stable/2958179

[24] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. *Google research* (2010).

[25] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R. Ganger. 2006. Stardust: Tracking Activity in a Distributed Storage System. *SIGMETRICS Perform. Eval. Rev.* 34, 1 (June 2006), 3–14. https://doi.org/10.1145/1140103.1140280

[26] Jan E. Trost. 1986. Statistically nonrepresentative stratified sampling: A sampling technique for qualitative studies. *Qualitative Sociology* 9, 1 (01 Mar 1986), 54–57. https://doi.org/10.1007/BF00988249

[27] W3C Propagation format [n. d.]. W3C Propagation format for distributed trace context: Trace Context headers. https://w3c.github.io/distributed-tracing/report-trace-context.html. Accessed May 2018.

[28] Zipkin [n. d.]. Zipkin. https://zipkin.io/. Accessed May 2018.