Automating instrumentation choices for performance problems in distributed applications with VAIF

Mert Toslali[★], Emre Ates[★], Alex Ellis[†], Zhaoqi Zhang[†],
Darby Huye[†], Lan Liu[†], Samantha Puterman[★], Ayse K. Coskun[★], Raja R. Sambasivan[†]
[★]Boston University, [†]Tufts University

ABSTRACT

Developers use logs to diagnose performance problems in distributed applications. However, it is difficult to know a priori where logs are needed and what information in them is needed to help diagnose problems that may occur in the future. We present the Variance-driven Automated Instrumentation Framework (VAIF), which runs alongside distributed applications. In response to newly-observed performance problems, VAIF automatically searches the space of possible instrumentation choices to enable the logs needed to help diagnose them. To work, VAIF combines distributed tracing (an enhanced form of logging) with insights about how response-time variance can be decomposed on the critical-path portions of requests' traces. We evaluate VAIF by using it to localize performance problems in OpenStack and HDFS. We show that VAIF can localize problems related to slow code paths, resource contention, and problematic third-party code while enabling only 3-34% of the total tracing instrumentation.

CCS CONCEPTS

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

KEYWORDS

performance, distributed tracing, distributed systems, logging

1 INTRODUCTION

Logs are the de-facto data source engineers use to diagnose performance problems in deployed distributed applications. However, it is difficult to know a priori *where* logs are needed to help diagnose problems that may occur in the future [23, 40–42]. Exhaustively recording all possible distributed-application behaviors is infeasible due to the resulting overheads. As a result of these issues, distributed applications can contain lots of log statements, but rarely the right ones in the locations needed to diagnose a specific problem [23, 41]. New performance problems cannot be diagnosed quickly because the detailed logs needed to locate their sources are not present.

Diagnosing problems observed in deployment requires the ability to customize logging choices during runtime. Two sets of complementary techniques allow for such customization: dynamic logging and automated control of logging choices. The former allows developers to insert new logs in pre-defined [10, 15, 23] or almost arbitrary locations [18] of an application. But, it can result in high diagnosis times because engineers must manually explore the vast space of possible logging choices to locate the source of the problem. Only after doing so can they identify the problem's root cause and fix it.

To reduce diagnosis times, researchers have developed automated techniques to choose the needed logs [8, 13, 20, 42, 43]. However, they focus on correctness problems, not performance, or are designed for individual processes, not distributed applications. For example, Log20 [42] helps diagnose non-fail-stop correctness problems by enabling logs to differentiate unique code paths. However, fast code paths need not be differentiated for performance problems, and slow ones may need additional logs to further pinpoint the problem source. Log^2 [13] identifies which logs provide insight into performance problems in individual processes. Its value is diminished for distributed applications because it is unaware of slow requests' workflows—i.e., the application processes involved in servicing them.

This paper's goal is to create a logging framework that automatically enables the logs needed to diagnose performance problems in request-based distributed applications. We find that the combination of three insights about the critical-path sections of requests workflows, distributed tracing (an enhanced form of logging), and requests' performance variance makes such a framework possible.

The insights are as follows. First, in many distributed applications, requests whose workflows are *expected* to have similar critical paths should perform similarly [30]. If they do not—i.e., they exhibit high response-time variance—the expectation is incorrect, and there is something unknown about their critical paths. This unknown behavior may be performance problems, such as slow functions, resource contention, or load imbalances. Second, distributed tracing captures graphs (traces) of requests' workflows with resolution equal to the number of logging points in the application. (Distributed tracing calls log points tracepoints.) Third, high response-time variance can be localized to sources of high variance within critical-path portions of requests' workflow traces, giving insight into where more tracepoints must be enabled to explain the unknown behavior. For problems that manifest as consistently-slow requests instead of high variance ones, a similar process that focuses on high-latency areas of critical paths can be used.

We present the design of the Variance-driven Automated Instrumentation Framework (VAIF). VAIF is comprised of a distributed-tracing infrastructure that allows tracepoints to be enabled or disabled and control logic that decides where to enable tracepoints based on the performance-variation insights. It uses various search strategies (e.g., binary search) to decide which tracepoints to enable. During normal operation, VAIF operates identically to distributed tracing today and generates traces with a default level of tracepoints enabled. When developers must diagnose why requests are slow, they "push a button" and VAIF automatically explores which additional tracepoints must be enabled to locate the problem source. Similar to dynamic instrumentation, VAIF's approach reduces the burden of deciding which logs to enable a priori. It additionally eliminates the manual effort required to search the space of possible tracepoint choices.

We implemented two prototype VAIFs for OpenStack [26] and HDFS [35] by modifying their existing tracing implementations. In both applications, we find that our prototypes can enable tracepoints to locate the sources of real and synthetically injected sources of variance and latency even when only a minimal number of tracepoints are enabled by default. We find that many real sources of variance and latency correspond to bug reports in developer mailing lists. Our prototypes only enable 3-37% of the tracepoints they could enable to localize these issues.

We present the following contributions.

(1) We identify requirements for any logging (or tracing) infrastructure must satisfy to address key challenges that limit their utility for localizing problems. We identify insights that allow these challenges to be addressed for performance problems related to slow code paths, code with unpredictable performance, and resource contention.

(2) Building on the insights, we present the design of VAIF, an automated instrumentation framework that combines distributed tracing with control logic that automatically enables the tracepoints needed to localize new performance problems. VAIF's control-logic components are modular and can be applied to different distributed applications instrumented with tracing without modifications. Our VAIF design includes elements to make it practically useful: a novel data structure for explaining VAIF's tracepoint decisions to developers and mechanisms to limit VAIF's explorations when resources available to tracing are constrained.

(3) We demonstrate the efficacy of our VAIF prototypes by using them to localize seven sources of high variance and consistently-slow performance in OpenStack and HDFS. We find that those localized regions correspond to bug reports in developer mailing lists. We demonstrate VAIF's control-logic algorithms scale to trace sizes observed in OpenStack, HDFS, DeathStarBench's social network [17], and a large Internet company.

2 TOWARDS AUTOMATION

This paper addresses key challenges that reduce the value of logging in helping diagnose performance problems. This section introduces these challenges and derives requirements that any logging framework must satisfy to address them. It describes key insights and how these requirements can be met by combining distributed tracing with control logic that focuses on requests' response-time variance. Our approach for satisfying the requirements focuses on requests' critical paths. It can help diagnose problems due to slow code paths or functions, resource contention, or problematic third-party code.

We start this section by providing a background of request-based distributed applications.

Request/response-based distributed applications: These applications consist of processes that coordinate to service requests that they receive from clients (e.g., read a file) or generate themselves (e.g., garbage collect some data). One or more processes may be logically grouped into services and/or service components to reflect a clean breakdown of functionality. Requests' *workflows* describe the order and timing of work done within and among the processes, services, or service components involved in servicing them. Requests' performance is characterized by their response times, which depend on their critical paths. A request's *critical path* is the highest latency concurrent path of its workflow that must complete before

a response is sent to the client. Critical-path latencies depend on algorithm runtimes, request parameters (e.g., read/write sizes in a storage system [35]), and the resources used by requests or available to them at the time of their execution.

2.1   Logging challenges

Past research has identified three challenges with logging that curtail its value for localizing problems. Such localization identifies the areas first or most affected by problems, giving developers strong starting points for their diagnosis efforts [14]. The challenges are: *1)* No perfect one-size-fits-all logs leading to a tussle between informativeness and overhead, *2)* Extremely large log search spaces, and *3)* Data overload leading to a needle-in-the-haystack problem.

We argue that these challenges must be addressed separately for correctness and performance problems. This is because the logs to help localize these two classes of problems have different goals. Logging for correctness problems aims to identify the first divergence from normal execution that will lead to problematic points in the code, such as failure locations [8, 39, 41, 43] or undesired outputs [42]. In contrast, logging for performance problems aims to identify areas of the code or resource conditions that lead requests' overall execution to be slow.

We describe the challenges below and identify requirements they impose on logging frameworks.

No perfect one-size-fits-all instrumentation leading to a tussle between logging generality and overhead: Past research has argued that the logs needed to localize the source of one problem may not be useful for others [23, 36, 41, 42]. The lack of one-size-fits-all logs leads to a tussle to identify which log statements are most useful and should be enabled by default. For example, Zhao et al. [42] state that Hadoop, HBase, and Zookeeper have been patched over 28,821 times over their lifetimes to add, remove, or modify static log statements embedded in their code. They also point out that the 2,105 revisions that modify logs' verbosity levels reflect the tussle between a desire to balance overhead and informativeness of log statements.

This challenge results in the following requirement:

R1: *Logging frameworks must allow logs to be enabled selectively by developers during runtime or must automatically enable logs in response to problems observed during runtime.*

Extremely large logging search spaces: Assume a distributed application that allows log points to be inserted or enabled at every function's entry, exit, and exceptional return. (This is similar to the distributed applications used by Mace et al. [23] and Erlingsson et al. [15].) Here, the possible locations where log statements can be enabled is a function of the number of procedures in the applications' code base and the number of machines on which the application executes.

Even modestly-sized distributed applications can have search spaces with 100s or 1000s of possible log points.

To address this scalability challenge, we refine *R1* to require logging frameworks to automatically enable tracepoints. We also add the following requirement:

R2: *Automated Logging frameworks must be capable of narrowing down the search space when exploring what logs are needed to localize a newly-observed problem.*

Data overload amounting to needle-in-a-haystack problem: Existing logging infrastructures capture voluminous amounts of data. For example, Facebook's Canopy, a distributed-tracing infrastructure captures 1.16 GB/s of trace data, and individual traces contain 1000s of tracepoints [19]. Problem diagnosis, even when the needed logs are present, can feel like trying to find a needle in a haystack [29].

This challenge is partially addressed by *R1 (automatically choose what instrumentation to enable)*. To avoid the needle-in-haystack problem for cases where there may be multiple problems in the application simultaneously, we add the following requirement:

R3: *Automated logging frameworks must be capable of explaining their logging decisions.*

2.2   Key insights

We discuss insights that let us address the requirements and discuss how the requirements are addressed next.

The first insight is that in many distributed applications, requests with similar critical paths—i.e., requests that are processed similarly by the distributed application—will have similar response times. Existing use of this insight involves using separate performance counters for different request types or API calls, such as READS and GET ATTRIBUTES in a distributed-storage application. Separate counters are used because there is an expectation that requests of different types will have very different critical paths and thus have different response times.
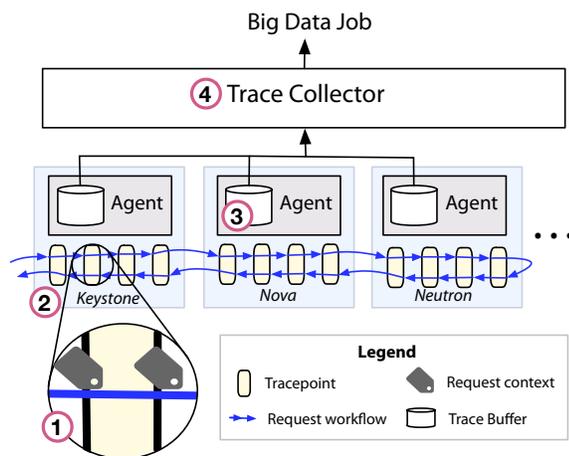
The second insight is that distributed tracing [28, 31, 33], which is an enhanced form of logging, can identify requests' critical paths with resolution equal to the amount of tracing instrumentation present. This is because it records graphs (called *traces*) of requests workflows. Today, distributed tracing is becoming increasingly popular and an ever-growing number of distributed applications are being instrumented with it [12, 19, 21, 26, 34, 35, 38].

This insight, combined with the previous one, means that that if requests' whose workflow traces have identical critical paths *do not* perform similarly—i.e., their response-time variance is high—there is some unknown behavior that is not captured in their traces. This behavior may represent performance problems, such as slow code paths or functions executing, differences in resources used or available to requests, poorly-written algorithms that unintentionally increase variance, or third-party code with unpredictable performance.

Figure 1 shows how most distributed-tracing infrastructures work. ① Tracing infrastructures propagate per-request *context*—unique request IDs and logical clocks—along with requests' execution (🖛 in the figure). ② They tag records of logging points executed by requests with requests' context. (Logging points that record context are called *tracepoints* and shown by ▯ in the figure.) ③ To avoid impacting performance, tracepoint records are cached in fixed-size memory buffers within local *tracing agents*. They are flushed to a centralized *collector* periodically or when the system is idle. ④ Asynchronously, a big-data collects tracepoint records from the collector and orders ones with the same request ID to create traces of requests' workflows.

Tracepoints contain a name describing the behavior they record (e.g., VM_LIST_START) in OpenStack or CACHE_MISS in a storage system. They also contain an arbitrary number of key/value pairs, which developers use to record request/function parameters or information about resources used/available at the time of requests' execution.

This paper represents request workflow traces as directed-acyclic graphs in which nodes are tracepoint names and edges are happens-before relationships between events. Graphs are labeled with requests' response times, and edges are labeled with inter-tracepoint latencies. Fan-outs in the graphs represent concurrent activity , and fan-ins represent synchronization. Hierarchical caller/callee relationships between groups of tracepoints are either represented by additional *hierarchical edges* or inferred by the nesting of START / END annotations within tracepoint names. This representation is sufficient to express relationships created by all current tracing infrastructures.

Figure 1: Distributed tracing architecture. Traces are being collected for a simplified version of OpenStack [26]. The blue line shows the workflow of a VM_LIST request.

The third insight is that the variance-sum law [37] can be applied to traces of requests' critical paths. For a set of requests whose trace critical paths appear identical as per the enabled tracepoints, this law can be interpreted as follows. The variance of requests' response times is the variance of the latencies of their critical-path trace edges plus their covariances.

This insight means that we can identify areas in the codebase in which unknown behavior resides by identifying the edges of requests' critical-path traces that contribute most to the variance. The unknown, potentially problematic behavior resides within the code regions that execute between the tracepoints that form these edges.

2.3 Addressing the requirements

Based on the insights, the requirements can be satisfied for many classes of performance problems by combining two technologies. The first is a distributed-tracing infrastructure that allows tracepoints to be selectively enabled or disabled during runtime. The second is a control mechanism for distributed tracing that (mostly) uses variance to guide which tracepoints to enable. We next describe the principles of this control mechanism and how they address the requirements. We use the term critical-path traces to refer to the critical-path portions of requests' workflows. We define identical critical path traces as those which execute the same tracepoints in the same order and whose nodes have identical names.

Principle #1: Identify requests whose traces exhibit identical critical paths but which exhibit high response-time variance. Identify edges of their traces' critical paths that contribute most to the variance. Enable additional tracepoints in the code regions corresponding to these areas.

Principle One differentiates slow code paths from fast ones and/or isolates code with unpredictable performance. It addresses *R1: enable tracepoints (logs) automatically in response to problems* and *R2: narrow down the search space.*

This principle is a direct application of the law of total variation to critical-path traces. Applying this law narrows down the search space to tracepoints that can execute between the critical-path trace edges that contribute most to response-time variance (*R2*). Enabling some tracepoints in this area adds them to future traces, either differentiating critical-path traces further to separate fast ones from slow ones or further isolating areas from which high variance arises.

Iteratively applying this principle until requests with identical critical-path traces exhibit low response-time variation or until no additional tracepoints can be enabled accomplishes the following. *1)* It sufficiently differentiates fast critical paths from slow ones or *2)* isolates high variance coming from black-box third-party code, problematic algorithms, or differences in resource usage/availability (*R1*).
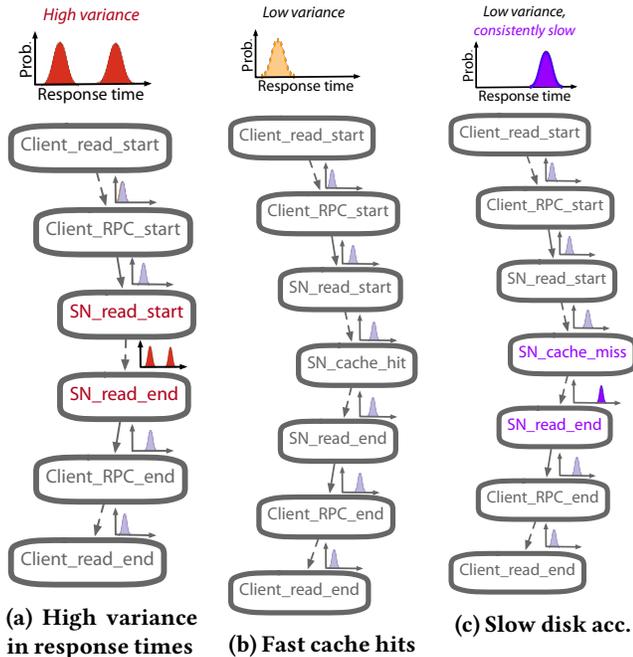
Example of differentiating slow critical paths from fast ones: Figure 2 shows three groups of identical critical-path

traces from a distributed-storage system (e.g., Ceph [38] or HDFS [35]). Nodes indicate tracepoint names, and edges indicate happens-before relationships. Edges are annotated with distributions of edge latencies. In Figure 2a, the response-time variance of READ requests with identical critical-path traces is high. The figure shows that the trace edge spanning storage node accesses is the dominant contributor to the variance. Figure 2b and 2c shows that enabling tracepoints to differentiate cache hits from misses distinguishes fast critical paths (cache hits) from consistently slow ones. (The figures show normal distributions, but our principles hold for arbitrary ones.)

Example of isolating high variance due to third-party code: Figure 3a shows that the response-time variance of DIRECTORY LIST requests with identical critical-path traces is high. The figure shows that the trace edge spanning the metadata server, which stores directory information, is the dominant contributor to the overall variance. Figure 3b shows that enabling tracepoints within the metadata server eventually reveals that the variance emanates from a third-party database that is storing the directory info.

Principle #2: Identify requests whose traces have identical critical paths, have low variance but have high response times (i.e., are *consistently slow*). Identify critical-path trace edges that are dominant contributors to response times and enable tracepoints in the code regions corresponding to these areas.

Principle Two localizes problems due to slow functions. It addresses *R1: automatically enable instrumentation* and *R2: narrow down the search space*. It is needed because Principle

(a) High variance

(b) Variance isolated to DB
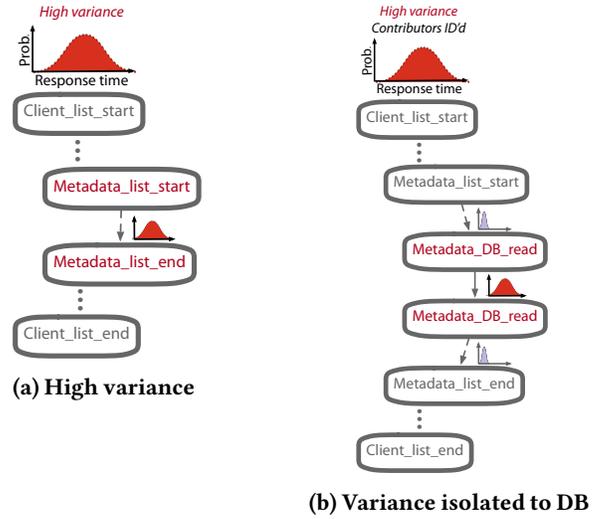
Figure 3: Applying Principle #1 to isolate high variance emanating from third-party code.

One identifies slow critical paths but does not localize slow performance to specific code areas. It is similar to the first principle, except it focuses on response times and edge latencies directly instead of variance.

Example: Requests corresponding to the critical path traces in Figure 2c have low variance but have high response times. The majority contributor to response times is the edge between the cache_miss tracepoint and the *SN_end* tracepoint. Iteratively isolating dominant contributors to overall response times and enabling tracepoints in these areas eventually identify the function(s) that contribute most to response times.

Principle #3: Identify requests whose traces exhibit identical critical paths but which exhibit high variance in their response times. Identify which key/value pairs exposed by already-enabled tracepoints correlate highly with requests' response time. Augment tracepoint names with these keys and ranges of their values or directly surface them to developers.

Principle Three localizes problems related to resource usage/availability. It also differentiates slow critical paths from fast ones when keys' values record how much work requests must perform (e.g., read/write sizes in a storage system). It is an enhancement to Principle One that explores reasons for variation that are not due to differences in critical paths themselves but rather due to external factors at the time of requests' execution.

Example: OpenStack VM_CREATE requests exhibit high variance. The edge that contributes most to variance is build_semaphore_start → build_semaphore_end. The first tracepoint of this edge exposes a "mutex_queue_length" key, whose values correlate highly with response times. Augmenting the "mutex_queue_length" key and ranges of

(a) High variance in response times   **(b) Fast cache hits**   **(c) Slow disk acc.**

Figure 2: Applying Principle #1 to differentiate fast critical paths from slow ones.

its values (e.g., 0-5, 6-10, > 10) differentiates critical paths as per their queuing times. This localizes the problem to high resource contention in this area.

Principle #4: Maintain a history of the tracepoints enabled on behalf of high variance or consistently-slow performance along with the statistics that motivated these decisions. This principle allows the framework to explain why it made the decisions it did to localize problems (*R3*).

Contrast to related work: Many logging frameworks do not automatically enable logs or tracepoints [15, 23] (*R̸1*) or are designed to automatically enable instrumentation for fail-stop correctness problems only [8, 39, 41–43]. Log^2 [13] partially addresses *R1* and *R2* by deciding which already-enabled logs to persist in storage, not which ones to enable in the first place. It cannot explain its decisions in terms of requests' performance (*R̸3*) because it is oblivious to requests' workflows or critical paths. Some logging frameworks [42] generically enable logs to differentiate unique code paths. This approach is neither sufficient nor necessary for performance diagnosis. It is insufficient because additional logs may be needed to identify where on a code path a problem lies; it is not necessary when code paths are fast and need not be differentiated.

3 VAIF

VAIF is an automated instrumentation framework that combines distributed tracing and control logic based on the principles. It is deployed alongside running distributed applications. In normal operation, VAIF operates identically to existing distributed tracing, generating traces using tracepoints that developers wish to have always on. These tracepoints may be ones developers have found useful in the past or ones used for use cases other than performance diagnosis, such as correctness. When new performance problems occur, developers can use VAIF to automatically enrich traces with the additional tracepoints needed to localize them.

VAIF localizes problems due to slow code or those with unpredictable performance (high variance). Such unpredictability may emanate from areas of the application itself, third-party code the application uses, or from areas of the application that could benefit from additional tracing instrumentation. VAIF also explores whether key/value pairs exposed in tracepoints explain high variance. It allows developers to specify important keys that they suspect will explain variance and bin ranges for them. VAIF will augment tracepoint names with these keys if they explain variance. It will surface other keys whose values explain variance in its output.

Like manual dynamic-instrumentation approaches [15, 23], VAIF frees developers from the tussle between generality and overhead. Unlike manual approaches, it also frees them from having to search the space of tracepoint choices to enable additional ones. When enabling instrumentation, VAIF works in a continuous cycle. At each iteration, it uses the principles

to hypothesize (guess) which tracepoints should be enabled next within a high variance or slow area of the application. It uses the results of previous hypotheses to guide future ones. It uses a novel data structure, called the *hypothesis forest*, to explain the results of its hypotheses to developers. VAIF's analyses are most useful for on-path problems. It also provides value for off-path problems by identifying the critical-path areas most affected by them.

The rest of this section describes VAIF in more detail. Our discussions are agnostic to whether tracepoints are enabled or disabled using dynamic instrumentation [10, 15, 23] or by modifying existing tracing infrastructures' tracepoints to execute conditional checks before emitting tracepoint records. We assume tracepoints are uniquely addressable. Some methods for addressing them include using: their names (as done for Linux kernel logs); hashes of their names and their keys (if names are not unique); an external registry that assigns addresses to tracepoints. We assume unique names from now on.
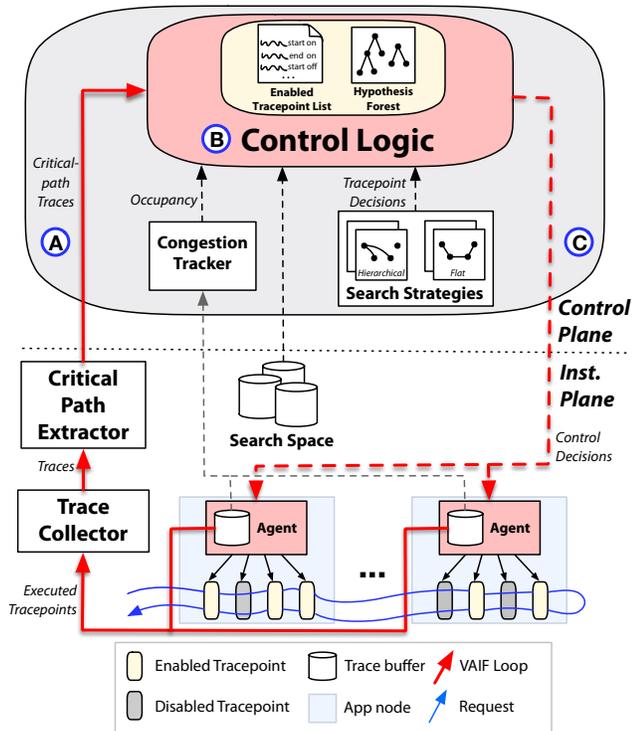
3.1 Design

Figure 4 shows VAIF's design, which builds upon existing distributed tracing. It consists of a control plane and an instrumentation plane. Components in the *control plane* implement the control logic whereas those in the *instrumentation plane* implement the control logic's hypotheses or provide custom information about the application.

VAIF works in a continuous loop, which is shown in *red* in the figure. At each iteration, VAIF's instrumentation-plane components gather new critical-path traces (Ⓐ in the figure). The control-plane components examine them to identify hypotheses of which tracepoints should be enabled next and which key/value pairs additionally explain high variance (Ⓑ). Hypotheses are sent to the instrumentation plane components (Ⓒ), which enable the relevant tracepoints and the cycle repeats. VAIF pauses its explorations if any of the tracing agents' queues are congested. This prevents cases in which VAIF does not observe the effects of new hypotheses because tracepoints records were dropped.

We next describe VAIF's key components and its inputs and outputs. Sections 3.2 and 3.3, further discuss the control logic, how (most of) the search space can be automatically constructed, and the search strategies.

3.1.1 Components.
Control plane: The control plane consists of the control logic, two search strategies for deciding which tracepoints to enable in high-variance or slow areas, and a congestion tracker that periodically receives queue occupancies from tracing agents. The search strategies are designed to be generically applicable to many distributed applications. The congestion tracker informs the control logic when any tracing agents' queues are in danger of being congested, which we

Figure 4: VAIF design. The thick red line shows VAIF's continuous loop. Solid lines are traces/tracepoints and dashed lines are control signals. A generic distributed application instrumented with tracing is shown in the instrumentation plane.

define as over 50% occupancy. We use this conservative definition because VAIF does not know how many times tracepoints will execute once enabled. The control plane also maintains important state: a global list of tracepoints that have been enabled by VAIF and the hypothesis forest.

VAIF's control-plane components are modular and intended to be used with different distributed applications and/or tracing infrastructures without modifications.

Instrumentation plane: The instrumentation plane consists of an application instrumented with tracing, a critical-path extractor that extracts critical-path portions from traces and sends them to VAIF's control-plane components, and a search space that describes the application's tracepoints. The critical-path extractor works by identifying the highest-latency trace path from the tracepoint indicating request reply to that indicating request start. Concurrency and synchronization may result in multiple paths for a single trace, each with different latencies. The search space names all of the tracepoints in the distributed application, including the keys that will be used in grouping. It also lists tracepoints' concurrency/synchronization tracepoint names as these must be enabled for critical-path extraction.

Legacy instrumentation-plane components require modifications to be used with VAIF. First, the tracing infrastructure's libraries must allow tracepoints to be selectively enabled or disabled during runtime. They must also let developers specify which tracepoints should be considered always on. Second, tracing agents co-located with processes must report queue lengths and receive updates about which tracepoints to enable or disable. Third, tracing infrastructures must preserve happens-before relationships between tracepoint records to allow critical paths to be extracted. This can be done by exposing APIs to capture them directly (as done by X-Trace [16, 22], Canopy [19], and Stardust [32]) or by learning them over a large number of traces (as done for traces that preserve only hierarchical caller/callee relationships, such as Dapper [24] and Artillery [11]).

3.1.2 Usage.

Starting VAIF's exploration: VAIF takes two inputs to start its explorations. The first is the application search space. The second is a list of tracepoints corresponding to start of execution of request types (or endpoints) that are experiencing problems. (We assume tracepoints that name the corresponding replies can be programmatically derived otherwise, they would need to be provided as well.)

VAIF also takes as input two optional parameters. The first is a threshold for identifying groups of critical-path traces that exhibit high variance, specified as a coefficient of variation (CV or σ/μ). We use CV for this unpredictability condition because it is a unitless measure that reflects the intuition that groups with high response-time spread compared to their mean are more unpredictable than those with low spread. The second is a threshold for identifying groups as consistently slow (CS). It is specified as a percentile of the relevant request type's response-time distribution. VAIF considers any group of traces that show either CV or mean latency greater than these thresholds as potential problems. Default values of : CV threshold = 10%, CS threshold = 95% are used if these optional parameters are not specified.

VAIF's output and how to use it: VAIF outputs new traces whose critical paths are enriched with the additional tracepoints needed to localize problems. Developers can query the hypothesis forest to identify why tracepoints observed in a given trace were enabled. For example, for a given trace, the forest might show that enabling a tracepoint around a cache differentiated critical paths and generated two new groups, increasing predictability (lower CV) for one group and isolating unpredictability (increasing CV) for the other group. Developers can also examine the hypothesis forest directly to identify groups of requests with high response-time variation or groups that are consistently slow.

Shutting down VAIF: Developers can shut down VAIF after they have diagnosed the problem at hand. Before

terminating, VAIF will disable all of the additional tracepoints it enabled.

3.2 Control logic & hypothesis forest

At each cycle, VAIF's control logic explores hypotheses of which tracepoints should be enabled to localize problems. Hypotheses themselves are of the form "differentiating traces by whether they include or a lack a newly-enabled tracepoint helps localize the problem." Localization amounts to 1) differentiating groups of identical critical-path traces with high variance, 2) isolating high-variance application areas within groups, or 3) isolating application areas that lead to consistently-slow performance. To explain its decisions, it maintains a history of its hypotheses and their outcomes in the hypothesis forest. We describe the hypothesis forest first, then how the control logic explores hypotheses and constructs the forest.

3.2.1 Hypothesis forest.

Figure 5 shows an example hypothesis forest. Each tree in the forest encodes hypotheses made on behalf of a different request type or endpoint that VAIF is initialized with (e.g., OpenStack's VM_LIST in the figure). This reflects the intuition that request type is a basic predictor of performance and that different request types may experience different problems that benefit from different tracepoints.

Nodes of hypothesis trees (hypothesis nodes) contain pointers to the results of applying hypotheses. Hypotheses result in two nodes, one for traces that include the enabled tracepoint and the other for ones in which it is absent. Each node includes a field that names the hypothesis tracepoint and whether it should be present or absent from traces (e.g., +☐ or ~☐ in the figure). The root node of each tree shows results for traces that include the request-type start tracepoint.

Results are: 1) groups of identical critical-path traces that either include or exclude the tracepoint and 2) any keys in included tracepoints that explain variance. Groups store important performance information needed for VAIF's analyses–a representative trace, response-time distributions of requests assigned to them, trace edge-latency distributions, and the number of requests assigned to each group. Tracepoints enabled by VAIF on behalf of other paths or trees are removed from traces before grouping. Such processing allows VAIF to measure the effects of each hypothesis independently w/o interference from other hypotheses. Always-on tracepoints are not removed as VAIF does not make hypotheses about them.

3.2.2 Control logic.

Algorithm VAIF control logic shows the pseudocode. We describe important aspects below.

Initialization (lines 2-9): Hypothesize() is initialized with a search strategy (*search*), statistical thresholds for identifying high variance and consistently slow groups (*cs* and *cv*), a
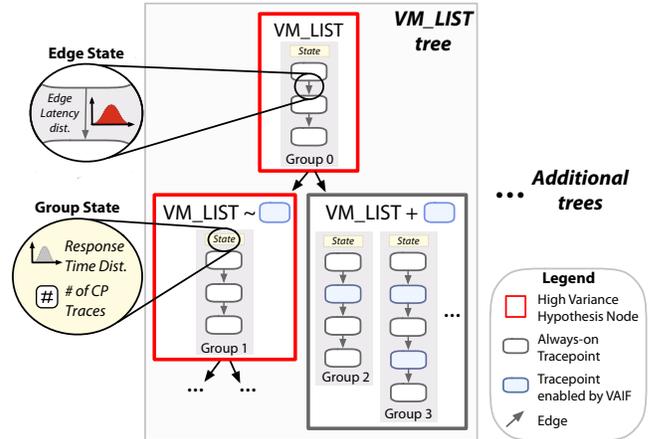

Figure 5: Example hypothesis forest.

set of mandatory tracepoints that must be enabled for VAIF to work (*mdtry*), and tracepoints that indicate the start and end of monitored request types' execution (*req_types*). Mandatory tracepoints include the concurrency and synchronization points listed in the search space and those in *req_types*. VAIF initializes the hypothesis forest with root nodes corresponding to the start tracepoints in (*req_types*) and enables the mandatory tracepoints if they are not always-on ones.

Checking for congestion: (lines 11- 13) The congestion tracker is consulted to check if any tracing agents' queue occupancies over 50%. Hypothesize() sleeps until this condition ceases to hold.

Consuming new traces (lines 14- 15): New critical-path traces observed in the interval between the previous cycle and the current one are added to the hypothesis forest's leaf nodes. The leaf to which to add a trace is identified by matching its tracepoints to hypothesis-tree paths. Once the leaf node is identified, the trace is processed to remove extraneous tracepoints and connect surrounding edges. Finally, the trace is added to the group that matches its (processed) critical path.

Key/value pairs (line 16): Groups are analyzed to determine if key/value pairs in tracepoints that were enabled in the previous cycle are correlated with groups' response times. The search space is consulted to identify the subset of the correlated keys that have also been specified by developers in the search space. Tracepoint names are augmented with these keys and the developer-specified bin ranges for them. (Names of tracepoints specified in the hypothesis nodes are not modified.) Remaining correlated keys are surfaced in affected groups' hypothesis nodes.

Identifying potential problems (lines 18-19): Leaves of the hypothesis tree are analyzed to identify which ones contain problematic groups. These are ones with the most number of groups that exceed the CV or CS threshold (*cv_gs* and *cs_gs*) respectively. Groups must contain enough samples for statistical confidence to be considered (30 in our implementation).

Algorithm VAIF control logic

1: **procedure** Hypothesize(*search*, *mdtry*, *req_types*)
2:   init *hyp*(*req_types*)   ▷ Hypothesis tree
3:   init *tps_enabled*   ▷ Enabled tracepoint list
4:   init *prev*   ▷ (+) Hypothesis nodes created in last cycle
5:   init *tps*   ▷ Trace points enabled in this cycle
6:   init *ct*   ▷ Congestion Tracker
7:   *cv* ← 0.1   ▷ CV threshold
8:   *cs* ← 95   ▷ Consistently-slow threshold
9:   enable(*mdtry*)
10:   **for** ;; **do**   ▷ Start Cycle
11:     **while** *ct*.congested_danger() **do**
12:       sleep(*cycle_time*)
13:     **end while**
14:     *traces* ← *collector*.get_new_traces()
15:     *hyp*.add_traces(*traces*)
16:     *search*.key_value(*prev*)
17:     *prev*.make_empty()   ▷ Only this cycle's results
18:     *cv_gs*, *cv_nodes* ← *hyp*.id_high_cv(*cv*)
19:     *cs_gs*, *cs_nodes* ← *hyp*.id_high_cs(*cs*)
20:     *tps*.add(helper(*cv_gs*, *cv_nodes*, *prev*, *VAR*))
21:     *tps*.add(helper(*cs_gs*, *cs_nodes*, *prev*, *LAT*))
22:     enable(*tps*, *tps_enabled*)
23:     sleep(*cycle_time*)
24:   **end for**
25: **end procedure**
26: **procedure** Helper(*groups*, *hyp_nodes*, *prev*, *type*)
27:   init *tps*   ▷ Chosen tracepoints
28:   **for** *i* = 1…*length*(*groups*) **do**
29:     *tp* ← search.find((*groups*[*i*], *type*)
30:     *prev*.add(*hyp_nodes*[*i*].add_child(+*tp*))
31:     *hyp_nodes*[*i*].add_child(∼*tp*)
32:     *tps*.add(*tp*)
33:   **end for**
34:   **return** *tps*
35: **end procedure**

Generating new hypotheses: (lines 20-21) The search strategy is called to suggest tracepoints to enable for problematic groups (*search.find*()). The strategy uses group's edge-latency distributions to decide where a new tracepoint should be enabled. For a high CV group, it chooses the edge that contributes most to the overall variance. For a consistently-slow group, it chooses the edge with the largest mean latency. New nodes are created in the hypothesis forest to test inclusion or absence of the selected tracepoints in future traces. (New nodes are only created if they don't already exist as children of the relevant parent hypothesis node.)

Enabling tracepoints and sleeping (lines 22-23): The enabled tracepoint list is updated with the tracepoints selected by the search strategy and is replicated to the tracing agents. The control loop sleeps for a pre-determined duration to allow new traces to be gathered.

Stopping condition for problematic groups: The most granular tracepoints are already enabled within edges that account for the majority (≥50%) of overall variance or latency.

3.3   Search space & search strategies

Search space: The search space contains: 1) potential critical paths in requests' workflows when all tracepoints are enabled; 2) tracepoints that demarcate concurrency or synchronization; 3) keys within tracepoints that should be incorporated in grouping and bin ranges for them (needed for *search.key_value*()). All tracepoints are labeled with whether they are always on. Search strategies use the search space to suggest the next tracepoint to enable between high-variance or -latency trace edges (i.e., *search.find*()). This requires *matching* the group critical-path trace representative against potential critical paths in the search space to determine which path it is most likely to be.

Potential critical paths: These are learned by running exhaustive workloads against the application with all tracepoints enabled. For traces with multiple concurrent branches, each unique path within them is stored separately as a potential critical path. We expect potential critical paths will be learned as part of organizations' code or coverage tests. These tests can be run when deploying an application or updating it. VAIF will still provide value if the paths stored in the search space are incomplete.

Developer-specified keys and bin ranges: These are stored as key name, tracepoint to which they belong, and a set of bin ranges. We expect developers will specify only a few select keys (e.g., ones that correspond to request sizes or ones that indicate queue lengths for a highly-contended resource.)

Concurrency / synchronization points: These are identified automatically as fan-out and fan-in points in learned paths.

Matching to potential critical paths: For a given group, we transform its representative trace into a regular expression. For example, the trace "A->B->C" is transformed into ".*A.*B.*C.*". The expression is applied to all potential critical paths to identify potential matches. We choose among multiple matching potential critical paths randomly weighted by frequency of their occurrence during learning.

Search strategies: The two search strategies either use the hierarchical or happens-before edges in traces.

Hierarchical search: This strategy uses hierarchical edges (called spans in OpenTelemetry [27]). Given a pair of tracepoints that denote a hierarchical level (a span), this strategy determines which pair of tracepoints (subspan) to enable next. In the case where there are multiple subspans, we use binary search to pick the middle one.

Flat search: This strategy uses only the happens-before relationships. Given a pair of tracepoints, this picks a

tracepoint that occurs between them. Tracepoints are ordered by happens-before relationships and binary search is used to pick among them.

3.4 Limitations & discussion

Limitations of using current tracing infrastructures: VAIF cannot identify if the observed variance is a result of the application code itself or code in lower layers (e.g., kernel). This is because current tracing infrastructures are limited to capturing elements of requests' workflows in the application only. In some cases, key/value pairs exposed within tracepoints can provide insights into variance coming from lower layers (e.g., flow IDs for the network).

Limitations of push-button, statistical approach: VAIF cannot provide value for transient or infrequent problems that disappear before developers use VAIF. Many existing tracing infrastructures cannot collect traces for these problems because they use a low trace sampling rate [31]. VAIF's approach of enabling tracepoints only when needed may allow for higher sampling rates, allowing traces of these problems to be collected with always-on tracepoints.

Potential for combinatorial explosion: Such potential exists when VAIF is used with applications that have many potential critical paths. In such cases, VAIF's search strategies will have to match against a very large number of paths to choose tracepoints. VAIF's hypothesis forest may contain an excessively large number of groups if VAIF must enable many tracepoints or if many tracepoints are always on. § 5.2 shows that matching time is not an issue for the applications and workloads we use. VAIF mitigates explosion in groups by exploring only one hypothesis (tracepoint) at a time per branch in the hypothesis forest and processing traces to remove tracepoints irrelevant to the current hypothesis.

Dealing with asynchronous design patterns: VAIF requires additional help to provide value for asynchronous design patterns where the latencies of interest are not reflected in critical paths' response times. Developers must establish additional tracing edges to stitch together asynchronous patterns' paths (e.g., as done using OpenTelemetry's `link` relationship). They must also identify tracepoints whose timestamps capture the latency of interest. We employed this tactic to capture paths for asynchronous OpenStack requests, such as VM_CREATE and VM_DELETE.

4 VAIF PROTOTYPES

We wrote two prototype VAIF implementations for Open-Stack [26] and HDFS [35]. Both implementations use the same control-plane components, which amount to 7k LoC in Rust. Prototypes' tracing infrastructures use modified versions of these applications' existing tracing infrastructures (OSProfiler [25] and X-Trace [16] for OpenStack and HDFS respectively). Our prototypes address tracepoints by either unique name (HDFS) or a combination of name and line number in code (OpenStack).

Modifications to OSProfiler and OpenStack: We wrote a python-based tracing agent that uses the filesystem to store the enabled tracepoint list. We modified the decorator function OSProfiler places around select class objects to check the tracepoint list before emitting tracepoint records. These modifications total 80 LoC. A tracepoint incurs 0.1ms for its conditional check, a significant portion of which is due to Python itself. This overhead is much lower than OpenStack's response times, which are on the order of 10s of seconds. We additionally modified OSProfiler to capture concurrency and synchronization and link together OpenStack's asynchronous calls.

Modifications to X-Trace: We modified (50 LoC) the X-Trace logger library to implement the conditional check. A tracepoint incurs 0.05ms of overhead for its conditional check due to VAIF. HDFS requests take on the order of seconds to execute. HDFS instrumented with X-Trace already captures concurrency and synchronization.

Creating search spaces: We collected search-space paths using workload generators that we built. Our OpenStack workload generator uniformly issues a mixture of CREATE, LIST, and DELETE request types for the same set of volume, server, and floating IP resources. The workload runs for a specified number of rounds and chooses a randomly selected number of concurrent workers per round (between 5-25). For HDFS, we use a similar workload generator parameterized to issue file writes (1MB), varying sizes of file reads (10kb, 100kb, 1MB) and LIST requests for randomly selected files. Experimental evaluation sections (§5 and §6) use the same workload generators.

5 EXPERIMENTAL EVALUATION

We evaluate our prototype implementations of VAIF using two applications (OpenStack and HDFS) and two trace datasets (a large Internet company and DeathStarbench's Social Network [17]). We seek to answer the following questions: *(1)* How scalable are VAIF's search space and matching process? *(2)* How big are traces when only the tracepoints needed for VAIF to start its explorations are enabled? *(3)* How do different search strategies perform with problems injected in various locations? *(4)* How many tracepoints does VAIF enable without any injected problems?

5.1 Experimental setup

Benchmark Applications.

OpenStack: This is a widely-used distributed application for managing clouds. We use the OpenStack Stein release. Our cluster consists of 9 Compute and 1 Controller node.

HDFS: This is a widely-used distributed-storage application. It exhibits complex and highly concurrent behaviors.

System	# of Unique Tracepoints	# of Unique / Overall Paths in Search Space	Min/Mean/Max Path Length	Search Space Size (MB / # Tracepoints)	Construction Time (s)	Time to Match (μs)
OpenStack	296	46 / 561	8 / 515 / 911	4.8 / 23694	0.7	730
Internet company	4194	2815 / 27196	2 / 10.3 / 2060	3.8 / 28945	13.3	499
Deathstar	413	103 / 3569	20 / 46.6 / 133	8.2 / 48070	11.2	230
HDFS	111	1032 / 78832	208 / 513 / 573	54.0 / 529468	470.6	7123

Table 1: Search statistics. The results show that VAIF's search space and matching can be used for all four systems.

We use the latest version of HDFS that includes X-Trace (2.7.2). Our HDFS cluster has 6 DataNodes and 1 Namenode.

Large Internet Company traces: The 18k traces we have access to are from a single region. The traces are obfuscated, so we use them only for scalability experiments (§ 5.2).

Social Network traces: This open-source dataset is collected from a version of the Social Network application that is instrumented with X-Trace [7]. It contains 8,000 traces consisting of COMPOSE_POST, REGISTER, and FOLLOW requests.

Infrastructure. All experiments are run on 9 CloudLab nodes running Ubuntu 18.04 and Linux Kernel 4.15.0. Each node has 8-core CPUs and 64 GB of memory.

VAIF. Our prototypes use CV threshold = 10% and CS threshold = 95% percentile. We use 30 seconds as the cycle time. VAIF only makes decisions for groups that have accumulated 30 samples.

5.2 Scalability of search

Methodology: We analyze 1) the scalability of VAIF's search space on four different trace datasets, and 2) the overhead of VAIF's control plane in terms of matching durations. To construct the search space's potential critical paths from OpenStack and HDFS, we enable all of the tracepoints and run the same workload generators discussed in the implementation (§ 4). We use all of the traces in the Social Network and Internet Company datasets to construct their search spaces.

Results: Table 1 presents the results based on the ratio of unique to total potential critical paths, search space size, and time to match. (For brevity, potential critical paths are listed as paths in the Table.) The number of unique tracepoints and the trace-path lengths vary depending on the system. The ratio of unique to total potential critical paths is very low across all applications, with large Internet Company exhibiting the highest ratio (0.05). VAIF exploits this repetitive behavior by keeping only unique potential critical paths in the search space.

We find that matching time, which dictates the minimum quanta at which VAIF can make decisions, is on the order of 100s of microseconds for most of the applications. The large Internet Company results show that VAIF accommodates large traces from real production workloads and the matching time is only 500μs. It is higher for HDFS (7ms) due to high concurrency that results in numerous paths even though the number of unique tracepoints is low. In practice, decision periods would be in the order of seconds or minutes to collect

a sufficient number of request traces, thus microsecond to millisecond lower limits for matching are negligible.

5.3 Minimal trace sizes

Methodology: We use our workload generators to collect traces from OpenStack and HDFS with only mandatory tracepoints enabled—i.e., those that indicate request-type start/end and concurrency/synchronization. (We call such traces *workflow skeletons*.) We compare these to traces collected by unmodified OpenStack's OSProfiler and HDFS's X-Trace, which have all tracepoints enabled by default. (We call unmodified OSProfiler or X-Trace *Vanilla tracing* and their traces *Vanilla traces*.)

Results: Figure 6 shows average trace sizes for different request types' traces. Skeletons are 95% smaller on average across all request types. Most request types' execution is sequential, so the skeletons only have tracepoints for request start/end. Other request types exhibit concurrent behavior (e.g., WRITE in HDFS), requiring additional tracepoints in the skeleton to demarcate concurrency and synchronization.

(a) OpenStack.

(b) SocialNetwork.   **(c) HDFS.**
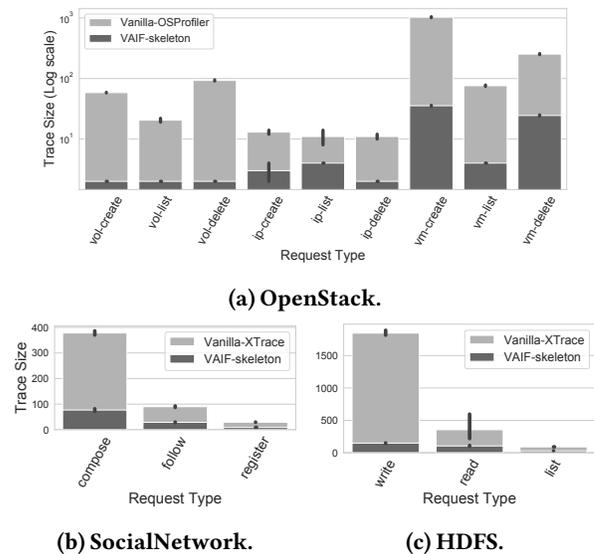
Figure 6: Skeleton vs. Vanilla. OpenStack's results are shown in log scale due to the large difference between skeleton and vanilla trace sizes for this application.

5.4 Comparison of search strategies

Methodology: We run VAIF with a delay injected into a location in OpenStack and compare Hierarchical and Flat searches' efficacy to localize it. We consider the most challenging case for the search strategies: 1) VAIF starts with only mandatory tracepoints enabled and so has little initial guidance about where to enable tracepoints; 2) delays are injected to affect VM_CREATE requests, which generate the largest traces and execute the most number of unique tracepoints compared to other OpenStack requests. We repeat the experiment 10 times with delay locations chosen randomly and delay latencies chosen from a normal distribution ($\mu = 5s$ and $\sigma = 30$). VAIF is re-initialized between each run.

Results: Figure 7a shows the average number of tracepoints enabled to localize delayed regions for both search strategies. Both search strategies localize delays with 100% precision. Delays may reside in areas that are far from mandatory tracepoints and/or in hierarchically deep locations in the codebase. Yet, flat search enables only 10 tracepoints on average and hierarchical only enables 15. We observe that Flat search performs better than Hierarchical search. Figure 7b additionally compares sizes of VM_CREATE traces generated by both search strategies to Vanilla OSProfiler tracing. We find that VAIF's search strategies reduce trace sizes by 89% on average.
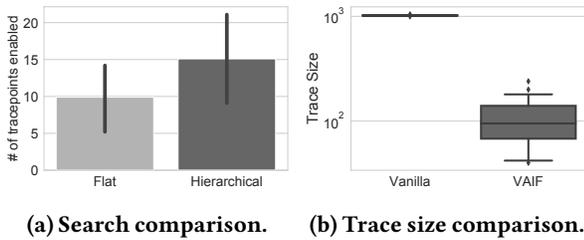

(a) Search comparison.   **(b) Trace size comparison.**

Figure 7: Comparison of search strategies.

5.5 Nominal operations

Methodology: We analyze the sizes of traces VAIF generates in OpenStack without any injected problems. We compare the result with the Vanilla OSProfiler tracing (all tracepoints enabled). We start VAIF with only mandatory tracepoints enabled and let it run until there are no problematic groups or VAIF localizes the problems with most granular tracepoints enabled. As per our findings in (§ 5.4), we use the Flat search strategy from now on.

Results: Figure 8 shows the distribution of average trace sizes for nine different request types on OpenStack (same as § 5.3). When enabling instrumentation to differentiate critical paths or isolate the code areas that contribute most to variance or response times, VAIF enables significantly fewer tracepoints than Vanilla OSProfiler. As a result, VAIF reduces trace sizes for different request types by 90% on average.
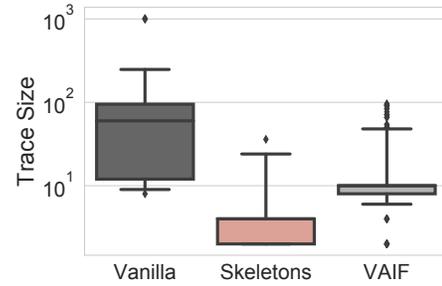

Figure 8: Average per-request-type Vanilla, Skeleton, and VAIF trace sizes under nominal operation.

6 CASE STUDIES

VAIF serves as a fundamental step for developers to diagnose performance problems, localizing them to a specific area of the system. We envision that developers will use VAIF in two ways; 1) match anomalous traces (e.g., slow requests) to the hypothesis forest to figure out the performance hypotheses, 2) query the hypothesis forest by the request type to inquire about the sources of performance problems.

This section presents seven case studies of how we used VAIF to identify performance problems in OpenStack and HDFS. All but one of the case studies were discovered by examining the output of the nominal-run experiment (§ 5.5). One problem was re-injected to show effectively it can be localized using VAIF. Table 2 presents an overview of the case studies and VAIF's output for them. The table includes one additional metric, correlation coefficient (R), which is the correlation between the problematic group's response times and localized edge's latencies. We discuss four of the case studies below.

Case 1. In OpenStack, instances can be deleted using the VM DELETE command. We find that some VM DELETE requests take 28 seconds, where the mean latency is around 20. We query the hypothesis forest by the slowest request. VAIF finds the delete groups have high CV values (0.18) to begin with. In exploration of variance, VAIF enables three tracepoints and it localizes the high variance to *unplug_os_vif* function. This function constitutes 85% of the variance and 42% of response time in this request type. Our analysis from there leads us to a bug report about the performance of this function ([5, 6]). For this problem, VAIF helps diagnose a performance bug due to a delay in a function.

Case 2&3. All instances on OpenStack can be listed using the command VM LIST. Matching the slowest trace to the hypothesis forest shows that the request's latency emanates from three edges. This trace's group shows high CV (0.2), and the enabled tracepoints constitute 63% of all variance and 60% of the latency. We further examine the code corresponding to those three edges and find the following; 1) two edges (*keystone_post&get*) correspond to where identity service (keystone) is utilized for authentication token, 2) the third

Case	App	Localized to	Description	CV	$\propto\sigma^2$	$\propto\mu$	R	δ
1	OS	Unpredictable perf. (lib.)	OS-vif library shows latency variation [5, 6]	0.12	85%	42%	0.74	85%
2	OS	Unpredictable perf. (service)	Identity service degrades by entries [4]					
3	OS	Unpredictable perf. (impl.)	Lack of instrumentation in a long function [3]	0.2	63%	60%	0.76	75%
4	OS	Unpredictable perf. (lib.)	Inefficient implementation in libvirt driver [1, 2]	0.12	85%	30%	0.86	96%
5	OS	Resource Contention	Too low limit on simultaneous vm creations [9]	0.21	96%	73%	0.98	97%
6	OS	Slow codepath	Consistently slow workflows in ip-create requests	0.3	81%	65%	0.96	66%
7	HDFS	Unpredictable perf. (impl.)	Retry mechanism in code	0.11	86%	20%	0.94	77%

Table 2: Case studies. We report the localization descriptions and following measures; coefficient of variation (CV) of the problematic group, variance contribution of the localized edge ($\propto\sigma^2$), response-time contribution of the localized edge ($\propto\mu$), correlation between edge and request latency (R), and reduction in traces (δ) compared to Vanilla tracing. OS corresponds to OpenStack.

edge corresponds to a function (*get_all*) that constitutes 2000 LoC and performs numerous DB lookups to get every instance, including deleted ones. We corroborate these findings in the bug reports ([3, 4]), which state that VM List experiences latency variations due to a) the token table getting large in identity service, and b) the function not being able to scale well with the number of VMs and users. In this case, VAIF helps diagnose performance problems by isolating latency to (1) a specific service and operation and (2) an inefficient function. The latter case also provides an insight to developers as inefficient tracing (i.e., more tracepoints can be added to the 2000 LoC).

Case 5. We re-inject the resource-contention problem described in Section 2.3. The root cause is a configuration flag that artificially limits the number of concurrent VM_CREATE requests. VAIF finds that VM CREATE groups have high CV (0.21) initially. VAIF finds that the highest-variance edge constitutes 96% of the variance and 73% of the response time. The first tracepoint of the highest variance edge (*_build_semaphore_start*) is where a nova-compute manager semaphore is acquired. Further, VAIF finds a high correlation between a key-value pair exposed by the tracepoint (i.e., the number of simultaneous VM creations) and request's latency (R=0.85 w/ p-val 10^{-5}).

7   RELATED WORK

Dynamic instrumentation: Much work has focused on allowing logs or customizable probes to be inserted in applications at arbitrary or pre-determined locations [10, 15, 23]. These tools provide a powerful basis to build automated approaches to customizing instrumentation. VAIF can leverage these approaches to enable tracepoints. PivotTracing [23] is designed to correlate monitoring data collected for specific requests across the processes involved in their workflows. It could be a good fit to VAIF if the query language it exposes is extended to support VAIF's specific use case.

Automatically customizing instrumentation during runtime: Past approaches explore how to customize instrumentation to diagnose problems [8, 20, 42, 43]. But these methods generally focus on correctness, not performance

and are not designed for distributed applications.

Enhancing instrumentation offline: Many existing techniques aim to enhance instrumentation offline before applications are run [40, 41]. For example, Yuan et al. identify what extra information is needed in existing logs to help diagnose failures [41]. In another paper, Yuan et al. [40] create a tool that automatically inserts instrumentation at key locations determined by an analysis of common problems types. VAIF complements these methods by helping diagnose new, unanticipated performance problems during runtime.

Discarding instrumentation: Log^2 [13] collects logs that span semantically meaningful intervals and discards ones with low latency or variance. Its after-the-fact approach allows it to make less greedy choices than VAIF, but it requires all logs to be enabled in the first place. Its choices of which logs to keep might be both more indiscriminate and more useful than VAIF's choices. It may be more indiscriminate because it is unaware of requests' workflows and thus may enable logs in areas of the application that are not performance sensitive. It may be more useful for off-critical-path problems.

8   SUMMARY

It is difficult to know where logs must already be enabled to help debug performance problems that may occur in the future. This paper presents the design of VAIF, which combines distributed tracing and variance-based control logic to automatically explore which tracepoints to enable. We demonstrated the efficacy of VAIF's implementation by using it to diagnose problems in OpenStack [26] and HDFS [35].

REFERENCES

[1] 2016. Instance stuck resuming from suspend state during load test. https://bugzilla.redhat.com/show_bug.cgi?id=1425516.

[2] 2016. nova libvirt driver instance stuck. https://ask.openstack.org/en /question/91508/nova-libvirt-driver-instance-stuck-on-spawning/.

[3] 2016. Nova list is extremely slow with lots of vms. https://bugs.launchpad.net/nova/+bug/1160487.

[4] 2016. OpenStack Identity service is responding slowly. https://docs.openstack.org/operations-guide/ops-maintenance-slow.html.

[5] 2016. OVS plugin VIF plugging slow for VMs. https://ask.openstack.org/en/question/128160/ovs-plugin-vif-plugging-slow-for-vms-with-multiple-nics/.

[6] 2016. Update on os-vif progress. https://openstack-dev.openstack.narkive.com/gLpnyrJl/nova-neutron-update-on-os-vif-progress-port-binding-negotiation.

[7] V. Anand and J. Mace. 2021. X-Trace DeathStarBench Dataset. https://gitlab.mpi-sws.org/cld/trace-datasets/deathstarbench_traces.

[8] Piramanayagam Nainar Arumuga and Ben Liblit. 2010. Adaptive bug isolation. In *International Conference on Software Engineering*. ACM Press, New York, New York, USA, 255–264.

[9] Emre Ates, Lily Sturmann, Mert Toslali, Orran Krieger, Richard Megginson, Ayse K. Coskun, and Raja R. Sambasivan. 2019. An Automated, Cross-Layer Instrumentation Framework for Diagnosing Performance Problems in Distributed Applications. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 165–170. https://doi.org/10.1145/3357223.3362704

[10] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. 2004. Dynamic instrumentation of production systems. In *ATC '04: Proceedings of the 2004 USENIX Annual Technical Conference*.

[11] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. 2014. The mystery machine: end-to-end performance analysis of large-scale internet services. In *OSDI' 14: Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*.

[12] CockroachDB. 2019. https://www.cockroachlabs.com/.

[13] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. 2015. Log^2: A cost-aware logging mechanism for performance diagnosis. In *ATC '15: Proceedings of the 2015 USENIX Annual Technical Conference*.

[14] Kiciman Emre and Lakshminarayanan Subramanian. 2005. Root cause localization in large scale systems. In *Root cause localization in large scale systems*, Vol. Proc. 1st Workshop on Hot Topics in Systems Dependability (HotDep).

[15] Ulfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. 2011. Fay: extensible distributed tracing from kernels to clusters. In *SOSP '11: Proceedings of the 23nd ACM Symposium on Operating Systems Principles*.

[16] Rodrigo Fonseca, Michael J. Freedman, and George Porter. 2010. Experiences with tracing causality in networked services. In *INM/WREN '10: Proceedings of the 1st Internet Network Management Workshop/Workshop on Research on Enterprise Monitoring*.

[17] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013

[18] Sudhanshu Goswami. 2005. https://lwn.net/Articles/132196/.

[19] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An end-to-end performance tracing and analysis system. In *SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles*.

[20] Ben Liblit, Alex Aiken, Alice X Zheng, and Michael I Jordan. 2003. Bug isolation via remote program sampling. In *PLDI '03: Programming Language Design and Implementation*. ACM.

[21] Jonathan Mace. 2017. *End-to-End Tracing: Adoption and Use Cases*. Survey. Brown University. https://cs.brown.edu/~jcmace/papers/mace2017survey.pdf.

[22] Jonathan Mace and Rodrigo Fonseca. 2018. Universal context propagation for distributed system instrumentation. In *EuroSys'18: Proceedings of the Thirteenth EuroSys Conference*.

[23] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: dynamic causal monitoring for distributed systems. In *SOSP '15: Proceedings of the 25th Symposium on Operating Systems Principles*.

[24] 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 Workshop on Hot Topics in Cloud Computing*.

[25] Mirantis OSProfiler [n.d.]. OSProfiler. https://docs.openstack.org/osprofiler/latest/.

[26] Openstack [n.d.]. OpenStack web site. https://www.openstack.org.

[27] OpenTelemetry website [n.d.]. OpenTelemetry website. http://opentelemetry.io/.

[28] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, and Rebecca Isaacs. 2020. *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media.

[29] A. Rabkin and R. H. Katz. 2013. How Hadoop Clusters Break. *IEEE Software* 30, 4 (2013), 88–94.

[30] Raja R. Sambasivan and Gregory R. Ganger. 2012. Automated diagnosis without predictability is a recipe for failure. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing*. USENIX Association, 21–21.

[31] Raja R. Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H Sigelman, Rodrigo Fonseca, and Gregory R. Ganger. 2016. Principled workflow-centric tracing of distributed systems. In *SoCC '16: Proceedings of the Seventh Symposium on Cloud Computing*.

[32] 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 *NSDI'11: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*.

[33] Yuri Shkuro. 2019. *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. Packt Publishing Ltd.

[34] Benjamin H. Sigelman, Luiz A. Barroso, Michael Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a large-scale distributed systems tracing infrastructure*. Technical Report dapper-2010-1. Google.

[35] The Apache Hadoop Distributed File System 2013. The Apache Hadoop Distributed File System. http://hadoop.apache.org/hdfs/.

[36] Marc-André Vef, Vasily Tarasov, Dean Hildebrand, and André Brinkmann. 2018. Challenges and Solutions for Tracing Storage Systems: A Case Study with Spectrum Scale. *ACM Transactions on Storage (TOS)* 14, 2 (May 2018), 18–24.

[37] Larry Wasserman. 2010. *All of Statistics: A Concise Course in Statistical Inference*. Springer Publishing Company, Incorporated.

[38] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell D E Long, and Carlos Maltzahn. 2006. Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*.

[39] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: error diagnosis by connecting clues from run-time logs. In *ASPLOS '10: Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and*

Operating Systems.

[40] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be conservative: enhancing failure diagnosis with proactive logging. In *OSDI' 12: Proceedings of the 10th conferences on Operating Systems Design & Implementation.*

[41] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2012. Improving software diagnosability via log enhancement. *ACM SIGPLAN Notices* 47, 4 (June 2012), 3–14.

[42] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles.*

[43] Zhiqiang Zuo, Lu Fang, Siau-Cheng Khoo, Guoqing Xu, and Shan Lu. 2016. Low-overhead and fully automated statistical debugging with abstraction refinement. In *OOPSLA '16: Proceedings of the ACM international conference on Object oriented programming systems languages and applications.*