VAIF: Variance-driven Automated Instrumentation Framework

Mert Toslali[⋆]*, Emre Ates*[⋆]*, Darby Huye*[†]*, Alex Ellis*[†]*,*
Zhaoqi Zhang[†]*, 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. But, 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 summarize our work on 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.

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 [16, 32–34]. 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 [16, 33]. 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 customizing 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 [4, 9, 16] or almost arbitrary locations [11] 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 root cause and fix it.

To reduce diagnosis times, researchers have developed automated techniques to choose the needed logs [3, 7, 13, 34, 35]. However, they focus on correctness problems, not performance, or are designed for individual processes, not distributed applications. For example, Log20 [34] 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 need additional logs to further pinpoint the problem source. Log^2 [7] 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.

In Toslali et al. [27], we presented the Variance-driven Automated Instrumentation Framework (VAIF). It is a logging framework that automatically enables the logs needed to diagnose performance problems in request-based distributed applications. We found 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 made VAIF possible.

The insights are as follows. First, in many distributed applications, requests whose workflows are *expected* to have similar critical paths should perform similarly [21]. 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.

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(s). Similar to dynamic instrumentation, VAIF's approach reduces the burden of deciding which logs to enable a priori. It also eliminates the manual effort required to search the space of possible tracepoint choices.

We implemented two prototype VAIFs for OpenStack [18] and HDFS [26] by modifying their existing tracing implementations. In both applications, we found that our prototypes can enable tracepoints to locate the sources of real and synthetically injected sources of variance and latency. We found that many real sources of variance and latency correspond to bug reports in developer mailing lists. Our prototypes only enabled 3-37% of the tracepoints they could enable to localize these issues.

The rest of this paper summarizes Toslali et al. [27]. We focus on motivating the need for automated instrumentation frameworks (Section 2), VAIF's design (Section 3), and a short overview of our evaluation (Section 4).

2 Toward automated logging choices

This section introduces challenges in logging to help diagnose performance problems. It derives requirements that any instrumentation framework should satisfy to address the challenges. It describes how these requirements can be met by combining distributed tracing with control logic that focuses on requests' response-time variance.

2.1 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 [8]. The challenges are: 1) No perfect one-size-fits-all logs leading to a tussle between informativeness and cost (e.g., overhead), 2) Extremely large log search spaces, and 3) Data overload leading to a needle-in-the-haystack problem.

These challenges must be addressed separately for correctness and performance problems as logging for these two classes have different goals. Logging for correctness must identify the first divergence from normal execution that leads to problematic regions in the code [3, 31, 33, 35]. In comparison, logging for performance must identify regions of the code or resource conditions that lead requests to be slow.

No perfect one-size-fits-all instrumentation. Past research argues that the logs needed to localize the source of one problem may not be useful for others [16, 28, 33, 34]. The lack of one-size-fits-all logs leads to a tussle to identify which log statements are most helpful and should be enabled by default. For example, Zhao et al. [34] 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 enabled at every function's entry, exit, and exceptional return. (This is similar to the distributed applications used by Mace et al. [16] and Erlingsson et al. [9].) 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 add a requirement stating that frameworks must automatically narrow down the search space when exploring new problems.

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

The 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 [12]. Problem diagnosis, even when the needed instrumentation is present, is as difficult as finding a needle in a haystack [20].

This challenge is partially addressed by *R1* and *R2*. 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. 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. An 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 different critical paths and thus have different response times.

The second insight is that distributed tracing [19, 22, 24], 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 [6, 12, 14, 18, 25, 26, 30].

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 available to requests, poorly-written algorithms that unintentionally increase variance, or third-party code with unpredictable performance.

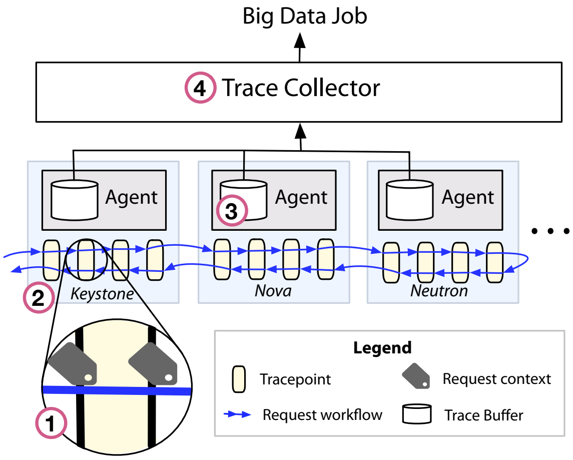


Figure 1: Distributed tracing architecture. Traces are being collected for a simplified version of OpenStack [18]. The blue line shows the workflow of a VM_LIST request.

Figure 1 shows how most distributed-tracing infrastructures work. ① Tracing infrastructures propagate per-request *context* (e.g., request IDs) 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. ④ Asynchronously, a big-data job 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.

The third insight is that the law of total variation [29] 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 equation 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

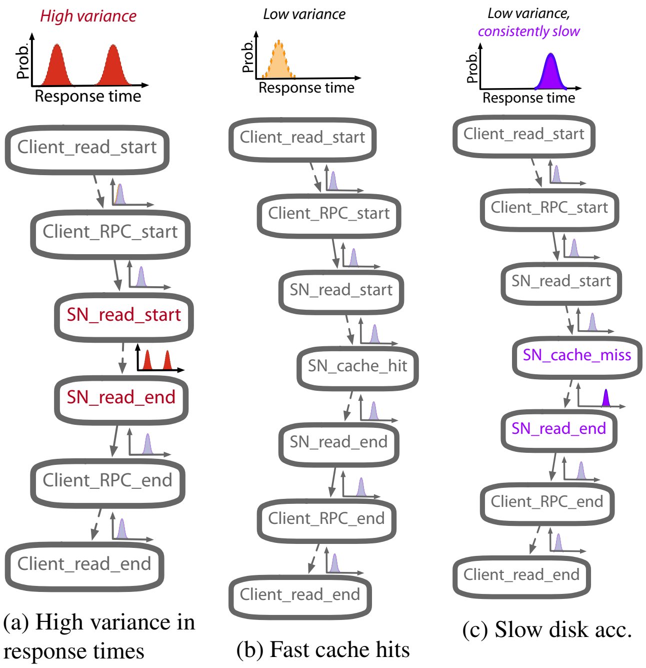
Based on the insights, the requirements can be satisfied for many classes of performance problems by combining two technologies. The first is a distributed-tracing infrastructure that allows tracepoints to be selectively enabled or disabled during runtime. The second is a control mechanism for distributed tracing that automatically enables tracepoints and considers key/value pairs exposed in them to *explain* variance as per the insights above. The control logic also includes additional principles to diagnose problems that result in identical critical paths having low variance but are very slow and to explain its decisions. 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 with 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*).



(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

As an example of differentiating fast critical paths from slow ones, Figure 2 shows three groups of identical critical-path traces from a distributed-storage system, similar to Ceph [30] or HDFS [26]. 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 response-time 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. We refer readers to Toslali et al. [27] for more examples on applying principles.

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 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.

Principle #3. Identify requests whose traces exhibit identical critical paths with 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 in that it 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 (e.g., resource contentions).

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*).

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 [9, 16], 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.

3.1 Design

Figure 3 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.

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 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

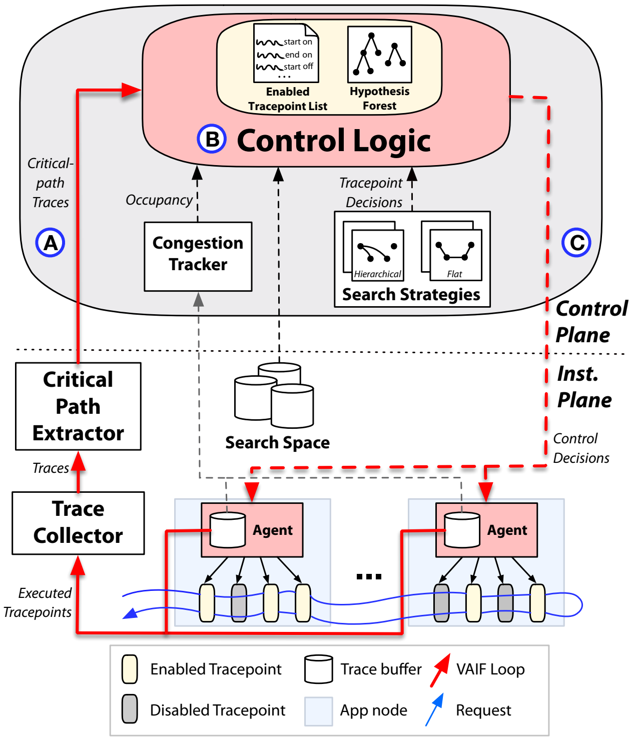


Figure 3: VAIF design. The thick red line shows VAIF's continuous loop. Solid lines are traces/tracepoints and dashed lines are control signals. A generic distributed application instrumented with tracing is shown in the instrumentation plane.

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 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 [10,15], Canopy [12], and Stardust [23]) or by learning them over a large number of traces (as done for traces that preserve only hierarchical caller/callee relationships, such as Dapper [17] and Artillery [5]).

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.

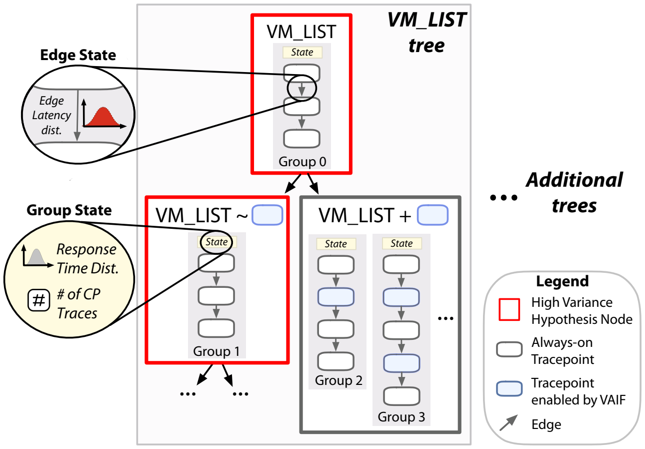


Figure 4: VAIF's hypothesis forest. This figure shows a VM_LIST tree from the forest

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.

3.2.1 Hypothesis forest

Figure 4 shows an example tree from the 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

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:    $cv \leftarrow 0.1$                                       ▷ CV threshold
8:    $cs \leftarrow 95$                          ▷ 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:     $traces \leftarrow collector$.get_new_traces()
15:     hyp.add_traces(traces)
16:     search.key_value(prev)
17:     prev.make_empty()          ▷ Only this cycle's results
18:     $cv\_gs, cv\_nodes \leftarrow hyp$.id_high_cv(cv)
19:     $cs\_gs, cs\_nodes \leftarrow hyp$.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 $i = 1 \ldots length(groups)$ do
29:     $tp \leftarrow$ search.find((groups[i], type)
30:     prev.add(hyp_nodes[i].add_child($+tp$))
31:     hyp_nodes[i].add_child($\sim tp$)
32:     tps.add(tp)
33:   end for
34:   return tps
35: end procedure
```

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. See Toslali et al. [27] for details about supported search strategies (*search.find*()) and how the search space is constructed.

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 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).

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.

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.

4 Diagnosing problems with VAIF

This section presents a case study of how we used VAIF to identify various performance problems in OpenStack. Openstack 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. VAIF enables one tracepoint per cycle per hypothesis node. For a more comprehensive set of experimental evaluation, we refer readers to Toslali et al. [27].

Unpredictable performance of VM List requests. 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 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 ([1, 2]), 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).

5 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.

References

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

[2] Openstack identity service is responding slowly. https://docs.openstack.org/operations-guide/ops-maintenance-slow.html, 2016.

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

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

[5] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. 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*, 2014.

[6] CockroachDB. https://www.cockroachlabs.com/, 2019.

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

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

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

[10] Rodrigo Fonseca, Michael J. Freedman, and George Porter. 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*, 2010.

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

[12] 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. Canopy: An end-to-end performance tracing and analysis system. In *SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

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

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

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

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

[17] Gideon Mann, Mark Sandler, Darja Krushevskaja, Sudipto Guha, and Eyal Even-dar. Modeling the parallel execution of black-box services. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Cloud Computing*, 2011.

[18] Openstack web site. https://www.openstack.org.

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

[20] A. Rabkin and R. H. Katz. How hadoop clusters break. *IEEE Software*, 30(4):88–94, 2013.

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

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

[23] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *NSDI'11: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011.

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

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

[26] The apache hadoop distributed file system, 2013. http://hadoop.apache.org/hdfs/.

[27] Mert Toslali, Emre Ates, Alex Ellis, Zhaoqi Zhang, Darby Huye, Lan Liu, Samantha Puterman, Ayse K. Coskun, and Raja R. Sambasivan. Automating instrumentation choices for performance problems in distributed applications with vaif. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 61–75, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Marc-André Vef, Vasily Tarasov, Dean Hildebrand, and André Brinkmann. Challenges and solutions for tracing storage systems: A case study with spectrum scale. *ACM Transactions on Storage (TOS)*, 14(2):18–24, May 2018.

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

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

[31] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 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*, 2010.

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

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

[34] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 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*, 2017.

[35] Zhiqiang Zuo, Lu Fang, Siau-Cheng Khoo, Guoqing Xu, and Shan Lu. 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, 2016.