Received 00 Xxxx XXXX; | Revised 00 Xxxx XXXX; | Accepted 00 Xxxx XXXX

DOI: xxx/xxxx

ARTICLE TYPE

Localizing software performance regressions in web applications by comparing execution timelines

Frolin S. Ocariza, Jr.*[1] | **Boyang Zhao**[2]

[1]Performance and Reliability Team, SAP SE, Vancouver, British Columbia, Canada
[2]Performance and Reliability Team, SAP SE, Vancouver, British Columbia, Canada

Correspondence
*Frolin S. Ocariza, Jr., Performance and Reliability Team, SAP SE, 910 Mainland Street, Vancouver, British Columbia, Canada. Email: frolin.ocariza@sap.com

Summary

A performance regression in software is defined as an increase in an application step's response time as a result of code changes. Detecting such regressions can be done using profiling tools; however, *investigating* their root cause is a mostly-manual and time-consuming task. This statement holds true especially when comparing *execution timelines*, which are dynamic function call trees augmented with response time data; these timelines are compared to find the *performance regression-causes* – the lowest-level function calls that regressed during execution. When done manually, these comparisons often require the investigator to analyze thousands of function call nodes. Further, performing these comparisons on *web applications* is challenging due to JavaScript's asynchronous and event-driven model, which introduce noise in the timelines. In response, we propose a design – ZAM – that automatically compares execution timelines collected from web applications, to identify performance regression-causes. Our approach uses a hybrid node matching algorithm that recursively attempts to find the longest common subsequence in each call tree level, then aggregates multiple comparisons' results to eliminate noise. Our evaluation of ZAM on 10 web applications indicates that it can identify performance regression-causes with a path recall of 100% and a path precision of 96%, while performing comparisons in under a minute on average. We also demonstrate the real-world applicability of ZAM, which has been used to successfully complete performance investigations by the performance and reliability team in SAP.

KEYWORDS:
Software performance, bug localization, JavaScript, web applications

1 | INTRODUCTION

Software developers often encounter situations in which the application being developed has slowed down after changes are made to the code, configuration, or environment in which the application is accessed or executed. Such slowdowns are termed *performance regressions*. Identifying and understanding performance regressions is important, as application slowdowns have a significant negative effect on the user experience, and can therefore deter many users from continuing to use an application. As an example, in their empirical study of browser bugs, Zaman et al. found that a larger percentage of users threaten to switch browsers due to performance issues, compared to non-performance issues [65]. Similarly, unresponsive web applications may cause users to abandon their usage of the web application altogether [46, 54].

Prior work – both in research and industry – has focused primarily on automating the process of detecting performance issues, including performance regressions [1, 35, 10]. This includes work done by Nistor et al. on detecting performance problems based on memory-access patterns [37], as well as a proposed detection approach by Shang et al. based on a regression model [55]. There also exists many profilers that can aid in this detection, including gprof [16], JSMeter [48], and Dynatrace [11]; modern web browsers such as Chrome and Firefox also include developer tools that support performance profiling. However, while detecting performance regressions is important, the ultimate goal is to fix these regressions. Therefore, we also need to *investigate* the causes of these performance regressions; in this paper, we refer to these causes by the term *performance regression-causes*.

Unfortunately, the process of investigating performance regression-causes is very time consuming when done manually. To give a sense of the effort involved in these investigations, a typical investigation scenario for web applications carried out by performance teams – including those in SAP and Google [5] – involves the following steps:

1. Generate execution timelines for two versions of the application: one version before the performance regression is observed, and another version after. Here, an *execution timeline* is a call tree describing the functions that executed while the user is interacting with the application, where the call tree is augmented with information on how long each function took to execute;

2. Analyze and compare the performance results included in the execution timelines. For example, the investigator would usually have to compare the functions in the call trees one by one, recursively going down the call trees until the lowest-level function that regressed is discovered. Knowing this lowest-level function is an important step in this process, as it either provides a strong lead to the code change that caused the regression or, in many cases, contains the code change itself.

The above process is further complicated by variance in the performance numbers, which often necessitates multiple execution timelines to be generated per version, and hence, for multiple timelines to be analyzed and compared.

It is particularly difficult to conduct the above investigations when performance regressions are observed at the *client-side of web applications*, due to challenges associated with JavaScript – the de facto programming language at the client. For example, modern web applications rely on features that exploit the asynchronous nature of JavaScript (e.g., timers, promises, callbacks, etc.), which can lead to large execution timelines involving multiple call trees – one for each asynchronous execution; therefore, these timelines will be very difficult for the investigator[1] to compare manually, given their size. In addition, JavaScript is an event-driven language, which means simple user interactions (e.g., mouseouts, hovers, scrolls, etc.) may trigger event handlers that are irrelevant to the performance regression, thereby introducing noise to the timelines being compared. While difficult, *conducting these client-side investigations is important, because the client-side is what the user directly interacts with*, and hence, performance regressions will be noticeable.

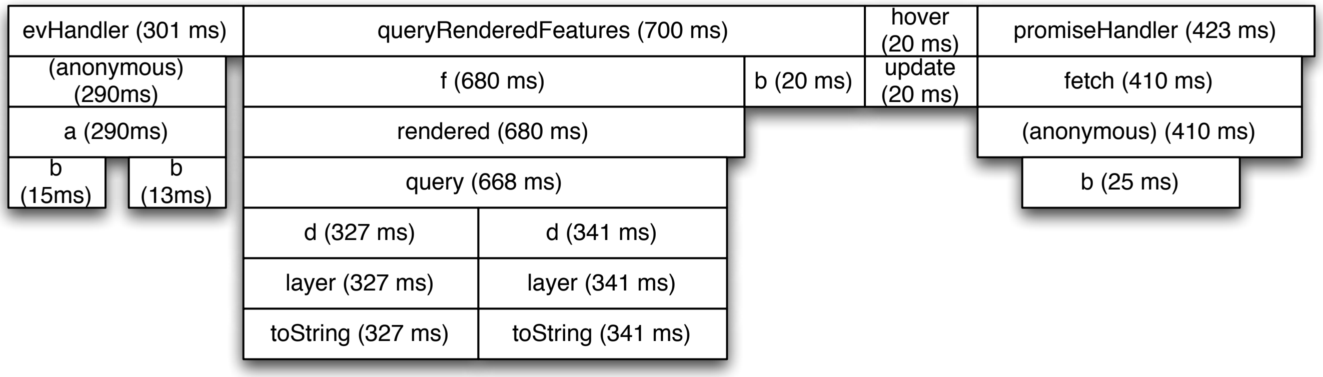
Various approaches have been suggested that attempt to automate the process of finding performance regression-causes [8, 44], including those that mine source code history to infer source code patterns that may correspond to performance regression-causes [52, 34, 23]. However – in addition to their lack of explicit support for web applications, which have their own set of unique challenges as discussed above – a huge drawback to these approaches is that they assume the existence of a repository of past performance regression-causes that have been found, which are not widely available [61]. In addition, their analyses do not take execution times into account, even though the execution time is the very metric used to judge whether a performance regression has been introduced.

To address the above issues, we have come up with an approach – implemented in a tool called ZAM – that automatically finds performance regression-causes in web applications by comparing the execution timelines of different versions of the web application (e.g., before a code change and after a code change). In this work, we decided to focus on web applications because of their ubiquity, and because many of the challenges associated with JavaScript described above are potentially extensible to other domains and would therefore be useful to address. Further, execution timelines can be generated automatically for web applications, using, for instance, Chrome DevTools; therefore, web applications are an ideal testing ground for our approach.

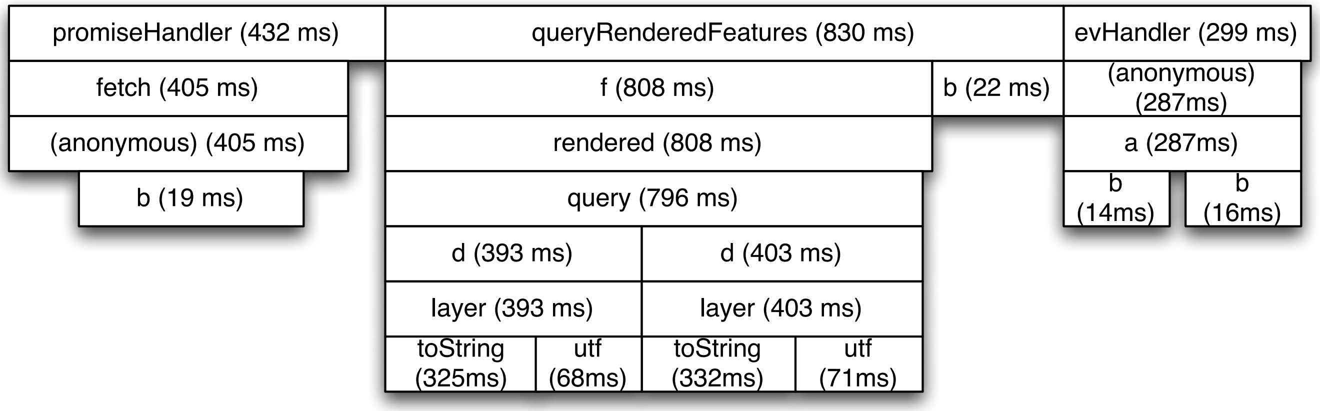
The main advantage of our approach over others is that it does not presuppose the existence of a repository of past performance regression-causes, and it works directly with an execution trace of the application under test, and therefore factors the execution times in the analysis. The latter allows for more precise results, as it aims to reveal code changes based on *actual* increases in execution time, as opposed to trying to pinpoint changes based merely on code patterns or counter signatures. In addition, to account for variance in the performance numbers, our approach is capable of aggregating the results of multiple pairs of execution timelines. Finally, unlike other comparison-based approaches such as Visual Studio [30], dotTrace [20], JProfiler [12], and Yourkit [64] (which are designed for more traditional languages), our web application-based approach is designed to handle the presence of asynchrony and large numbers of event handlers in such applications; in particular, it uses a hybrid node matching algorithm that allows it to scale to the large number of asynchronous function calls made in web applications, while still maintaining reasonable accuracy in the presence of noise (e.g., coming from various user events). *To the best of our knowledge, ours is the first approach that is capable of automatically identifying performance regression-causes in web applications, based on a direct comparison of multiple pairs of execution timelines.*

In this paper, we make the following main contributions:

[1]In this paper, we refer to the developer or tester tasked with identifying the performance regression-cause (and is therefore the user of our proposed design) as the *investigator*. We use this term to distinguish the users of our design from the users of web applications.



(a) Timeline taken before the regression is introduced (i.e., "old timeline")



(b) Timeline taken after the regression is introduced (i.e., "new timeline")

FIGURE 1 Timelines taken before and after a performance regression

- A fully automated approach for identifying performance regression-causes for web applications by comparing previously generated execution timelines, using a hybrid node matching algorithm; the output of this approach is a tree that describes the function call path to each performance regression-cause. As discussed in Section 2, performance regression-causes, in this context, are defined as the lowest-level functions in the execution timeline that regressed in performance;
- An implementation of the above approach in a tool called ZAM. ZAM has been implemented as a Node.js application;
- An empirical study that evaluates the accuracy, usefulness, real-world applicability, and performance of ZAM. Based on our results, ZAM performs its comparisons in under a minute on average and has a path recall of 100% in our experiments, as well as a path precision of about 96%. Further, even in scenarios where ZAM is unable to find the exact performance regression-cause, ZAM is still capable of finding part of the path to the performance regression-cause (called a *partial path*), which is still very useful as it leads the investigator much closer to pinpointing this cause. Finally, we also show that ZAM is capable of identifying performance regression-causes of real-world performance bugs, and it has been applied successfully to performance investigations conducted by our performance and reliability team in SAP.

2 | BACKGROUND AND MOTIVATION

In this section, we introduce a motivating example that we will be using throughout the paper. We also define some terms that are needed to understand our approach, and we outline the challenges in designing our approach.

2.1 | Motivating Example

We use the term "execution timeline" in this subsection informally to refer to profiling data, which includes information on the functions that executed at runtime and the time it took each function to execute (i.e., the response time); however, the term will be defined more formally in Section 2.2.[2]

Figure 1 shows two examples of execution timelines that are generated when interacting with a JavaScript-based web application. The execution timeline in Figure 1b was generated *after* a performance regression was introduced to the web application, while the execution timeline in Figure 1a was generated *before* this regression was introduced. This running example is based on a performance regression observed in a real-world web application,[3] with function names and response times altered for simplicity and illustration. In addition, not all rectangles are drawn to scale to make sure the function names are still visible.

As seen in Figure 1a, the execution timeline generated before the regression (i.e., the "old timeline") reveals that four separate asynchronous function calls executed at runtime: `evHandler`, `queryRenderedFeatures`, `hover`, and `promiseHandler`. The rectangles shown below each of these asynchronous function calls represent the call tree. For example, `queryRenderedFeatures` called both `f` and `b`; in turn, `f` called `rendered`, which in turn called `query`, and so on. The numbers in parentheses represent the response times of each function; in general, the response time of a function `foo` includes both the self-time (i.e., the amount of time taken executing code *within* `foo` itself) and the total time it took to execute the functions that were called by `foo`.

In contrast, for the execution timeline generated after the regression (i.e., the "new timeline"), only three of the above four asynchronous function calls were made (`hover` did not execute). In addition, the order in which these asynchronous function calls executed is also different in the new timeline compared to the old timeline. Finally, new calls to the function `utf` are made by the function `layer`, found underneath the call to `queryRenderedFeatures`.

We can compare the old timeline with the new timeline to help us localize the root cause of the performance regression. In this case, based on a manual analysis of the timelines, we can see that the largest regression among all the asynchronous function calls can be observed at `queryRenderedFeatures`, which slowed down by 130ms. Drilling down further on this function, we can see that each of the function calls to `d` and `layer` slowed down by about 70ms. As the timelines show, the reason these functions slowed down is that `layer` makes a new call to `utf` that takes 70ms to execute. Therefore, these new calls to `utf` are the ones that appear to introduce the performance regression. We can then search for the commit that added these new calls to `utf` in the source code, which would help the developer who introduced the change to further localize the regression at the source code level, and eventually, come up with a fix.

The root cause is deliberately made obvious in this running example to make the example easier to follow; nonetheless, even in these simplified execution timelines, we can already observe some of the difficulties in analyzing and comparing them manually. For example, the fact that some asynchronous function calls execute in different orders makes it difficult to visually match which calls in the old timeline correspond to the calls in the new timeline. Further, even if we peek ahead and notice that new calls to `utf` are being made at the bottom of the timeline, we cannot immediately assume that these calls introduced the performance regression. This is because higher-level functions may have also seen performance improvements that would offset the additional time taken by the new calls, which would indicate that the overall performance regression is introduced elsewhere. Therefore, determining the lowest-level function that introduced the regression presupposes finding the *path* to this function; in other words, when drilling down on `queryRenderedFeatures`, for example, it is necessary to analyze the functions one level at a time, which is tedious as we would have to constantly switch back and forth between looking at the old timeline and looking at the new timeline.

Finally, while these timelines already highlight some of the difficulties with manual analysis, full timelines generated from real-world web applications are much bigger and much more complicated, with hundreds of asynchronous calls and thousands of function calls in total, as we demonstrate later in our evaluation (Section 5.3). In turn, since investigators often look at regressions at the granularity of milliseconds, pinpointing the root cause will be very difficult to do by simply looking visually at the timelines. *These difficulties point to the usefulness of devising an automated approach for comparing execution timelines for web applications.*

2.2 | Definitions

We will now introduce terminology that will help us explain our proposed approach. First, we define an *execution trace* as a series of call trees, each representing an asynchronous function call that executed during a series of user interactions with the application. This series of user interactions can include any method of interaction with the application's interface (e.g., clicks, hovers, scrolls, etc.). We also define a *top-level function* as the starting point of a call tree in the execution trace; since the execution trace can contain multiple call trees, there can also be multiple top-level functions in the execution trace. For example, in the running example, the old timeline contains four top-level functions: `evHandler`, `queryRenderedFeatures`,

[2] For simplicity, we use the terms *execution timeline* and *timeline* interchangeably.
[3] https://github.com/mapbox/mapbox-gl-js/issues/5648

`hover`, and `promiseHandler`; in contrast, the new timeline contains only three top-level functions: `promiseHandler`, `queryRenderedFeatures`, and `evHandler`.

Based on the above definitions, we can now more formally define an *execution timeline* as follows:

Definition 1 (Execution Timeline). An *execution timeline* is a tree $T(V, E)$ whose descendants are the call trees in the execution trace. Therefore, T is rooted at a node $r \in V$ whose children represent the top-level functions of each call tree. In addition, each node $v \in V$ in the call trees is labeled with the following:

- $v.\mathrm{name}$: The function name;
- $v.\mathrm{component}$: The component name (i.e., file name);
- $v.\mathrm{time}$: The total response time;
- $v.\mathrm{children}$: An array of v's child nodes, ordered by time of execution (i.e., timestamp)

Note that for the root $r \in V$, $r.\mathrm{name}$ and $r.\mathrm{component}$ are both left blank, $r.\mathrm{children}$ contains the top-level functions of the call trees (as stated in the definition), and $r.\mathrm{time}$ is defined to be the sum of the response times of all the top-level functions.

As with the running example, when comparing execution timelines, we generally refer to the baseline (i.e., before the regression) as the *old timeline*, and we refer to the regressing timeline as the *new timeline*. Further, if a node v_{old} in the old timeline and a node v_{new} in the new timeline correspond to the same function call, v_{old} and v_{new} are said to be *matching nodes*.

We can now therefore define a *performance regression-cause* as follows:

Definition 2 (Performance Regression-Cause). Given a real number $\mu > 0$, an old timeline $T_{\mathrm{old}}(V_{\mathrm{old}}, E_{\mathrm{old}})$, and a new timeline $T_{\mathrm{new}}(V_{\mathrm{new}}, E_{\mathrm{new}})$, a *performance regression-cause* is a node $v_{\mathrm{new}} \in V_{\mathrm{new}}$ which satisfies all of the following conditions:

1. Either v_{new} has a matching node $v_{\mathrm{old}} \in V_{\mathrm{old}}$, and $v_{\mathrm{new}}.\mathrm{time} - v_{\mathrm{old}}.\mathrm{time} \geq \mu$, **or** v_{new} does not have a matching node in the old timeline and $v_{\mathrm{new}}.\mathrm{time} \geq \mu$
2. The ancestor nodes of v_{new} have matching ancestor nodes in the old timeline, and for each matching ancestor nodes a_{new} and a_{old}, $a_{\mathrm{new}}.\mathrm{time} - a_{\mathrm{old}}.\mathrm{time} \geq \mu$
3. If v_{new} has a matching node $v_{\mathrm{old}} \in V_{\mathrm{old}}$, then for each matching child node c_{new} and c_{old} of v_{new} and v_{old} (respectively), $c_{\mathrm{new}}.\mathrm{time} - c_{\mathrm{old}}.\mathrm{time} < \mu$. Further, for any child node c_{new} of v_{new} with no matching node in the old timeline, $c_{\mathrm{new}}.\mathrm{time} < \mu$

In other words, the performance regression-causes represent the *lowest-level function(s)* that regressed by at least μ based on a comparison of the old timeline and the new timeline. Note that we define performance regression-causes in this manner because the goal of our execution timeline comparisons is precisely to pinpoint these lowest-level functions, which either contain the code changes introducing the performance regression themselves, or provide a strong lead that can help developers localize those changes at the source code level.

Lastly, we define the *critical graph* as follows:

Definition 3 (Critical Graph). Given a real number $\mu > 0$, an old timeline $T_{\mathrm{old}}(V_{\mathrm{old}}, E_{\mathrm{old}})$, and a new timeline $T_{\mathrm{new}}(V_{\mathrm{new}}, E_{\mathrm{new}})$, a subtree $T_{\mathrm{sub}}(V_{\mathrm{sub}}, E_{\mathrm{sub}})$ of T_{new} is a *critical graph* if and only if the set of all the leaves in T_{sub} is the same as the set of all performance regression-causes in T_{new}.

This means that the critical graph represents the new timeline, but with nodes irrelevant to performance regressions filtered out. The critical graph therefore provides a summary of all the performance regression-causes when comparing the old and new timelines, as well as the call stacks of each of these performance regression-causes.

With the above definitions in place, we can state the problem in more formal terms as follows: *Given an old timeline* T_{old} *and a new timeline* T_{new}*, is there an approach that can generate, with high recall and precision, the critical graph that corresponds to a comparison of* T_{old} *and* T_{new}*?* We propose such an approach in Section 3.

2.3 | Challenges

One of the challenges in automating the comparison of execution timelines of web applications is that additional JavaScript function calls may be added and the order of function calls may change from one version to another. These changes occur both as a natural byproduct of the changes made to the source code, and, in the case of web applications, as a result of the asynchronous and event-driven nature of JavaScript. They may also occur due to timing differences introduced by profiling overhead, when recording execution timelines. As a result, inferring the matching nodes can be difficult, as a simple sequential mapping of the nodes based on the order of the function calls at each level of the call tree would not suffice.

Another complication is introduced by the fact that execution timelines are often very noisy, i.e., they tend to include many function calls irrelevant to the performance regression. This issue is especially present in web applications, due to extraneous event handlers that may be triggered by the user – perhaps inadvertently – while interacting with the application (as is the case with `hover` in Figure 1a). More generally, the versions of the application being compared often encompass multiple commits, since performance regression tests are often done not on a per-commit basis but after every several commits; the former is simply infeasible given the amount of time it takes to run performance tests (usually hours) and the frequency of commits (multiple commits per hour). The large number of commits makes the execution timelines more susceptible to noise, given the number of changes that are likely unrelated to the performance regression.

An additional factor that contributes to this noise is the fact that in web applications, which is the primary domain in which JavaScript is used, there is an inherent interplay between the JavaScript code execution, as well as other components such as renders, paints, and network requests. This interplay affects the way in which the JavaScript call tree is displayed in the timeline; however, these additional components are difficult to "filter out" when simply looking at the timelines manually.

Finally, performance varies from execution to execution, which adds further noise to the timelines. Thus, a comparison of an old and a new timeline may show extraneous regressions that do not appear in other timeline pairs. As a result, multiple pairs of timelines often need to be generated and compared, which suggests the need for a method to aggregate the results from each comparison.

2.4 | Scope

Before we describe our algorithm, we would first of all like to clarify the scope of our design and of this paper. First, as mentioned previously, our proposed approach is specifically designed for the client-side of web applications. As mentioned in Sections 1 and 2.3, our choice to focus on web applications is driven by unique challenges associated with this domain, as well as the ubiquity and widespread usage of such applications. Therefore, for the purposes of this research, we do not claim full generality to all application types; nonetheless, in Section 6, we briefly touch upon certain aspects of our design that may generalize to other application types.

Furthermore, our approach is designed to assist with localizing performance regressions that manifest as increases in response times in the web application. We would like to note that the term "performance regression" is often used to more generally refer to regressions not only in response times, but also other non-functional aspects of the execution such as memory and CPU usage. Nonetheless, response time increases are commonly encountered in web applications and are impactful [46, 54]; therefore, finding ways to facilitate the process of localizing them is important and is a worthwhile research pursuit.

Finally, we understand that there are other important aspects of conducting software performance investigations, including the reproduction of the environment. Nevertheless, this is not the focus of this paper, but rather, the focus is on finding performance regression-causes given that the regression can already be reproduced. As explained in Section 1, this is challenging to do manually.

3 | APPROACH

Figure 2 shows an overview of our automated approach. As seen in this figure, our approach takes three inputs: (1) the set of old timelines S_{old}; (2) the set of new timelines S_{new}; and (3) a real number μ, which indicates the minimum difference in response time between matching nodes that is considered a significant regression by our approach (this symbol was introduced in Section 2.2). Note that the number of timelines in S_{old} must be equal to the number of timelines in S_{new}; in other words, $|S_{old}| = |S_{new}|$.

Our approach contains two main modules. The first module is *timeline pair comparison* (discussed in Section 3.1), which compares pairs of execution timelines one by one, with each pair consisting of an old timeline and a new timeline. This module contains two main algorithms: (1) `findCriticalGraph()`, which traverses the timeline pairs to filter out nodes that are irrelevant to the performance regression, as well as some "non-useful" nodes (i.e., nodes that may mislead the approach and are therefore susceptible to generating false positives, including anonymous functions and some functions with minified names); and (2) `findMatchingNodes()`, a hybrid algorithm that tries to infer matching nodes among a list of nodes in the same tree level in the execution timeline pairs. The output of this first module is a set of critical graphs C, where one critical graph is generated for each timeline pair compared.

The second module is *critical graph aggregation* (discussed in Section 3.2), which takes the set of critical graphs C generated by the previous module, and aggregates the graphs. The final output is an aggregated critical graph that takes into account the results of all the timeline pair comparisons.

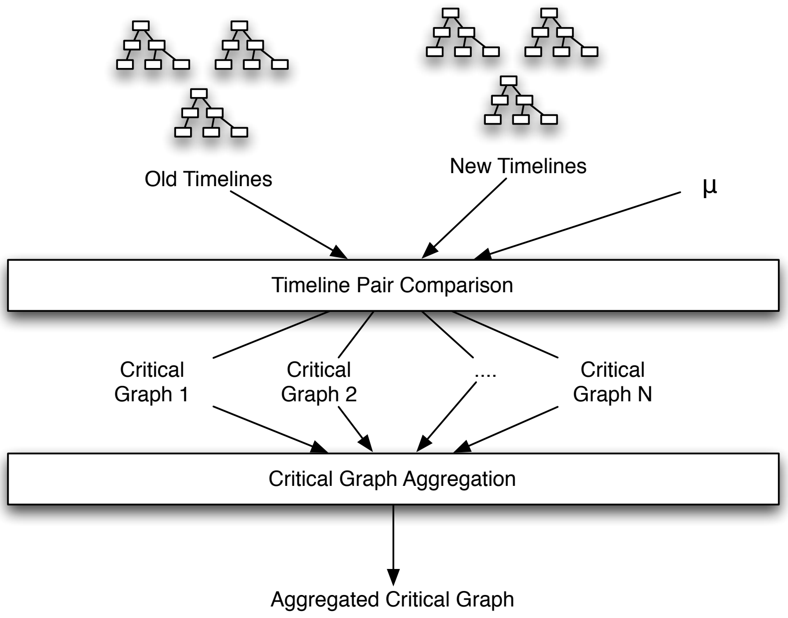


FIGURE 2 Overview of our approach

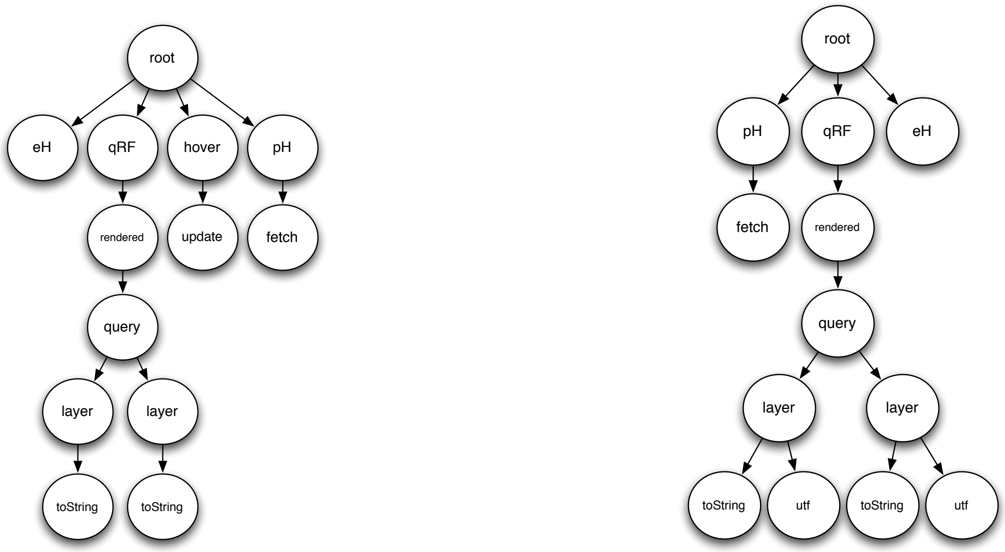
Algorithm 1: compareTimelinePairs

```
Input: $S_{old}$: The set of old timelines
Input: $S_{new}$: The set of new timelines, with $|S_{old}| = |S_{new}|$
Input: $\mu$: The minimum response time difference to track
Output: $C$: Set of critical graphs
1 $C \leftarrow \emptyset$;
2 $\mathbf{A_{old}} \leftarrow$ toArray($S_{old}$);
3 $\mathbf{A_{new}} \leftarrow$ toArray($S_{new}$);
4 for $i \leftarrow 1$ to $|S_{old}|$ do
5     $r_{old} \leftarrow$ pruneTree($\mathbf{A_{old,i}}$);
6     $r_{new} \leftarrow$ pruneTree($\mathbf{A_{new,i}}$);
7     $G \leftarrow$ findCriticalGraph(($r_{old}$, $r_{new}$), $\mu$, true);
8     $C \leftarrow C \cup \{G\}$;
9 end
```

3.1 | Timeline Pair Comparison

Our approach begins at the *timeline pair comparison* module, which takes the sets of execution timelines provided as input by the investigator, compares the timelines pairwise, and generates a set of critical graphs (defined in Section 2.2) based on each comparison. Algorithm 1 shows the high-level algorithm for this module, which starts by initializing the set of critical graphs that will eventually be output (line 1) and converting the two sets of execution timelines S_{old} and S_{new} into arrays $\mathbf{A_{old}}$ and $\mathbf{A_{new}}$, respectively (lines 2-3). The latter is accomplished by invoking the `toArray()` function, and its purpose is to assign the order in which the timelines are compared, and the pairings. In particular, the old timeline from index i in $\mathbf{A_{old}}$ (denoted by $\mathbf{A_{old,i}}$) is paired with the new timeline from index i in $\mathbf{A_{new}}$ (denoted by $\mathbf{A_{new,i}}$), and these timelines are compared; thus, the total number of comparisons is equal to the number of old timelines provided as input (which itself is equal to the number of new timelines provided as input). Currently, the timelines are ordered alphabetically based on the name of the file containing the timeline data; this gives the investigator some freedom to decide the pairing. It is worth noting, however, that (1) the effect of different pairings is out of the scope of our design and of this paper, and (2) once the pairings have been decided, the order in which the timeline pairs are compared does not actually change the result of our design.

In lines 4–9, our approach iterates through the indices of the $\mathbf{A_{old}}$ and $\mathbf{A_{new}}$ arrays. In each iteration, our approach first takes the execution timelines in the ith position of each array (i.e., $\mathbf{A_{old,i}}$ and $\mathbf{A_{new,i}}$) and passes them as input to the `pruneTree()` function (line 5). The purpose of the `pruneTree()` function is to remove timeline nodes whose name is blank, "(anonymous)", or consists of only a single character. Nodes representing anonymous functions are removed from the timeline as they often provide little valuable information to the investigator and, more importantly, they make the approach more susceptible to false positives since the approach might mistake two different anonymous functions to be the same



(a) Pruned old timeline **(b)** Pruned new timeline

FIGURE 3 Pruned timelines, with some function names abbreviated and response times omitted for simplicity (eH refers to `evHandler`, qRF refers to `queryRenderedFeatures`, and pH refers to `promiseHandler`)

given that they have the same value for the name.[4] Nodes with single-character names are also removed since they often represent the minified names assigned to some functions, which often change from commit to commit. Whenever a node v is removed from an execution timeline, the parent of v becomes the parent of the children of v. Figure 3 shows the pruned execution timelines corresponding to the old and new timelines from the running example (with some function names abbreviated).

Note that the calls to `pruneTree()` return the roots of the pruned execution timelines. These roots – labeled as r_{old} and r_{new} – are passed to the `findCriticalGraph()` method (line 7), which is responsible for comparing the two (pruned) timelines to which these roots correspond, and calculating the critical graph for this comparison. Each critical graph generated from each comparison is placed in the set of critical graphs C (line 8), which is the output of this module.

The pseudocode for `findCriticalGraph()` is shown in Algorithm 2. This recursive function takes three inputs: (1) a pair of nodes (r_{old}, r_{new}); (2) the minimum response time difference μ; and (3) *useAlternate*, which is used by the `findMatchingNodes()` algorithm that we will describe later. The key idea to note here is that the `findCriticalGraph()` function assumes that the pair of nodes (r_{old}, r_{new}) represents matching nodes; this way, based on the way the algorithm is designed, the burden to decide which nodes are matching is exclusively placed on the `findMatchingNodes()` algorithm.

The `findCriticalGraph()` algorithm starts by creating a new graph that is initialized with only one node – a root node r_G that is a copy of the node r_{new} (lines 1–3). The algorithm then invokes the `findMatchingNodes()` algorithm, which outputs an array $\mathbb{M}$ of node pairs, with each pair representing matching nodes between the children of r_{old} and the children of r_{new} (line 4), where the response time difference is at least μ. Note that for some node pairs (v_{old}, v_{new}) in $\mathbb{M}$, the value of v_{old} may be `null`; the conditions under which such pairs are included in the array are explained later in this section when we describe `findMatchingNodes()`.

The rest of the algorithm (lines 5–18) consists of a for loop that iterates through each pair of matching nodes (v_{old}, v_{new}) in $\mathbb{M}$. For each of these pairs, two possible scenarios are encountered, depending on the value of v_{old}. In particular, if v_{old} is `null`, this would indicate that the node v_{new} represents a *new* function call with a response time of at least μ that is present in the new timeline, but was not present in the old timeline, and is therefore a performance regression-cause. As a result, v_{new} is added as a child of r_G (lines 8–10), and no further exploration is done.

On the other hand, if v_{old} is *not* `null`, then v_{old} and v_{new} represent matching nodes where v_{new}'s response time is slower than v_{old}'s response time by at least μ. In this scenario, the `findCriticalGraph()` function is recursively called with (v_{old}, v_{new}) as input (line 12) to see if there are any matching descendant nodes whose response times differ by at least μ; the root node resulting from this recursive call is added as a child of r_G, and the recursion halts once a performance regression-cause is encountered (i.e., there are no longer any matching descendant nodes whose response times differ by at least μ). Therefore, by the end of the algorithm, the graph G would only consist of the nodes that lead up to the performance

[4]We considered using the line and column numbers as potential identifiers for anonymous functions, but the problem is that these numbers frequently change from commit to commit, and will therefore not be of much use in distinguishing functions.

Algorithm 2: findCriticalGraph

```
Input: ($r_{old}$, $r_{new}$): A pair of nodes, where $r_{old}$ is from the old timeline and $r_{new}$ is from the new timeline. $r_{old}$ and $r_{new}$ are assumed to
       be matching nodes.
Input: $\mu$: The minimum response time difference to track
Input: useAlternate: If true, an alternate matching algorithm will be used instead of "longest common subsequence"
Output: G: The critical graph when comparing the tree rooted at $r_{old}$ and the tree rooted at $r_{new}$
1  $r_G \leftarrow$ copy of $r_{new}$;
2  $r_G$.children $\leftarrow$ [];
3  G($V_G$, $E_G$) $\leftarrow$ ({$r_G$}, $\emptyset$);
4  $\mathbf{M} \leftarrow$ findMatchingNodes(($r_{old}$.children, $r_{new}$.children), $\mu$, useAlternate);
5  for i $\leftarrow 1$ to $|\mathbf{M}|$ do
6      ($v_{old}$, $v_{new}$) $\leftarrow \mathbf{M}_i$;
7      if $v_{old}$ = null then
8          $r_G$.children.add($v_{new}$);
9          $V_G \leftarrow V_G \cup \{v_{new}\}$;
10         $E_G \leftarrow E_G \cup \{(r_G, v_{new})\}$;
11     else
12         H $\leftarrow$ findCriticalGraph(($v_{old}$, $v_{new}$), $\mu$, false);
13         $r_H \leftarrow$ root of H;
14         $r_G$.children.add($r_H$);
15         $V_G \leftarrow V_G \cup \{r_H\}$;
16         $E_G \leftarrow E_G \cup \{(r_G, r_H)\}$;
17     end
18 end
```

regression-causes, and hence, by definition, is a critical graph. Figure 4 shows the critical graph generated after comparing the timelines from the running example, with μ set to 50ms (we discuss considerations taken when deciding a value for μ in Section 6).

Lastly, Algorithm 3 shows the pseudocode for `findMatchingNodes()`. This algorithm takes the following as input: (1) a pair of arrays ($\mathbf{B_{old}}$, $\mathbf{B_{new}}$), where the arrays contain the nodes that need to be matched; (2) the minimum response time difference μ; and (3) the boolean value *useAlternate*, which forces the algorithm to use the "alternate matching algorithm" described shortly. The output is an array $\mathbf{M}$ of matching node pairs (v_{old}, v_{new}), where v_{old} may be `null`, as discussed above.

The algorithm begins by initializing the array $\mathbf{M}$ (line 1) and invoking the main matching algorithm used by our approach, which finds the "longest common subsequence" when comparing two arrays (line 2). The longest common subsequence (LCS) between two arrays is defined as the longest sequence of values found in both arrays, where the values in the sequence are not necessarily positioned consecutively in either array. In this case, the `longestCommonSubsequence()` call in line 2 returns a pair of arrays $(\mathbf{J}, \mathbf{K})$, where $\mathbf{J}$ and $\mathbf{K}$ contain the indices of the matching nodes in $\mathbf{B_{old}}$ and $\mathbf{B_{new}}$, respectively. For example, if the first element of $\mathbf{J}$ is 5 and the first element of $\mathbf{K}$ is 7, then the 5th element of $\mathbf{B_{old}}$ and the 7th element of $\mathbf{B_{new}}$ have been identified as matching nodes; the same principle applies to the second element of $\mathbf{J}$ and $\mathbf{K}$, the third element, the fourth element, and so on.

We decided to infer the matching nodes based on LCS for various reasons. First, the LCS problem is a well-studied computer science problem, and in fact, our approach uses an algorithm similar to the ones suggested in Rosetta Code [50]. Second, finding the LCS allows us to preserve the order in which the functions are called – even in cases where the same function is called in multiple, non-consecutive instances – and it considers the possibility of function calls being added or removed across versions.

There is a caveat to finding the LCS, however: the LCS problem applied to arrays with arbitrary lengths and values is NP-complete [24], and the speed of the algorithm depends on the values in the arrays and the array's sizes. In the case of execution timelines for web applications, function calls in the same level of the timeline tend to follow more or less the same sequence from version to version, with only slight alterations that are mostly due to function calls that have been added or removed; in such cases, the LCS algorithm tends to execute very quickly. Nonetheless, there may still be cases where some functions are refactored to the point that the function call sequence becomes drastically different, and the LCS algorithm often does not scale when encountering such situations. In addition, the order of execution for top-level functions often changes in web applications – as is the case in our running example where the `evHandler` and `promiseHandler` functions executed in different orders in different executions – which, again, will make the LCS algorithm execute slowly.

To account for the above scalability issue, we have devised a simpler alternate matching algorithm (lines 3–22) which, though potentially less accurate than LCS, is much more scalable. This alternate matching algorithm only considers the order of functions with the same component and name, and not the overall order of the functions. Further, it only executes under two conditions. First, it always executes when matching top-level functions, for the reasons just discussed; this is the reason why the `compareTimelinePairs()` function in Algorithm 1 sets the value of *useAlternate* to `true` when calling `findCriticalGraph()` (whereas the recursive call to `findCriticalGraph()` in Algorithm 2 sets this same variable to `false`).

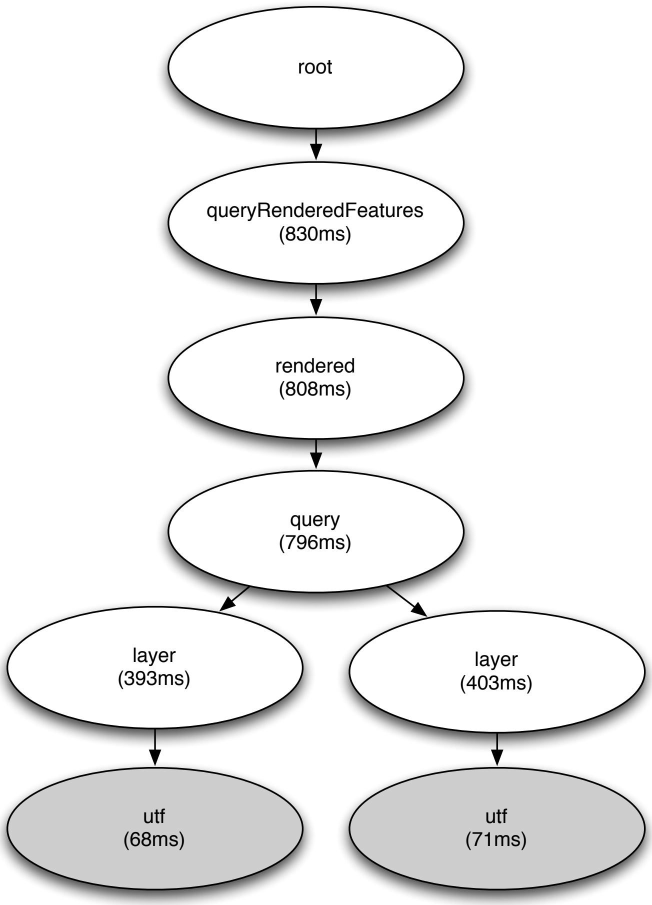


FIGURE 4 Critical graph generated by our approach for the running example. Grey-filled nodes are performance regression-causes.

Second, the alternate matching algorithm also executes when the execution time of the LCS algorithm exceeds a threshold, at which point $\mathbf{J}$ and $\mathbf{K}$ are both set to `null`; the value we have set for this threshold is discussed in Section 4.

The alternate matching algorithm sets the values of the arrays $\mathbf{J}$ and $\mathbf{K}$ in a way similar to the LCS algorithm. After initializing the values of these arrays to empty arrays (line 4), the alternate algorithm iterates through the nodes in $\mathbf{B_{old}}$ and places the index that the node appears in $\mathbf{B_{old}}$ in a hash map $\mathsf{hm_{old}}$ (lines 6–12). The hash map keys are defined by the component and the function name of the node, and the hash map values are arrays of integers containing the indices to $\mathbf{B_{old}}$. Once the hash map has been populated, the algorithm iterates through the nodes in $\mathbf{B_{new}}$ (lines 13–21). If a node's component and function name matches a key in the hash map, the index of that node in $\mathbf{B_{new}}$ is added to the array $\mathbf{K}$, and the matching index from the hash map is added to the array $\mathbf{J}$. This process continues until all nodes in $\mathbf{B_{new}}$ have been analyzed.

Once the arrays of indices $\mathbf{J}$ and $\mathbf{K}$ have been populated by our hybrid node matching algorithm (i.e., either by the LCS algorithm or the alternate matching algorithm), our approach iterates through the indices of $\mathbf{B_{new}}$ (lines 23–34) and adds pairs of matching nodes to the output array $\mathbf{M}$. In each iteration, the approach determines whether the current index i is in $\mathbf{K}$. If so, it retrieves the index – called *kIndex* – in which the value i appears in $\mathbf{K}$ (line 26), and it subsequently retrieves the index values that appear in the *kIndex*th position of both $\mathbf{J}$ and $\mathbf{K}$. These index values are then used to retrieve the corresponding matching nodes from $\mathbf{B_{old}}$ and $\mathbf{B_{new}}$, respectively, and these matching nodes are added as a pair to $\mathbf{M}$ if their difference in response time is at least μ (line 29). On the other hand, if the value i does not appear in $\mathbf{K}$, this would indicate that the corresponding node in $\mathbf{B_{new}}$ has no matching node in $\mathbf{B_{old}}$. The node is still added as a pair to $\mathbf{M}$ if its response time is at least μ, albeit it is paired with the `null` value (line 32).

3.2 | Critical Graph Aggregation

Once the timeline pair comparison module has generated a set of critical graphs, these graphs are fed as input into the *critical graph aggregation* module. As the name suggests, the purpose of this module is to put together the intermediate critical graphs generated by the previous module in

```
Algorithm 3: findMatchingNodes
  Input: $(\mathbf{B_{old}}, \mathbf{B_{new}})$: A pair of arrays containing the nodes that need to be matched, where $\mathbf{B_{old}}$ is an array of child nodes from
         the old timeline and $\mathbf{B_{new}}$ is an array of child nodes from the new timeline.
  Input: $\mu$: The minimum response time difference to track
  Input: useAlternate: If true, an alternate matching algorithm will be used instead of “longest common subsequence”
  Output: $\mathbf{M}$: An array of matching nodes, represented as pairs ($v_{old}$, $v_{new}$). Note that for some pairs in this array, $v_{old}$ may be null if
          $v_{new}$ has no matching node but $v_{new}$.time $\geq \mu$
1  $\mathbf{M} \leftarrow$ [];
2  $(\mathbf{J}, \mathbf{K}) \leftarrow$ longestCommonSubsequence($\mathbf{B_{old}}, \mathbf{B_{new}}$);
3  if $\mathbf{J}$ is null $\vee$ useAlternate then
       /* Either longestCommonSubsequence took too long to execute or top-level functions are being
          compared, so try alternate matching.                                                    */
4      $(\mathbf{J}, \mathbf{K}) \leftarrow$ ([], []);
5      $hm_{old} \leftarrow$ hash map of (key, value) pairs (k, v);
6      for i $\leftarrow 1$ to $|\mathbf{B_{old}}|$ do
7          next $\leftarrow \mathbf{B_{old,i}}$;
8          if (next.component, next.name) $\notin hm_{old}$.keys then
9              $hm_{old}$[(next.component, next.name)] $\leftarrow$ [];
10         end
11         $hm_{old}$[(next.component, next.name)].add(i);
12     end
13     for i $\leftarrow 1$ to $|\mathbf{B_{new}}|$ do
14         next $\leftarrow \mathbf{B_{new,i}}$;
15         if (next.component, next.name) $\in hm_{old}$.keys then
16             oldIndex $\leftarrow hm_{old}$[(next.component, next.name)]$_1$;
17             $\mathbf{J}$.add(oldIndex);
18             $\mathbf{K}$.add(i);
19             $hm_{old}$[(next.component, next.name)].remove(oldIndex);
20         end
21     end
22 end
23 for i $\leftarrow 1$ to $|\mathbf{B_{new}}|$ do
24     next $\leftarrow \mathbf{B_{new,i}}$;
25     if i is in the array $\mathbf{K}$ then
26         kIndex $\leftarrow \mathbf{K}$.indexOf(i);
27         oldIndex $\leftarrow \mathbf{J}_{kIndex}$;
28         if next.time - $\mathbf{B_{old,oldIndex}}$.time $\geq \mu$ then
29             $\mathbf{M}$.add(($\mathbf{B_{old,oldIndex}}$, next));
30         end
31     else if next.time $\geq \mu$ then
32         $\mathbf{M}$.add((null, next));
33     end
34 end
```

order to generate an “aggregated critical graph”, which is the final output of our approach. This aggregated graph includes all the nodes that appear in *all* the intermediate critical graphs, and excludes all other nodes. Hence, this is an important step that aims to decrease the number of false positives reported by our approach (where a false positive, in this context, refers to a function call that is reported as a performance regression-cause, but is actually not a performance regression-cause) and therefore attempts to eliminate the noise often seen in execution timelines, which is listed as one of the challenges in Section 2.3. Incorporating this step in our approach also means that the number of execution timelines included in the input will have an effect on the final output. In our evaluation in Section 5, we test the effect of varying the number of execution timelines.

Our algorithm for aggregating two critical graphs is similar to our alternate matching algorithm from lines 3–22 of Algorithm 3. That is, when comparing each level of the critical graph, the approach first iterates through the nodes in that level from the first critical graph and stores the indices in a hash map. Afterwards, the approach iterates through the nodes from the second critical graph and, whenever a node with the same component and function name is encountered based on the hash map, that node is added to the aggregated critical graph and further explored recursively. This comparison is done for each subsequent critical graph. In other words, the first and second critical graphs are compared, which generates an aggregated graph; this aggregated graph is then compared with the third critical graph, which generates another aggregated graph; and so on, until all critical graphs have been included in a comparison.[5] In essence, our algorithm looks for the largest common subtree among all the critical graphs.

[5]The response times displayed in the aggregated critical graph are the averages of the corresponding response times across all the intermediate critical graphs

4 | IMPLEMENTATION

The approach described in the previous section has been implemented in a tool called ZAM,[6] and this tool has helped SAP's performance and reliability team in investigating and localizing many performance regressions, as we demonstrate in Section 5; in particular, running execution timelines by ZAM has now been incorporated as one of the core steps when trying to investigate client-side performance regressions in SAP Analytics Cloud (SAC) [53]. We have implemented the tool as a Node.js application. It takes two folders as input – one containing the old timelines, and the other containing the new timelines; these timelines are saved as JSON files that have been recorded using Chrome DevTools [15]. In addition, the investigator is required to specify the value for the minimum response time difference μ in milliseconds. Finally, ZAM allows the investigator to specify other optional parameters, such as the maximum number of execution timeline pairs to compare, as well as an output file name. The output of ZAM is a file written in the DOT description language [17, 13]; the contents of this file can be viewed using DOT viewers such as Graphviz [49] or viz-js [9].

Note that the execution timelines have to be recorded manually by the investigator. Further, when running ZAM, there is no need for the investigator to have the web application itself open, so the analysis can be done offline. The process of recording an execution timeline is quite simple – the investigator simply needs to open Chrome DevTools, click on the "Performance" tab, and then start recording any action from there. Although the steps themselves are simple, the process is admittedly time-consuming. While the process of recording these timelines is completely out of the scope of our approach, we nonetheless provide a brief discussion of this process in Section 6, including some requirements needed if one were to consider automating it.

ZAM uses a Node.js module called DevTools Timeline Model [19] to extract call tree information from the input JSON files. Using this module, ZAM extracts the top-down aggregated call tree from the JSON files. Using the top-down call tree allows ZAM to identify regression-causes that result from repeated runs of the same function that accumulate into larger response times; note, however, that using the top-down call tree does not significantly simplify the matching of nodes at each level, given the sheer number of unique function calls to be matched in the first place. In Section 5, we show that the nodes and the unique function names found in typical execution timelines number in the hundreds or thousands, even when already aggregated top-down.

From experience in using the tool, we find that 10 seconds is a reasonable threshold to set when determining whether to switch to the alternate matching algorithm if the LCS algorithm is taking too long to execute (see Section 3.1 and Algorithm 3). While exploring the effects of changing this threshold is out of the scope of this paper, our evaluation results in Section 5 corroborate our current threshold choice, as our results demonstrate that ZAM does not take a significant amount of time to execute even for large call trees, while still retaining good accuracy.

5 | EVALUATION

Here, we conduct an evaluation of ZAM to assess its accuracy, usefulness, and performance. We are particularly interested in answering the following research questions:

RQ1 (Accuracy): How accurate is ZAM in identifying performance regression-causes in web applications?

RQ2 (Effects of Aggregation): What are the effects of varying the number of aggregated timeline pairs in ZAM?

RQ3 (Effects of Matching Algorithms): How often is each matching algorithm used by ZAM and what are the effects on the accuracy?

RQ4 (Usefulness): How useful is ZAM in pruning out irrelevant functions from the timelines to minimize manual inspection by the investigator?

RQ5 (Real Bugs): Is ZAM capable of finding the performance regression-cause in real-world bugs involving performance regressions?

RQ6 (Real-World Applicability): How helpful is ZAM in conducting real-world performance investigations?

RQ7 (Performance): How quickly can ZAM generate its output?

Note that while a comparison with analogous tools (e.g., YourKit, Visual Studio, etc.) could be potentially useful, carrying out such a comparison is not currently possible, not only because these tools are proprietary and closed-source, but also because they do not support web applications. Nonetheless, we believe the above research questions allow us to conduct a thorough evaluation on ZAM.

[6] ZAM is named after the character Zam Wesell from the *Star Wars* movies. She is a "changeling" who can alter her appearance, in the same way the appearance of timelines changes from execution to execution

5.1 | Experimental Subjects

We chose our subjects from among an online list of over 600 self-hosted web applications [21]. We chose ten web applications from this list in uniformly random fashion, by assigning a random ranking to each application and choosing the ten highest-ranked applications. In our selection, we filter out any applications that are not web applications (since the list also includes network services, for which ZAM is not designed). Further, since we are conducting our evaluation on a Windows environment, we ignore any applications that require non-Windows environments to be set up. We decided to limit the number of subject web applications to ten because as we describe in Section 5.2, we will be capturing a large number of timelines for each application, which takes a lot of effort; we feel that this number allows us to conduct our evaluation with reasonable effort, while still providing us statistically significant results. The subject applications are listed in Table 1.

The subject applications are deployed as locally hosted web applications on a Microsoft Surface running Windows 10, with 4 GB of RAM and a 1.60 GHz Intel Atom x7-Z8700 processor. As described in Section 5.2, we also collect execution timelines from the subject applications; the subject applications are executed and timelines are collected from this same machine.

5.2 | Evaluation Methodology

This section describes our methodology for answering each of our research questions.

Accuracy (RQ1). To answer RQ1, we conduct a fault injection experiment. Note that the goal of RQ1 is to measure the accuracy of ZAM in finding the performance regression-cause. To do so, we perform the following injections, which are based on the most common JavaScript performance issue root causes identified by Selakovic and Pradel [54]:

- **Injection 1** (*Inefficient API function*): A function that performs at least one call to an external API is randomly chosen from the subject application, and within this function, an external API call is randomly chosen. A sleep time between 100ms and 200ms is then injected in the chosen external API call. The root causes corresponding to this injection apply to over 50% of the performance issues analyzed in [54];
- **Injection 2** (*Long-running loops*): A function is randomly chosen from the subject application, and a for loop that executes multiple iterations (amounting to a slowdown of 100-200ms per call to the function) is injected at the beginning of the function. The root causes corresponding to this injection apply to at least around 20% of the performance issues in [54];
- **Injection 3** (*Repeated checks of the same condition*): A function that is called multiple times is randomly chosen from the subject application, and within this function, a boolean expression `expr` used as the condition for an if statement or a loop is randomly chosen. The expression is modified to `dummyZam() && expr`, where `dummyZam()` is a custom function that always evaluates to `true` and executes for about 10ms per call. The execution time of `dummyZam()` is deliberately made small, as the corresponding pattern mentioned in [54] refers to cases where the code slows down as a direct result of multiple evaluations of the same conditional. Note also that if no conditional exists in the chosen function, the body of the function is wrapped with an if, where the if condition is `dummyZam()`. The root cause corresponding to this injection applies to around 10% of the performance issues in [54].

Note that multiple root causes map to the first two injections above (e.g., all the API-related root causes identified in [54] are simply variations of the first injection pattern, in which a slowdown is observed in an API function). Some root causes from [54] are ignored as they are both too general and only apply to a small percentage of performance issues.

We perform two injections for each of the injection types listed above, for a total of six distinct injections per application. This means that in total, we collect 40 execution timelines for each application. More specifically, we collect 10 execution timelines for the baseline case (i.e., no functions modified); we then perform the six distinct injections to the application *one at a time*, and we collect 5 execution timelines for each injection, for an additional 30 execution timelines. The timelines are collected using Chrome DevTools (via Chrome v. 80.0.3987.132), and for each application, we perform the same series of user interactions to generate all the timelines. The user interactions carried out to generate the timelines are common use cases for each application, and are listed in Table 1.

Each of the six injections requires a function to randomly be chosen from the application, as stated in the injection specifications above. The functions chosen for each application are also listed in Table 1. The functions are chosen by extracting a list of functions recorded in the baseline timelines, and randomly choosing six functions from this list. Note that we ignore anonymous functions and functions whose name contains fewer than two characters, as these functions are pruned out by ZAM as discussed in Section 3.1.

Once we have the execution timelines, we run ZAM on these execution timelines as follows. For each app, we run ZAM with one set of baseline timelines used as the old timelines, and another set of baseline timelines used as the new timelines; this run constitutes our base case, where the result is expected to be empty. Once this is done, we then run ZAM with the first set of baseline timelines used as the old timelines, and the timelines generated from the first injection as the new timelines. We repeat the same comparison with the timelines generated from the second, third, fourth,

TABLE 1 The subject web applications used in our evaluation. Note that the "App Size" column only includes the size of actual code (HTML, CSS, and JavaScript) in the web application; other static assets such as images and documents are excluded.

Application Name	Application Type	App Size (KB)	Modified Functions	User Interactions
Read the Docs	Documentation Generator	86323	`hide_dropdown`, `open_dropdown`, `grep`, `matches`, `find`, `show`	Expand the "View Docs" dropdown, type "stable", and type away from the dropdown
Resource Space	File Manager	70246	`hideMyCollectionsCols`, `SlideshowChange`, `showInvisibly`, `map`, `dispatch`, `extend`	Logon to the homepage
GNU Social	Social Communication	61440	`beforeSend`, `CheckBoxes`, `superMatcher`, `nodeName`, `handle`, `fix`	Add the notice to favourites, remove the notice from favourites, click settings, and click on "EMAIL"
Live Helper Chat	Communication System	30310	`simulate`, `Ee`, `unary`, `instantiate`, `unary`, `select`	Logon to the homepage
Antville	Blogging Platform	87142	`set`, `promise`, `setRequestHeader`, `remove`, `compile`, `component`	Logon to the homepage, click "Create a site", and click "Cancel"
FlaskBB	Forum	111616	`show_toolbar`, `lt`, `removeClass`, `cookie`, `each`, `compile`	Logon to the homepage and click on the right button to show the toolbar
Searx	Search Engine	186368	`searchInputFocus`, `removeFocus`, `promise`, `nameToUrl`, `_data`, `on`	Search "star wars", and press the following hotkeys: i, Esc, f, b, j, k, n, r, h, h
h5ai	File Manager	289	`getLostPoint`, `rmAttr`, `getErrorCorrectPolynomial`, `exports`, `each`, `appTo`	Click on the checkmark beside the "private" folder twice, click on the "private" folder, and click on the "_h5ai" folder
JSBin	Debugging Tool	181248	`render`, `setCode`, `run`, `getRenderedCode`, `cancel`, `handleMessage`	Show the JS panel, paste a function definition in this panel, modify the sample code in the HTML pane, press "Run with JS", and hide the JS panel
Selfoss	Feed Reader	10209	`getResponseHeader`, `domManip`, `param`, `isPlainObject`, `select`, `merge`	Click on the first item in the feed, mark the item as read, mark the item as unread, star the item, unstar the item, and deselect the item

fifth, and sixth injections used as the new timelines, for a total of seventy comparisons (i.e., seven comparisons per subject application, including the base case run). For this particular research question, we will only be aggregating three pairs of execution timelines per comparison; however, in RQ2, we will be exploring the effects of altering the number of aggregated pairs. In addition, we have chosen to set the value of the minimum response time difference μ to 50 ms. We decided to assign a small value for μ because in realistic scenarios, the investigator running ZAM will not know how the observed performance regression is "distributed" across the different functions in the timeline; in other words, they would not know how much individual functions in the timeline may have regressed, and it is therefore safer to use a number much smaller than the observed regression.

Note that the injections described above are applied to a version of the subject application *different* from the version from which the baseline timelines were taken. More specifically, in eight out of the ten subject applications, the injections are applied to a version that is twenty GitHub commits ahead of the baseline commit. The only exceptions are Resource Space, where we used a build that is one minor version away from the base build, as it did not have a GitHub repository; and JSBin, where the injected version is only 17 commits away, as there were fewer than 20 commits ahead of the baseline build. This methodology is applied to more accurately reflect real-world investigation scenarios, where code changes apart from the regressing change are present; and as we will later highlight in Section 5.3, the impact of these extraneous code changes on the accuracy of ZAM is minimal.

For this research question, we measure four main metrics, described below

- *Node recall*: The number of ZAM outputs that correctly identify the modified function as the performance regression-cause, divided by the total number of ZAM outputs;
- *Path recall*: The number of ZAM outputs that correctly identify *at least* part of the call path that eventually calls the performance regression-cause (we call these *partial paths*), divided by the total number of ZAM outputs;
- *Node precision*: The number of leaf nodes in the ZAM output that match the modified function, divided by the total number of leaf nodes in the ZAM output;
- *Path precision*: The number of leaf nodes in the ZAM output that either match the modified function or are part of the partial path to the modified function, divided by the total number of leaf nodes in the ZAM output.

As a further clarification, consider the critical graph in Figure 4, and note that one of the full paths to a performance regression-cause found in this critical graph is as follows: `root` $\rightarrow$ `queryRenderedFeatures` $\rightarrow$ `rendered` $\rightarrow$ `query` $\rightarrow$ `layer` $\rightarrow$ `utf`. Any subpath of this path rooted at `root` is considered a *partial path*. Hence, path recall refers to the percentage of *critical graphs* generated by ZAM in our experiment that contain either a full path or a partial path to the performance regression-cause. In addition, path precision refers to the percentage of *all paths* across all critical graphs generated by ZAM that correspond to either a full path or a partial path to the performance regression-cause. Node recall and node precision are defined similarly, albeit only considering full paths.

We make a distinction between node recall/precision and path recall/precision because even if a regressing function is not *exactly* identified as a performance regression-cause by ZAM, identifying the path that leads to this function still points the investigator to the right direction and therefore, could still potentially be useful in helping the investigator identify the performance regression-cause more quickly. In this case, the value of identifying a partial path depends on how close it leads the investigator to the performance regression-cause, and for this reason, we also measure the average distance of all valid paths to the performance regression-cause, with smaller values interpreted as better (we call this metric the "average distance to cause").

Finally, we also determine the longest path from a top-level function to the performance regression-cause in each ZAM result, and we calculate the average of these longest paths. This metric provides an indication of how much effort the investigator may have had to exert if he or she had attempted to find the performance regression-cause manually, without the help of ZAM.

Effects of Aggregation (RQ2). For RQ2, we conduct the same experiment described above to answer RQ1, but with the number of aggregated timeline pairs varied, and ranging from 1 to 5.

Effects of Matching Algorithms (RQ3). For each run of ZAM performed in RQ1, we keep track of the number of times the LCS algorithm is used to perform node matching, versus the number of times our alternate matching algorithm is used. Our goal for this particular research question is to highlight the purpose served by each matching algorithm in our design.

Usefulness (RQ4). To answer RQ4, we measure the ratio of the number of nodes in each ZAM output to the total number of nodes in the execution timelines. For simplicity, the total number of nodes in the execution timelines is calculated from the first baseline timeline captured for each application.

Real Bugs (RQ5). In order to assess how ZAM fares when attempting to localize real-world performance regressions, we perform a small case study. In particular, we ran ZAM on three real-world performance regressions reported for three different GitHub projects: deck.gl[7], highlight.js[8], and async[9]. We chose a small number of performance regressions to give us more time to analyze in depth how ZAM behaves when used on these real-world regressions.

[7] https://github.com/uber/deck.gl/issues/1126
[8] https://github.com/isagalaev/highlight.js/issues/1031
[9] https://github.com/caolan/async/issues/883

TABLE 2 Evaluation results for ZAM, with the minimum response time difference μ set to 50 ms and the number of timeline pairs compared and aggregated set to 3.

Application Name	Node recall (%)	Path recall (%)	Node precision (%)	Path precision (%)	Average Distance to Cause	Average Longest Path	Total Paths in Base Comparison
Read the Docs	33.33	100.00	15.38	92.31	1.25	4.67	0
Resource Space	100.00	100.00	65.38	100.00	0.85	14.5	1
GNU Social	100.00	100.00	45.00	85.00	0.76	5.17	0
Live Helper Chat	100.00	100.00	28.22	99.59	5.48	22.17	0
Antville	100.00	100.00	71.43	100.00	1.11	6.17	0
FlaskBB	66.67	100.00	29.17	75.00	1.56	6.50	1
Searx	100.00	100.00	68.42	100.00	0.53	4.83	0
h5ai	66.67	100.00	62.32	89.86	1.35	3.00	0
JSBin	100.00	100.00	34.18	92.41	1.95	10.50	0
Selfoss	100.00	100.00	33.82	98.53	1.36	8.17	0
All	**86.67**	**100.00**	**41.16**	**96.09**	**2.92**	**8.57**	**2**

The performance regressions were selected by doing a GitHub search using the keyword `performance regression label:bug language:JavaScript`. To ensure that we have an oracle with which to compare the results output by ZAM, we only considered closed bug reports where the regression-introducing code change is either directly identified or can be inferred from either the initial report or the comments. Further, we only considered bug reports that sufficiently describe the user interactions that would reproduce the performance issue. The performance regressions eventually selected were the first three to appear in the search, not including issues that were filtered out using the conditions just described.

Note that the new timelines are generated using the build (or version) identified by the initial reporter as having the regression. Similarly, the old timelines are generated using either the build identified by the initial reporter as being a recent good build before the regression was introduced, or, if the reporter does not provide this information, any prior version of the product.

Real-World Applicability (RQ6). For this research question, we collect issues that SAP Analytics Cloud's performance and reliability team has worked on, in which ZAM was used. Each of these issues corresponds to a performance investigation (i.e., a performance regression is observed in our daily regression test, and we have to determine what caused this regression), and they are all archived in a bug tracking system. The issues collected span the period between September 2017 and May 2019, with the vast majority coming from the latter half of this time period as the tool became more mature for our team to use. To find these issues, we use the following filter: `description ∼ ‘‘Zam’’ OR summary ∼ ‘‘Zam’’ OR component in (‘‘Zam’’) OR comment ∼ ‘‘Zam’’`. In other words, we look for all the issues in which the keyword "Zam" appears in the description, summary, component value, or comment. Once we have all the issues satisfying this filter, we qualitatively analyze each matching issue to determine whether the resolution of the issue was client-side related; this is necessary, as our team generally runs ZAM at the beginning of a performance investigation as a sanity check, in case the root cause ends up being client-side related. We then categorize the remaining issues either as a *Success* if ZAM was able to find the performance regression-cause or a partial path to it, or a *Failure* if ZAM was not able to identify the performance regression-cause or a partial path to it.

Performance (RQ7). We added some instrumentation to the ZAM code to measure the execution time of each of ZAM's main modules. For each measurement, we report the average, median, 90th percentile, and maximum. Note that we executed ZAM on a machine that runs Windows 10 on a 3.60 GHz Intel Core i7-4790 processor, with 32 GB of RAM.

5.3 | Results

Accuracy (RQ1). Table 2 shows the results of our evaluation when ZAM is run on the subject applications, with the number of timeline pairs compared and aggregated set to 3. The results in this table show that overall in our experiment, ZAM had a node recall of 86.67% and a path recall of 100%. These results indicate that ZAM was able to *exactly* identify the performance regression-cause in the vast majority of its runs. In addition, *ZAM was able to identify at least a partial path to the performance regression-cause in **all** of its runs*. Based on our analysis of the ZAM outputs, the main source of inaccuracy likely comes from noise introduced by various factors, such as the reordering of some function calls and variance in performance numbers in some areas of the code, both of which lead to node mismatches. Nonetheless, in such cases, the identification of the partial path will at least provide a strong hint as to where the performance regression-cause is located.

The results in Table 2 also show that ZAM has an overall node precision of 41.16% and an overall path precision of 96.09%, with the number of timeline pairs compared and aggregated set to 3. These results indicate the following: while the leaf nodes identified by ZAM as a performance regression-cause do not always correspond to an actual performance regression-cause, ***nearly all*** *of these leaf nodes are either a performance regression-cause, or are a part of a partial path that leads to a performance regression-cause*. Therefore, the investigator will not have to waste much time having to "debunk" false positives, since the vast majority of the leaf nodes in ZAM's output will point the investigator to the performance regression-cause eventually. The false positives observed in our experiment belong to three separate categories:

- Most of the false positives that appear in the ZAM output correspond to the "(program)", "(idle)", and "(garbage collector)" markers, which can safely be ignored for the purposes of investigating client-side performance issues; these markers are functions artificially introduced by Chrome DevTools to help the investigator assess the application's end-to-end performance, and we decided to keep them as they can still help the investigator localize performance changes resulting from non-scripting-related causes (e.g., slower network requests, slower rendering, etc.);
- Some false positives are introduced by actual differences between the baseline commits and the injection commits; while these results are treated as false positives in this experiment by definition, *the fact that they represent legitimate performance regressions corroborate the real-world applicability of ZAM*, which is a question that we discuss further in RQ5 and RQ6;
- A small number of false positives also occur when the response time of an irrelevant function call varies by 50 ms or more, as this is the value we have set for μ.

It is also worth noting the results of the baseline comparison, in which one set of baseline timelines is compared against another set of baseline timelines, as described in Section 5.2. These baseline comparisons are expected to have *zero* paths in the critical graph generated by ZAM, so any path that appears is automatically a false positive. Overall, only two of the baseline comparisons had false positives – one for Resource Space and another for FlaskBB. For Resource Space, the false positive corresponds to the first type described above (i.e., the "(idle)" marker), while for FlaskBB, the false positive corresponds to the third type described above; in the latter case, the false positive eventually disappears when an aggregation of five timeline pairs is performed.

To estimate the manual effort required for the investigator to arrive at the performance regression-cause after looking at ZAM's output, we also measured the average distance from a leaf node in the partial path to the corresponding performance regression-cause (see "Average Distance to Cause" in Table 2). Our results indicate that *on average, the leaf nodes in the ZAM output are only 2-3 nodes away from the performance regression-cause*. This finding further corroborates our earlier assertion that partial paths are still useful in leading the investigator much closer to the performance regression-cause.

The second-last column in Table 2 shows the "average longest path", which is defined as the average length of the longest paths in each critical graph that lead to a performance regression-cause. In this case, the average longest path when considering all subject applications is 8.57 nodes; in other words, if the investigator were to compare the timelines manually, he or she would have potentially had to dig an average of 8-9 function calls deep or more, before arriving at the performance regression-cause; some applications have an even higher average, with Live Helper Chat at a depth of 22 function calls. Therefore, finding the performance regression-cause is not trivial, and requires one to not merely look at the top-level functions, but to actually dig deeper into the execution timelines. As our results indicate, ZAM is capable of automating this process, with high accuracy.

Lastly, the recall and precision values per injection type are very similar to the overall values in Table 2, and are therefore not included in this paper for simplicity.

Effects of Aggregation (RQ2). We now discuss the effects of varying the number of timeline pairs that are compared and aggregated. The results are summarized in the graphs in Figure 5, where the horizontal axes represent the number of timeline pairs. Based on the results, the node recall decreases as the number of pairs increases (Figure 5a), while the path recall remains completely flat at 100% (Figure 5b). The node recall result indicates that adding additional timeline pairs to the comparison may prevent some performance regression-causes to be exactly identified by our design because some may get filtered out inadvertently due to noise; however, based on the path recall result, *the partial paths that lead to these performance regression-causes still remain in the critical graph, even when the exact performance regression-causes are filtered out*.

With respect to the precision values, we can see in Figure 5c that the node precision decreases as more timeline pairs are compared; however, the path precision increases as more pairs are compared and aggregated, going from 91% when comparing only one pair of timelines, and increasing monotonically to 97% when comparing five pairs of timelines (Figure 5d). These two results indicate that *adding more timeline pairs to the comparison substantially reduces the number of false positives (i.e., leaf nodes that are neither performance regression-causes nor leading to one) while still retaining*

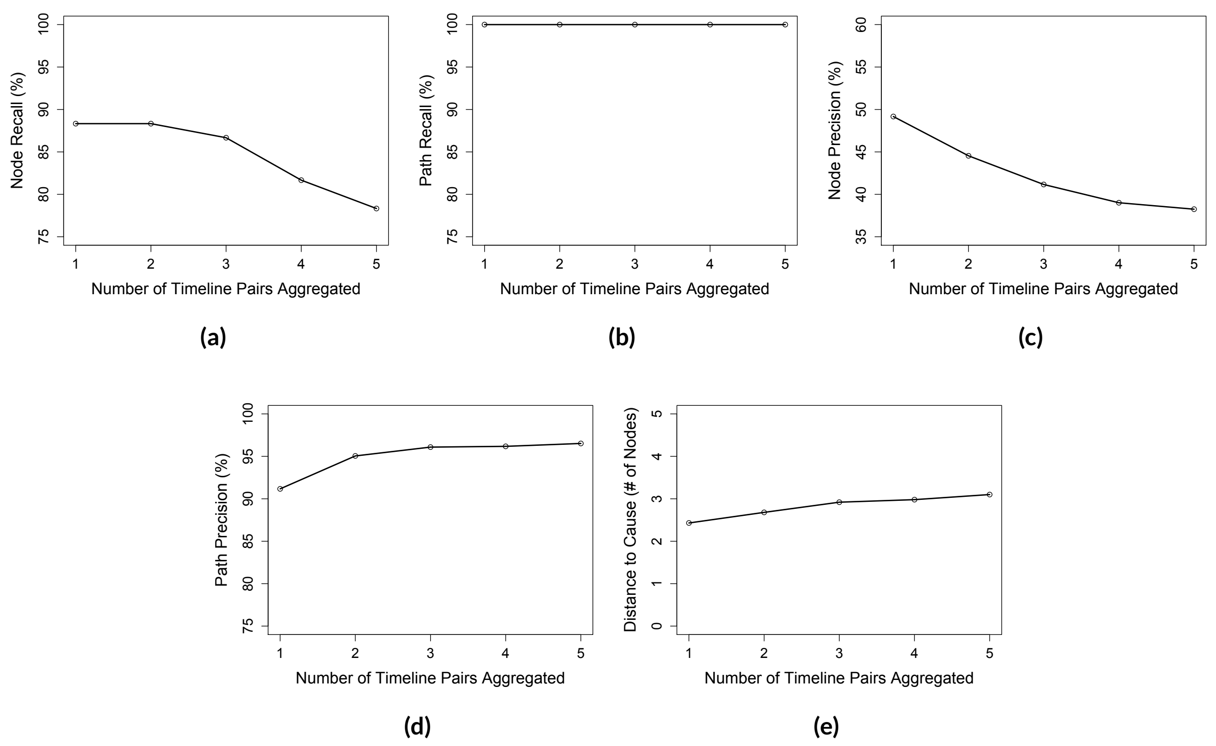


FIGURE 5 Graphs showing how accuracy changes with the number of timeline pairs compared. The following metrics are shown: (a) Node recall; (b) Path recall; (c) Node precision; (d) Path precision; (e) Average distance to the performance regression-cause. Note how the range of the vertical axes for (a), (b), and (d) is 75%-100%, while the range of the vertical axis for (c) is 35%-60%.

TABLE 3 Results showing the frequency of usage of each of the two matching algorithms used by ZAM.

Matching Algorithm	Total Uses	Average Uses Per Run	Percentage of Uses
LCS	6583	109.72	95.88%
Alternate	283	4.72	4.12%

almost all of the true positives. Notice, also, that the average distance to the performance regression-cause only increases slightly (Figure 5e), in this case ranging from 2.43 nodes when only one pair is compared to 3.10 nodes when five pairs are compared.

To summarize our findings in this subsection, adding more timeline pairs to compare and to aggregate will likely *decrease* the number of performance regression-causes that are exactly identified by ZAM as such; however, adding more pairs also *increases* the percentage of leaf nodes in the ZAM output that, at the very least, eventually leads to a performance regression-cause. Finally, increasing the number of pairs also does not significantly increase the average distance of the leaf nodes to the performance regression-cause in the ZAM output, if applicable. Therefore, when going from one pair to five pairs, ZAM's aggregation technique filters out many false positives (about 61% filtered out), while still retaining all of the true positives, at the small cost of slightly increasing the average distance to the performance regression-cause by around 0.7 nodes. We believe this result points to the overall effectiveness of our aggregation technique.

Effects of Matching Algorithms (RQ3). We summarize in Table 3 the overall usage of the two matching algorithms used by ZAM: LCS and the alternate matching algorithm. From this table, we can see that LCS succeeds in around 96% of all the node matchings, which in turn means that ZAM falls back to the alternate matching algorithm around 4% of the time. Note also that ZAM falls back to the alternate matching algorithm at least once in each run of the tool during our experiment. These results indicate that (1) the more accurate LCS algorithm, while computationally complex, is still

TABLE 4 Spearman correlation of each accuracy metric per injection versus the percent usage of the alternate matching algorithm per injection.

	vs. Node Recall	vs. Node Precision	vs. Path Precision
Spearman ρ value (compared to % usage of alternate matching algorithm)	-0.2280	0.1917	-0.1043

useful, as it is still used very frequently by the tool, and (2) our alternate matching algorithm is a necessary addition to our overall design, as it is able to act as a fallback in cases where LCS blows up.

To give us an idea of how much the usage of the alternate matching algorithm may have impacted the accuracy of the results, we computed the Spearman rank correlation's ρ value when comparing each of three accuracy measurements (node recall, node precision, and path precision) with the percent usage of the alternate matching algorithm, which is calculated as follows:

$$\frac{\text{\# of Times Alternate Matching Algorithm is Used}}{(\text{\# of Times Alternate Matching Algorithm is Used}) + (\text{\# of Times LCS is Used})} \times 100\%$$

This expression, along with the three accuracy metrics mentioned above, are calculated for each run of ZAM during our RQ1 experiment, which gives us arrays of data that we can correlate. Note that path recall is ignored as all its results are 100%, and a non-zero standard deviation is required to generate a Spearman ρ value. A positive correlation value indicates that an increase in usage of the alternate matching algorithm corresponds with an increase in accuracy, as defined by the specific metric; similarly, a negative correlation value indicates that an increase in usage of the same algorithm corresponds with a decrease in accuracy, again, as defined by the specific metric.

The results are summarized in Table 4. The main observation to make here is that the absolute correlation values themselves are not statistically significant, ranging from 0.10 to 0.23. The negative ρ values when comparing against node recall and path precision indicate (albeit not very strongly, given the small magnitudes of the correlations) that using the alternate algorithm may indeed have brought down the values of these two metrics. Such a result is expected, as the alternate matching algorithm is a more simplified version of LCS that trades off accuracy for speed, by virtue of the fact that the algorithm ignores the order of execution of the nodes to be matched. With that said, *the low correlation values indicate that this impact may not be as significant in practice.*

Usefulness (RQ4). Table 5 shows our evaluation results for RQ4. The second column represents the total number of function nodes in the execution timelines, while the third column represents the number of function nodes that appear in the critical graphs output by ZAM. Note that the numbers in these columns are summed over the results from the six injections in RQ1; hence, the average number of nodes per critical graph is not represented by the numbers on the second column, but rather, the numbers on the second column divided by 6. In addition, the fourth column represents the *compression ratio*, which we define in this context as follows:

$$\frac{\text{\# of Function Nodes in the Critical Graphs}}{\text{Total \# of Function Nodes in the Timelines}} \times 100\%$$

Therefore, the compression ratio gives an indication of how well ZAM is able to filter out irrelevant nodes from the execution timelines. Hence, smaller compression ratios mean the investigator will have to inspect fewer function nodes from the execution timeline, percentage-wise.

From Table 5, we can see that the compression ratio seen in most of our subject applications is less than around 2%, and the overall compression ratio is 2.34%. Therefore, paired with the results of RQ1, our design is able to significantly cut down the amount of effort required for the investigator to find the performance regression-causes – i.e., a much smaller number of nodes will need to be inspected after using ZAM.

Real Bugs (RQ5). We now present the results of our case study on real-world performance regressions.

Deck.gl: For this bug, the reporter observed that clicking on one of the settings (fp64) for polygon layers takes longer to process in the then-current version used by the reporter. Based on his manual investigation, the reporter determined that the performance regression-cause is related to the compilation of the shader, which, in this case, occurs at the `getProgramParameter` function. After running ZAM to localize this performance regression, it was able to find a partial path containing the same performance regression-cause. Furthermore, this partial path is only 8 nodes away from the performance regression-cause; in this case, none of these 8 nodes had any sibling nodes, which would allow the investigator to instantly

TABLE 5 Comparison of the total number of function nodes in the captured timelines with the number of nodes that appear in the ZAM output (with the minimum response time difference μ set to 50 ms and the number of timeline pairs compared and aggregated set to 3). The compression ratio is defined as third column divided by the second column, listed as a percentage. Smaller percentages indicate that ZAM is able to filter out a larger percentage of irrelevant functions nodes.

Application Name	Total # of Function Nodes	# of Function Nodes in ZAM Output	Compression Ratio (%)
Read the Docs	480	30	6.25
Resource Space	14412	272	1.89
GNU Social	3570	70	1.96
Live Helper Chat	25836	644	2.49
Antville	4752	107	2.25
FlaskBB	2490	49	1.97
Searx	3726	56	1.50
h5ai	5556	131	2.36
JSBin	13524	241	1.78
Selfoss	2478	194	7.83
All	**76824**	**1794**	2.34

pinpoint the performance regression-cause based on the partial path output by ZAM. Two false positives appeared in the output - the "(program)" and "(idle)" markers, which can be easily dismissed as discussed in the results for RQ1. We note a few more observations from this run of ZAM, listed below:

- The length of the partial path is 73 nodes, which is much longer than the average partial path length reported in Table 2. Given that the timelines captured for this application contain many branches and the difference in response time observed is under a second (around 900ms reported in the bug report, and around 100ms in our machine), localizing the performance regression by comparing 70+ node levels is very tedious and time-consuming. ZAM was able to perform the analysis in less than a minute;
- The builds from which the old and new timelines were captured are separated by 54 commits, which means many changes have been introduced to the application that are unrelated to the performance regression, and could therefore introduce noise. The fact that ZAM was able to locate the performance regression-cause under this condition demonstrates that ZAM works well even when there is a huge gap in commits between the builds being compared. This result also corroborates our results from RQ1, where the old and new timelines being compared are separated by 20 commits in most of the subject applications;
- The application contains hover events that encompass large portions of the viewport, thereby increasing the chances of additional noise being introduced as a result of these events. Again, ZAM was able to filter out these noises as a result of its aggregation;
- The performance regression-cause in this case does not correspond to the final root cause, which is in one of the path-layer shader files. However, setting a debugger breakpoint at the performance regression-cause and looking at the variable values for the shader types immediately reveals that the shader being used is related to path-layer. Therefore, the performance regression-cause provides a strong lead on the final root cause, eventually leading to the fix.

Highlight.js: The performance regression in this syntax highlighter application is observed when writing long class names – used as generics – in Java code. The reporter does not provide any suggestions for possible root causes in his initial description of the issue, but one of the developers eventually found that the root cause is a suboptimal regular expression in one component of the Java parser. When running ZAM for this performance regression, our approach was able to pinpoint the `exec()` function – which is responsible for evaluating the class name against the regular expression – as the performance regression-cause. The "(program)" marker described above once again appears in this run as a false positive. The following are some other valuable observations from this run:

- The developer was able to reproduce the issue on the same day the regression was reported, but no acknowledgement of the root cause was made until 15 days later, when the fix was provided. Of course, given the limited information the bug report comments provide, we cannot definitively conclude that this gap was caused primarily by difficulties in finding the performance regression-cause. However, in any case, ZAM was able to pinpoint the *exact* performance regression-cause in under a minute, which would have allowed the developer to quickly fix the issue the same day it was reproduced;

- The builds from which the old and new timelines were captured are separated by version updates, which contain plenty of commits in between. Once again, this result demonstrates ZAM's flexibility in finding the performance regression-cause even when many commits separate the old and the new timelines;
- The ZAM output also identifies the "(garbage collector)" marker as having regressed (since Chrome abstractly treats this marker as a function call), which helps the developer understand that the performance regression has an effect not only on the response time, but also on memory usage.

Async: Finally, for this performance regression, the reporter observed that the response time of calls to the `eachSeries()` function has increased significantly, by about 6 seconds. Once again, ZAM was able to exactly identify the performance regression-cause, as described in the bullet points that follow. The two false positives are once again the "(idle)" and "(program)" markers, which are easily dismissible as discussed earlier.

- Even though the regression is quite large in this example, the captured execution timelines themselves do not immediately make obvious the performance regression-cause, since the execution structure has changed after four minor version updates, and plenty of noise exists particularly in the timelines for the old build. However, ZAM was able to instantly pinpoint the `_delay()` function – which is aliased as "nextTick" in the code – as the performance regression-cause, since ZAM filters out extraneous noises from the timelines; this result matches up with the results eventually arrived at by the developers;
- In addition to the new execution structure, the additional six seconds in the response time are actually spread out over hundreds of calls to `_delay()`, which demonstrates why it is beneficial to use the top-down aggregated call tree when ZAM is running.

Real-World Applicability (RQ6). Here, we demonstrate the usefulness of ZAM in resolving real-world performance investigations. After applying the filter described in Section 5.2, we were able to collect a total of 73 tracked issues, corresponding to performance investigations in which ZAM was used by our performance and reliability team at SAP. Furthermore, after a qualitative analysis of each collected issue, we found that 22 of these had root causes unrelated to client-side JavaScript; thus, even though we ran ZAM for these 22 issues somewhere along the investigation for reasons described in Section 5.2, ZAM would not have been able to pick up on their root cause as ZAM is specifically designed to localize client-side JavaScript performance issues.

We therefore focused our analysis on the remaining 51 issues, and in doing so, we were able to categorize 43 as *Success* and 8 as *Failure*, where these categorizations are defined in Section 5.2; thus, *ZAM was able to successfully find the performance regression-cause, or a partial path to it, in over 84% of pertinent performance investigations*. We looked at all the issues labeled as *Failure* and found that ZAM did not succeed in localizing them for the following reasons:

- In 4 of the failed cases, the performance change was spread out across multiple *distinct* functions, with a small change in each function that accumulated to a larger one when all the performance changes are combined. We explore this limitation further in Section 6.1;
- In 2 of the failed cases, the performance regression-cause was located in the worker thread, which ZAM did not support at the time these investigations were conducted. Since then, we have added worker thread support in ZAM;
- For the remaining 2 failed cases, the reasons for failure are inconclusive, but we suspect that it is due to the fact that the response times only regressed by a very small amount in each issue (around 40-70ms), and the threshold μ may have been set to a value that is too large.

We should also note that in one of the issues labeled as *Success*, ZAM output a partial path to the performance regression-cause that was only one node in length; in this case, the investigator was not able to recognize it as a partial path, thereby causing him to abandon the ZAM result and find the performance regression-cause through some other means (it was only realized after the fact that ZAM did indeed output a partial path). However, for the remaining issues, the output of ZAM – the majority of which are partial paths – directly led the investigator to discover the performance regression-cause, and this information was used to find the regressing commit. These results indicate that ZAM is capable of helping investigators to accurately localize performance regressions.

Performance (RQ7). Finally, Table 6 shows the performance results for each major component of ZAM, with response times displayed in milliseconds (ms). As the table shows, most of the steps in ZAM take less than a millisecond to around 2ms to execute, on average. The exception is the "Comparison" phase, which takes around 9-10 seconds to execute on average; nonetheless, the maximum amount of time taken to complete this step was only a little over a minute (69.79s). Therefore, ZAM executes reasonably quickly, allowing the investigator to get results in under a minute on average per comparison.

TABLE 6 Performance results for ZAM. Note that 90th refers to the 90th percentile, and Max refers to the maximum.

Step Name	Response Times (ms)			
	Average	Median	90th	Max
File Extraction	0.75	1.00	2.00	3.00
Comparison	9186.43	3581.00	18726.80	69785.00
Aggregation	1.47	1.00	2.00	4.00
Output Generation	2.20	2.00	3.00	4.00

6 | DISCUSSION

Here we discuss some considerations when using ZAM, after which we discuss some of the limitations of our technique and some threats to validity.

With respect to the choice for the number of timeline pairs to input into our tool for comparison, our results for RQ2 suggest that increasing this number has an effect on accuracy; that is, including more pairs leads to greater path precision, albeit at the slight cost of being further away in node distance from the performance regression-cause. However, one additional factor that needs to be taken into consideration is the amount of time and effort it takes to record the execution timelines. From our experience, it generally takes about a minute to record one timeline if the series of user interactions are applied to the *initial* state of the web application (e.g., logon), which means it will take about two minutes to record a *pair* of timelines. In addition, there are many scenarios in which the investigator may have to apply the series of user interactions to some non-initial state, and the process of getting the web application to this non-initial state will take even more time, especially for large applications. Based on the results for RQ2, as well as our experience in recording execution timelines, we recommend that investigators using ZAM record anywhere between 2-4 timeline pairs as input. In particular, the numbers in this range would allow ZAM to generate outputs with around 95% path precision and near-perfect path recall (Figures 5d and 5b), while also allowing the investigator to record all the timeline pairs within only about 5-15 minutes or less, from experience.

Furthermore, the fact that ZAM is capable of handling noise in the timelines does allow some room for error on the investigator's part when interacting with the web application to record the timelines; as a matter of fact, some of the timelines we recorded in our evaluation did include some event handlers inadvertently being triggered (e.g., due to accidental hovers and clicks), although as the RQ1 and RQ2 results suggest, most of these errors got masked after the aggregation. Therefore, when recording timelines, investigators will be able to interact with the web application "normally", i.e., they would not have to worry too much about what handlers get triggered in their interaction, as ZAM is capable of filtering out the noise.

6.1 | Limitations

Currently, ZAM's analysis stops at finding the performance regression-cause. While this information is very useful in helping the investigator debug the performance regression, it does not directly inform the investigator how to fix the regression. It would be worthwhile to conduct some research on ways to automate the process of finding a repair for a performance regression, by analyzing the body of the functions identified by ZAM as a performance regression-cause. We include this consideration as part of future work.

In addition, as discussed earlier in this section, recording execution timelines can be effort-intensive. Extending ZAM to have the ability to automatically record these execution timelines given some pre-set workflow is ideal, as it would allow our tool to automate the *full* timeline comparison process. The main challenge here is in ensuring that the automated approach for recording the timelines does not interfere or tamper with the relative performance measurements for the function calls included in the timeline, particularly if an instrumentation-based approach were to be used. Once again, we leave this extension to future work.

While ZAM has been specifically implemented to analyze execution timelines captured from web applications, we believe certain aspects of the design may be applied to other domains and programming languages, as long as function call trees can be generated for them; for instance, the aggregation algorithm is agnostic to the execution characteristics of the application. As a matter of fact, designing our approach to work for JavaScript-based web applications allows it to be a more generalizable approach in some ways, since the call trees of major programming languages often start at only one top-level node and are less noisy, thereby making them "simplified versions" of the types of call trees generated from client-side JavaScript code.

With regards to our results for RQ6, while we found that ZAM was beneficial to the vast majority of real-world performance investigations we have conducted in our team, there were still occasions in which it was not able to identify the performance regression-cause. Half of these failed cases were due to the performance change being spread out across multiple distinct functions. One way to mitigate this issue is decreasing the threshold μ; however, this will come at the cost of potentially irrelevant functions being included in the output, especially if the variances for the response times are high. For future work, we will consider alternate ways of aggregating timeline data, such that these changes become more recognizable (e.g., aggregating by timeline region instead of just function names).

Lastly, ZAM is currently only capable of comparing the *response times* of functions; however, other performance metrics exist that may benefit from such a comparison, including CPU usage and memory usage. This limitation, however, pertains more to the implementation of ZAM rather than its design, as the design should be similar to the one described in Section 3. The only difference, in this case, will be the numbers compared. Furthermore, note that while the focus of our study is on analyzing end-to-end response times, which pertain to a single pass through a specific workflow/interaction per timeline taken, ZAM can still be used as a supplement when performing other types of performance tests. For example, when doing load testing, the investigator can run multiple instances of the same web application using tools such as LoadRunner [29] or Jmeter [3], and then take the execution timelines by manually interacting with one of those instances; ZAM can then be used to compare the timelines that are taken from this process.

6.2 | Threats to Validity

An external threat to validity is always present when considering a small – albeit reasonable – sample size for the subject applications. Nonetheless, we did our best to ensure generalizability by selecting our subjects from a large list containing many different types of web applications with various sizes, as Table 1 shows. In addition, note that we performed multiple injections per application, belonging to different injection types based on common performance issues identified in prior work [54], which allowed us to test on a greater variety of execution timelines.

In terms of internal threats to validity, we apply the same number of injections for each injection type (i.e., two each), instead of applying a number proportional to how frequently they appear in real scenarios among all issue types. From a purely theoretical standpoint, this may skew the overall recall and precision values; however, based on our results, no significant differences are observed across each injection type in terms of path/node recall and precision, and hence, we do not anticipate a significant difference in the overall results even if the proportions were altered. It is also worth noting that finding a precise ratio of the number of injections is difficult to do, given that multiple issue types can map to the same injection, and the study from which the injection types are based [54] provides no data on any overlaps.

Furthermore, in some cases, the end-to-end performance regressions observed in real-world web applications is spread out across multiple functions that regressed, and not just one, which may call into question our choice to inject only one function at a time. However, note that in our evaluation, there were many scenarios in which the function to which the for loop has been injected is called multiple times by multiple functions, and indeed, ZAM succeeded in finding many such multiple calls.

A construct threat to validity is in the node recall calculation, in which we consider a run of ZAM as successfully finding the injected performance regression if it identifies at least one call of the function as a performance regression-cause. In this case, if the modified function is called multiple times, and only one of those calls has been identified as a performance regression-cause, ZAM is still considered to have a node recall of 100%. Nonetheless, there are two things we would like to note. First, if a certain function regressed in performance, all the investigator really needs in most cases is for one call of that function to be identified as having regressed; once the function has been fixed, every call to that function – regardless of whether ZAM identified all of them or not – will most likely go back to its original performance. Second, we showed in our evaluation that ZAM *is* capable of finding multiple performance regression-causes pointing to the same regressing function.

Finally, another construct threat to validity is in the correlation measurements between the percent usage of the alternate matching algorithm when compared to accuracy metrics (RQ6). In particular, the correlation values themselves do not necessarily reflect the individual accuracy of each usage of the alternate matching algorithm; unfortunately, this individual accuracy is very difficult to quantify as no oracle is available, which would necessitate a manual inspection of all ~7000 node matching operations performed throughout the experiment. Nonetheless, it does point to how much the alternate matching algorithm may have *contributed* to the overall accuracy (i.e., precision and recall) of ZAM.

7 | RELATED WORK

We now describe prior work – both from research and industry – related to our proposed approach. We divide this section into four parts. First, we describe prior designs that attempt to localize the root cause of software bugs, with particular emphasis on performance issues. Next, we describe some industrial tools that carry out call tree comparisons. Lastly, we describe prior research on detecting performance issues in web applications.

7.1 Localizing Software Bugs

Due to the large body of work on software fault localization for functional bugs, this section focuses mainly on fault localization for web applications. In prior research, the first author of this work has proposed AUTOFLOX and VEJOVIS, which are automated techniques for localizing and repairing functional issues in client-side JavaScript, particularly those involving the DOM [38, 39]. Furthermore, the first author has worked on designs for detecting inconsistencies in JavaScript-based web applications that use Model-View-Controller (MVC) frameworks [40, 41]. In addition, Artzi et al. [4] and Samimi et al. [51] have both worked on techniques for automatically localizating HTML validation errors, while Pradel et al. [47] proposed an approach for analyzing type inconsistency in JavaScript. Other recent works have focused on memory leak localization in JavaScript [43] and race condition localization and detection [66, 32, 42]. Unlike our approach, these techniques do not look at performance issues in terms of web application response times, which generally use more complicated correctness metrics compared to functional bugs due in part to variance in the response time numbers.

7.2 Localizing Performance Issues

The goal of performance issue localization is to help the investigator find the root cause of a performance issue, or find information that leads the investigator closer to to this root cause. Several researchers have proposed various designs for performance issue localization, including Syer et al.'s approach for diagnosing memory-related performance issues using performance counters and logs [58] and Altman et al.'s technique for localizing the root cause of idle time in server applications [2]. Xu [63] has also proposed an approach for localizing performance issues, but with the analysis carried out at the design level instead of the code level. Unlike ZAM, none of these methods propose ways to localize performance regressions.

In terms of the localization of performance regressions, Malik et al. [25] previously worked on analyzing performance deviations in load tests; this analysis is not carried out at the code level, but rather, at the level of performance signatures of load test data. Heger et al. [18] also proposed an approach to isolate performance regression-causes based on performance-aware unit test cases. Finally, Nguyen et al. [34] and Luo et al. [23] designed their own automated techniques for finding performance regression-causes at the code level. Unlike ZAM, the aforementioned approaches rely on either a repository of past performance regression-causes or a suite of performance-aware unit tests, which are not always available. In addition, these approaches rely on a heuristic to pinpoint the performance regression-causes, while ZAM reasons its way through the timeline to find the cause. In this sense, ZAM's methodology is consistent with the results of an empirical study by Nistor et al. [36], which found that performance issues are fixed mainly through code reasoning.

7.3 Comparing Call Trees

While there are plenty of papers proposing designs for call tree construction, prior research on call tree comparisons is scarce. The only research papers we could find that compare call trees are those by Lhoták [22] and by Murphy et al. [31]; the latter comparison was done manually, while the former paper proposes an approach which performs an automated comparison. In both cases, the goal was to compare the effectiveness of various program analysis tools, and not to compare performance. In addition, the comparisons are done only with respect to the structure of the call trees, and the comparisons made are unidirectional, which means the structure of one graph is assumed to be the correct structure, while the other graph is compared against it, making the mapping of nodes simpler. In contrast, ZAM's comparisons are done with respect to both the structure and the content (e.g., performance numbers). Further, our comparison is *not* unidirectional, as the concept of a single correct execution does not apply to performance call trees, especially for an asynchronous language like JavaScript which can yield different call trees for different executions as demonstrated earlier.

In industry, there exists production tools that are able to compare performance profiling data, including Visual Studio [30], dotTrace [20], Yourkit [64], and JProfiler [12]. Visual Studio's comparator only compares functions or modules specified by the investigator in table view, and, unlike ZAM, does not analyze the performance call tree. The dotTrace, Yourkit, and JProfiler tools analyze performance call trees, but they differ from ZAM significantly in various ways. First, they do not perform filtration, which means they do not identify the performance regression-causes, but simply output the differences in performance numbers for functions in the call tree, in graph form; while this simplifies manual analysis in that only one graph needs to be considered instead of two, the number of functions to be analyzed remains roughly the same as in the purely manual case. Second, they do not perform aggregation, making them more prone to false positives, as demonstrated in Section 5. Lastly, they are designed to compare call trees generated from the execution traces of traditional programming languages (e.g., Java), which follow stricter programming semantics compared to JavaScript, are not event-driven, and are not minified; *this makes noise coming from extraneous function calls practically non-existent in these applications*, thereby making the node matching much simpler compared to web applications.

7.4 | Detecting Performance Issues

There is a large body of work on detecting performance bottlenecks in software [35, 7]. These works include techniques for finding bottlenecks based on repeating memory accesses [37], memoization opportunities [10], and low-utility data structures [62]. Shen et al. also propose an approach to find these bottlenecks using search-based profiling [56]. Similar techniques also exist for web applications, including those proposed by Selakovic and Pradel [54] and Gong et al. [14]. Further, Wang et al. have conducted empirical studies to analyze the bottlenecks of web page performance [59, 60]. While evidently useful, these approaches perform neither diagnosis nor localization of performance issues. Further, they do not specifically consider performance regressions, which differ from performance issues in the way their root causes are ascertained (i.e., the goal is not merely to find slow code, but to find code that became slower with respect to a baseline). ZAM, on the other hand, has localization as its main goal, and specifically targets performance regressions.

Maplesden et al. [28] introduce a technique that looks for repeated patterns in execution timelines, which allows their approach to reduce the size of an execution timeline. While both their approach and ZAM take similar inputs (i.e., execution timelines), their approach does not specifically identify the performance regression-cause when comparing pairs of timelines. In addition, their technique is focused on finding performance optimizations given one specific version of an application, as opposed to dealing with performance regressions between two versions of the same software, which tend to have smaller granularities in response time, and can therefore be more difficult to localize.

Lastly, there are many techniques that have been proposed whose aim is to automatically detect performance regressions [33, 57]. These techniques include Nguyen et al.'s approach for detecting regressions using statistical process control [33] and Malik et al.'s designs for tracking performance deviations from load test data [26, 27]. In some cases, the search for performance regressions is conducted in specific parts of the code, such as concurrent classes [45], thread pools [57], and I/O [6]. All of these approaches perform detection of performance regressions, but, unlike our proposed approach, they do not diagnose them to find performance regression-causes.

8 | CONCLUSION

In this paper, we presented ZAM, whose design allows for automated comparisons of execution timelines to localize performance regression-causes in web applications. ZAM takes pairs of execution timelines recorded from the client-side of web applications as input, and it outputs a graph showing the performance regression-causes, or partial paths to them. In our evaluation, we demonstrated that ZAM has high accuracy, achieving perfect path recall and a very high path precision (96.09%), even in the presence of noise coming from asynchronous function calls and event handlers in web applications. Further, ZAM succeeded in filtering out false positives from the critical graphs through its aggregation mechanism, thereby achieving greater path precision as more timeline pairs are included as input. ZAM is also useful in pruning out irrelevant function nodes from execution timelines, works well when applied to real-world performance regressions and investigations, and executes in under a minute on average. For future work, we would like to extend our approach to go beyond identifying performance regression-causes, and carry out source code analysis that would help the investigator easily identify fixes for the regression. We will also look into approaches for automatically generating execution timelines, which can be used to complement ZAM.

ACKNOWLEDGMENTS

We would like to thank members of the global performance and reliability team at SAP for helping us test and improve our design; special thanks go out to Jacques Buchholz, who supported the project. We would also like to thank Karthik Pattabiraman and Parry Fung for providing feedback on early drafts of this work.

References

[1] Ahmed, T. M., C.-P. Bezemer, T.-H. Chen, A. E. Hassan, and W. Shang, 2016: Studying the effectiveness of application performance management (apm) tools for detecting performance regressions for web applications: an experience report. *Proceedings of the International Conference on Mining Software Repositories (MSR)*, ACM, 1–12.

[2] Altman, E., M. Arnold, S. Fink, and N. Mitchell, 2010: Performance analysis of idle programs. *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, ACM, 739–753.

[3] Apache, 2020: *Jmeter*. https://jmeter.apache.org (Accessed: April 25, 2020).

[4] Artzi, S., J. Dolby, F. Tip, and M. Pistoia, 2010: Practical fault localization for dynamic web applications. *Proc. Intl. Conference on Software Engineering*, ACM, 265–274.

[5] Basques, K., 2018: *Performance analysis reference*. https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/reference#record (Accessed: July 2, 2018).

[6] Bezemer, C., E. Milon, A. Zaidman, and J. Pouwelse, 2014: Detecting and analyzing I/O performance regressions. *Journal of Software: Evolution and Process (JSEP)*, **26**, no. 12, 1193–1212.

[7] Burtscher, M., B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne, 2010: Perfexpert: An easy-to-use performance diagnosis tool for HPC applications. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, IEEE Computer Society, 1–11.

[8] Chen, J. and W. Shang, 2017: An exploratory study of performance regression introducing code changes. *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, IEEE Computer Society, 341–352.

[9] Daines, M., 2014: *viz-js*. http://viz-js.com/ (Accessed: February 19, 2018).

[10] Della Toffola, L., M. Pradel, and T. R. Gross, 2015: Performance problems you can fix: A dynamic analysis of memoization opportunities. *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, ACM, 607–622.

[11] dynaTrace Software GmbH, 2018: *Dynatrace*. https://www.dynatrace.com/ (Accessed: January 8, 2018).

[12] EJ Technologies, 2016: *JProfiler*. https://www.ej-technologies.com/products/jprofiler/whatsnew92.html (Accessed: February 19, 2018).

[13] Gansner, E. R., E. Koutsofios, S. C. North, and K.-P. Vo, 1993: A technique for drawing directed graphs. *Transactions on Software Engineering (TSE)*, **19**, no. 3, 214–230.

[14] Gong, L., M. Pradel, and K. Sen, 2015: JITprof: Pinpointing JIT-unfriendly JavaScript code. *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*, ACM, 357–368.

[15] Google, 2018: *Chrome DevTools Overview*. https://developer.chrome.com/devtools (Accessed: February 19, 2018).

[16] Graham, S. L., P. B. Kessler, and M. K. Mckusick, 1982: Gprof: A call graph execution profiler. *Proceedings of the SIGPLAN Symposium on Compiler Construction*, ACM, 120–126.

[17] Graphviz, 2018: *The DOT Language*. https://graphviz.gitlab.io/_pages/doc/info/lang.html (Accessed: February 19, 2018).

[18] Heger, C., J. Happe, and R. Farahbod, 2013: Automated root cause isolation of performance regressions during software development. *Proceedings of the International Conference on Performance Engineering (ICPE)*, ACM, 27–38.

[19] Irish, P., 2016: *DevTools Timeline Model*. https://github.com/paulirish/devtools-timeline-model (Accessed: February 19, 2018).

[20] JetBrains, 2018: *Comparing Profiling Data*. https://www.jetbrains.com/help/profiler/Studying_Profiling_Results__Comparing_Profiling_Data.html (Accessed: February 19, 2018).

[21] Kickball, 2018: *Awesome-Selfhosted*. https://github.com/Kickball/awesome-selfhosted (Accessed: January 30, 2018).

[22] Lhoták, O., 2007: Comparing call graphs. *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, ACM, 37–42.

[23] Luo, Q., D. Poshyvanyk, and M. Grechanik, 2016: Mining performance regression inducing code changes in evolving software. *Proceedings of the International Conference on Mining Software Repositories (MSR)*, ACM, 25–36.

[24] Maier, D., 1978: The complexity of some problems on subsequences and supersequences. *Journal of the ACM (JACM)*, **25**, no. 2, 322–336.

[25] Malik, H., B. Adams, and A. E. Hassan, 2010: Pinpointing the subsystems responsible for the performance deviations in a load test. *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, IEEE Computer Society, 201–210.

[26] Malik, H., H. Hemmati, and A. E. Hassan, 2013: Automatic detection of performance deviations in the load testing of large scale systems. *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 1012–1021.

[27] Malik, H., Z. M. Jiang, B. Adams, A. E. Hassan, P. Flora, and G. Hamann, 2010: Automatic comparison of load tests to support the performance analysis of large enterprise systems. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, IEEE Computer Society, 222–231.

[28] Maplesden, D., E. Tempero, J. Hosking, and J. C. Grundy, 2015: Subsuming methods: Finding new optimisation opportunities in object-oriented software. *Proceedings of the International Conference on Performance Engineering (ICPE)*, ACM, 175–186.

[29] Micro Focus, 2020: *LoadRunner*. https://software.microfocus.com/en-us/software/loadrunner (Accessed: April 25, 2020).

[30] Microsoft, 2015: *How to: Compare Performance Data Files*. https://msdn.microsoft.com/en-us/library/bb385753.aspx (Accessed: February 19, 2018).

[31] Murphy, G. C., D. Notkin, W. G. Griswold, and E. S. Lan, 1998: An empirical study of static call graph extractors. *Transactions on Software Engineering and Methodology (TOSEM)*, **7**, no. 2, 158–191.

[32] Mutlu, E., S. Tasiran, and B. Livshits, 2015: Detecting JavaScript races that matter. *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ACM, 381–392.

[33] Nguyen, T. H., B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, 2012: Automated detection of performance regressions using statistical process control techniques. *Proceedings of the International Conference on Performance Engineering (ICPE)*, ACM, 299–310.

[34] Nguyen, T. H., M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora, 2014: An industrial case study of automatically identifying performance regression-causes. *Proceedings of the International Working Conference on Mining Software Repositories (MSR)*, ACM, 232–241.

[35] Nistor, A., P.-C. Chang, C. Radoi, and S. Lu, 2015: Caramel: detecting and fixing performance problems that have non-intrusive fixes. *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 902–912.

[36] Nistor, A., T. Jiang, and L. Tan, 2013: Discovering, reporting, and fixing performance bugs. *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, IEEE Computer Society, 237–246.

[37] Nistor, A., L. Song, D. Marinov, and S. Lu, 2013: Toddler: Detecting performance problems via similar memory-access patterns. *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 562–571.

[38] Ocariza, F., G. Li, K. Pattabiraman, and A. Mesbah, 2016: Automatic fault localization for client-side JavaScript. *Software Testing, Verification and Reliability (STVR)*, **26**, no. 1, 69–88.

[39] Ocariza, F., K. Pattabiraman, and A. Mesbah, 2014: Vejovis: suggesting fixes for JavaScript faults. *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM, 837–847.

[40] — 2015: Detecting inconsistencies in JavaScript MVC applications. *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE Computer Society.

[41] — 2017: Detecting unknown inconsistencies in web applications. *Proceedings of the International Conference on Automated Software Engineering (ASE)*, IEEE Computer Society, 566–577.

[42] Petrov, B., M. Vechev, M. Sridharan, and J. Dolby, 2012: Race detection for web applications. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, ACM, 251–262.

[43] Pienaar, J. A. and R. Hundt, 2013: JSWhiz: Static analysis for JavaScript memory leaks. *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, IEEE Computer Society, 1–11.

[44] Pinto, F., U. Kulesza, and C. Treude, 2015: Automating the performance deviation analysis for multiple system releases: An evolutionary study. *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE Computer Society, 201–210.

[45] Pradel, M., M. Huggler, and T. R. Gross, 2014: Performance regression testing of concurrent classes. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, ACM, 13–25.

[46] Pradel, M., P. Schuh, and K. Sen, 2014: EventBreak: analyzing the responsiveness of user interfaces through performance-guided test generation. *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, ACM, 33–47.

[47] — 2015: TypeDevil: Dynamic type inconsistency analysis for JavaScript. *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 314–324.

[48] Ratanaworabhan, P., B. Livshits, D. Simmons, and B. Zorn, 2010: JSMeter: Measuring JavaScript behavior in the wild. *Proceedings of the USENIX Conference on Web Application Development (WebApps)*, ACM, 1–12.

[49] Research, A. L., 2018: *GraphViz*. http://graphviz.org/ (Accessed: February 19, 2018).

[50] Rosetta Code, 2017: *Longest common subsequence*. http://rosettacode.org/wiki/Longest_common_subsequence (Accessed: January 22, 2018).

[51] Samimi, H., M. Schafer, S. Artzi, T. Millstein, F. Tip, and L. Hendren, 2012: Automated repair of HTML generation errors in PHP applications using string constraint solving. *Proc. Intl. Conference on Software Engineering (ICSE)*, IEEE Computer Society, 277–287.

[52] Sandoval Alcocer, J. P., A. Bergel, and M. T. Valente, 2016: Learning from source code history to identify performance failures. *Proceedings of the International Conference on Performance Engineering (ICPE)*, ACM, 37–48.

[53] SAP, 2015: *SAP Analytics Cloud*. https://www.sap.com/products/cloud-analytics.html (Accessed: April 3, 2018).

[54] Selakovic, M. and M. Pradel, 2016: Performance issues and optimizations in JavaScript: an empirical study. *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM, 61–72.

[55] Shang, W., A. E. Hassan, M. Nasser, and P. Flora, 2015: Automated detection of performance regressions using regression models on clustered performance counters. *Proceedings of the International Conference on Performance Engineering (ICPE)*, ACM, 15–26.

[56] Shen, D., Q. Luo, D. Poshyvanyk, and M. Grechanik, 2015: Automating performance bottleneck detection using search-based application profiling. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, ACM, 270–281.

[57] Syer, M. D., B. Adams, and A. E. Hassan, 2011: Identifying performance deviations in thread pools. *Proceedings of the International Conference on Software Maintenance (ICSM)*, IEEE Computer Society, 83–92.

[58] Syer, M. D., Z. M. Jiang, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora, 2013: Leveraging performance counters and execution logs to diagnose memory-related performance issues. *Proceedings of the International Conference on Software Maintenance (ICSM)*, IEEE Computer Society, 110–119.

[59] Wang, X. S., A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, 2013: Demystifying page load performance with WProf. *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, USENIX, 473–486.

[60] — 2014: How speedy is SPDY? *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, USENIX, 387–399.

[61] Woodside, M., G. Franks, and D. C. Petriu, 2007: The future of software performance engineering. *Proceedings of the International Conference on the Future of Software Engineering (FOSE)*, IEEE Computer Society, 171–187.

[62] Xu, G., N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky, 2010: Finding low-utility data structures. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, ACM, 174–186.

[63] Xu, J., 2012: Rule-based automatic software performance diagnosis and improvement. *Performance Evaluation*, **69**, no. 11, 525–550.

[64] YourKit, 2018: *YourKit*. https://www.yourkit.com/ (Accessed: July 2, 2018).

[65] Zaman, S., B. Adams, and A. E. Hassan, 2012: A qualitative study on performance bugs. *Proceedings of the IEEE Working Conference on Mining Software Repositories (MSR)*, IEEE Computer Society, 199–208.

[66] Zhang, L. and C. Wang, 2017: RClassify: classifying race conditions in web applications via deterministic replay. *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 278–288.

How to cite this article: F. Ocariza and B. Zhao (2018), Localizing software performance regressions in web applications by comparing execution timelines, *Softw Test Verif Reliab., 2018;00:1–6*.