Noname manuscript No.
(will be inserted by the editor)

On the Effectiveness of Bisection in Performance Regression Localization

Frolin S. Ocariza, Jr.

Received: date / Accepted: date

Abstract Performance regressions can have a drastic impact on the usability of a software application. The crucial task of localizing such regressions can be achieved using *bisection*, which attempts to find the bug-introducing commit using binary search. This approach is used extensively by many development teams, but it is an inherently heuristical approach when applied to performance regressions, and therefore, does not have correctness guarantees. Unfortunately, bisection is also time-consuming, which implies the need to assess its *effectiveness* prior to running it. To this end, the goal of this study is to analyze the effectiveness of bisection for performance regressions. This goal is achieved by first formulating a metric that quantifies the probability of a successful bisection, and extracting a list of input parameters – the *contributing properties* – that potentially impact its value; a sensitivity analysis is then conducted on these properties to understand the extent of their impact. Furthermore, an empirical study of 310 bug reports describing performance regressions in 17 real-world applications is conducted, to better understand what these contributing properties look like in practice. The results show that while bisection can be highly effective in localizing real-world performance regressions, this effectiveness is sensitive to the contributing properties, especially the choice of baseline and the distributions at each commit. The results also reveal that most bug reports do not provide sufficient information to help developers properly choose values and metrics that can maximize the effectiveness, which implies the need for measures to fill this information gap.

Keywords Software performance · bisection · empirical study · bug localization

Conflict of Interest

Not Applicable

F. Ocariza
SAP Canada Inc.,
Vancouver, BC, Canada
E-mail: frolin.ocariza@sap.com

1 Introduction

A software performance regression[1] occurs when, after applying a series of code changes, the response time or resource usage metrics of an application degrades. Detecting and understanding such regressions is important, as they severely impact user experience (Zaman et al, 2012; Pradel et al, 2014; Selakovic and Pradel, 2016). While crucial, this investigation process is time-consuming, and many solutions have been proposed that aim to minimize the effort involved, both in research (Ocariza and Zhao, 2021; Graham et al, 1982; Ahmed et al, 2016; Nistor et al, 2015; Della Toffola et al, 2015) and in industry (YourKit, 2018; Microsoft, 2015; Dynatrace, 2018; Google, 2018). One such solution that is used in industry is *bisection*, which performs a binary search on the list of commits by testing the middle commit and recursively halving the list of commits depending on the result of the test. The search ranges from the last commit observed to not include the regression (the *good commit*) to the first commit observed to include the regression (the *bad commit*), and continues recursively until the first commit to manifest the regression (the *root cause* or the *bug-introducing commit*) is found.

Bisection is typically performed in the context of localizing functional regressions. It is useful for such regressions because the comparison metric is binary – i.e., the middle commit being tested is either functioning correctly based on some well-known specification, or it is not. In addition, due to the simplicity of this comparison metric, its behaviour is typically monotonic – i.e., the relevant component functions correctly from the good commit to the commit right before the root cause, and functions incorrectly from the root cause to the bad commit.

Unlike functional regressions, the metrics used to assess if a particular commit has a performance regression are neither binary nor monotonic, due to variance in the performance numbers. Nonetheless, bisection can still be intuitively applied to performance regression localization; in particular, the performance of a particular commit can still be measured and thereby compared to some baseline number. For instance, if logging on to a web application used to take around 2 seconds, but now takes around 10 seconds after a performance regression has been introduced, a commit can be tagged as "buggy" if its logon response time exceeds some value, say, 2 seconds. In fact, several companies already use bisection to localize performance regressions, including our performance team at SAP (Ocariza, 2020), as well as Google (2021a) and Microsoft (2018), among others. Unfortunately, while commonplace, *no studies have been conducted that attempt to understand the* ***effectiveness*** *of bisection when applied to real-world performance regressions, to the best of our knowledge.*

Understanding the effectiveness of bisection in real-world settings is beneficial, for various reasons. First, as mentioned above, there is variance in performance numbers, which means that bisection, when applied to performance regressions, is inherently heuristical; thus, due to its probabilistic nature, it would be useful to both quantify the likelihood that bisection will output the correct commit, and understand the conditions under which this likelihood is high. Practitioners can then use these findings to determine whether bisection is a suitable solution when localizing specific performance regressions, and if so, determine the parameters

[1] For simplicity, we will also refer to software performance regressions simply as *performance regressions*

with which to configure the bisection. Second, understanding the effectiveness also has research value, as it can allow researchers to formulate enhancements to the base bisection algorithm based on the results.

In this paper, an empirical study of over 300 bug reports from 17 popular GitHub projects is conducted, with each bug report describing a software performance regression. The goal of the study is to understand the most important properties that lead to an effective bisection, and to analyze how well these properties translate in real-world settings. In particular, the following contributions are made:

- The formulation of an *effectiveness measure* that can be used to quantify the probability of a successful bisection. This measure can be used to quantify the effectiveness of any bisection that involves performance regressions;
- An analysis of the main input properties that contribute to the effectiveness of a performance regression bisection. We call these the *contributing properties*;
- A quantitative and qualitative analysis of these contributing properties in practice, based on 310 bug reports describing software performance regressions.

The results from the above analysis show that the effectiveness of a bisection on performance regressions is impacted primarily by the choice of baseline value and the characteristics of the probabilistic distributions describing the good commit and the bad commit. Unfortunately, the study on the bug reports also indicates that most performance regressions reported in real-world applications do not contain sufficient information to make a proper baseline assessment. To a lesser degree, the length of the commit range is also shown to impact the effectiveness; however, based on the empirical study, bug reports tend to provide version numbers for the commit range instead of hashes, and these version numbers typically correspond to longer commit ranges as the results also show. Lastly and somewhat counter-intuitively, the study also reveals that the effectiveness can be sensitive to the transition index – i.e., the suspected location of the bug-introducing commit – which implies that it would be useful to measure the effectiveness of a bisection both before and after it executes.

2 Background and Motivation

This section goes into greater detail on what bisection is in the context of localizing software regressions in general, and describes the benefits of using it to localize performance regressions in particular. It also describes the challenges involved with bisecting such performance regressions; these challenges provide the primary motivation for this study.

2.1 Bisection

In order to understand how bisection can help localize software regressions in general, we will use an example. Suppose a software developer is running daily regression tests on an application. On Day 1, the regression test runs on Commit 1, which is the then-latest commit, and reports no failures. However, on Day 2, the regression test runs on Commit 99, and reports a failure, as shown in Figure 1.

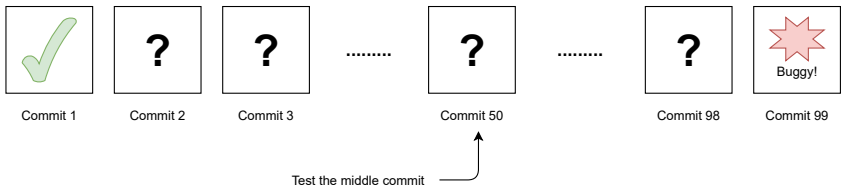


Fig. 1: An example of a software regression localization problem, with Commit 1 known to function properly, Commit 99 known to be buggy, and the rest of the commits untested. The goal is to find the first commit to manifest the bug. When performing bisection, Commit 50 – the middle commit – will be the first commit to test, and the search will continue recursively either on the left side or the right side, depending on the test results.

Having made this observation, the goal of the developer is to determine which commit introduced the bug, with the initial range of possible bug-introducing commits ranging from Commit 2 to Commit 99, inclusive.

One way to conduct this search is by using *bisection*. In binary search fashion, bisection will first take the middle commit – in this case, Commit 50 – and run the regression test on it. If the regression test passes, this likely indicates that the bug-introducing commit can be found on the right side of the array of commits, so the search space will be reduced to Commits 50-99. If, on the other hand, the regression test fails, this likely indicates that the bug-introducing commit can be found on the left side, and the search space will be reduced to Commits 1-50. This entire process will proceed recursively until there are only two commits remaining, at which point the second of these two commits is output as the bug-introducing commit.

The metric used to determine whether to continue the search on the left side of the array or the right side of the array is called the *bisection metric*. In the case of functional regressions, the metric used is "correctness" (i.e., does the test output the correct value or not, based on the specifications?). On the other hand, for performance regressions, the metric used is numerical, and the decision to either take the left side or the right side of the array is based on a comparison with a *baseline* value.

2.2 Bisecting Performance Regressions

Using bisection to localize performance regressions has found its way into industry (Ocariza, 2020; Google, 2021a; Microsoft, 2018), which is not too surprising as it provides many advantages. First of all, performance regressions are often difficult to localize using traditional debugging techniques, as many of these techniques are intrusive with respect to performance; for instance, setting breakpoints will change the underlying response time of an application. In contrast, bisection is minimally intrusive, as it simply runs performance regression tests without any external interference (e.g., from the developer). In addition, as with functional regressions,

performance regression bisection only cares about the final result of the regression test, which allows the developer to abstract out the details of why the performance regressed, and defer answering that question until the bisection provides an output; thus, the human effort required is reduced from a search-and-validation problem to simply a validation problem, which itself is simpified by a post-facto analysis of the code changes in the commit output by the bisection. Further, bisection is simple and intuitive when applied to performance bugs, with bisection paths taken based on a simple comparison with a baseline value; this simplicity is particularly important when validating the output of the bisection. Lastly, performance regression tests – especially end-to-end tests – often take a long time to execute since many samples of a performance metric need to be collected in order to generate a statistically significant result; as a result, performance regression tests often cannot be run on a per-commit basis, which means a search needs to be conducted across multiple commits when localizing a performance regression; bisection provides an efficient (i.e., $\mathcal{O}(\log n)$) way to conduct this search.

While advantageous, bisection also introduces additional challenges when applied to performance regressions. Unlike functional regressions, the bisection metric used to determine the bisection path is a numerical value with (often high) variance. Therefore, the choice of a bisection metric is much more crucial for performance regressions, both in terms of its stability and the baseline value used. As a corollary to the high variance, while monotonicity of the bisection metric is not guaranteed even for functional regressions, the bisection metric for performance regressions is almost guaranteed to *not* be monotonic, which once again makes choosing the baseline value crucial. Finally, as discussed further in Section 2.3, the amount of time it takes to run a performance regression bisection can be very high; hence, it would be useful to know if running a bisection is worth the investment, and if so, what parameters are needed to minimize the chances of a failed bisection. These challenges in accuracy and time cost point to the need for a better understanding of the effectiveness of bisection on performance regressions.

2.3 Goal and Motivation

The main goal of the study is twofold. First, it aims to identify and analyze the main properties that contribute to the effectiveness of a bisection; in this paper, we refer to these properties as the *contributing properties*. Second, the study also aims to understand what these contributing properties look like in reported, real-world performance regressions, both qualitatively (i.e., does the bug report provide information regarding these properties?) and quantitatively (i.e., if the information is provided, what is the magnitude?).

Addressing the first goal will help developers easily assess whether bisection is a suitable technique to use for specific performance regression localization problems. Making this assessment is very important, as performance regression tests – especially end-to-end regression tests – can take a long time to run, which means bisection itself takes a long time to execute, as it will have to run *several* of these tests; thus, running a failed bisection can be very costly. For instance, in recent test parallelization work conducted by Olianas et al (2021), their evaluation results show single iteration end-to-end tests running anywhere between 30-160 seconds without test parallelization, and 15-71 seconds with test parallelization. Since per-

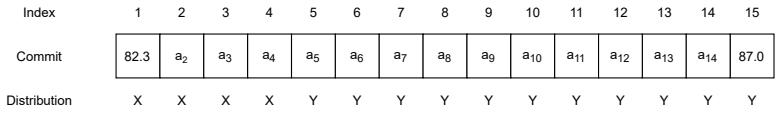


Fig. 2: An example of a performance regression, represented by 15-tuple $\mathcal{T}(82.3, 87.0, 1, 15)$

formance regression tests require multiple iterations in order to establish statistically significant values to compare against the baseline, these numbers will be even higher – e.g., with 50 iterations, the tests will take 25-133 minutes without test parallelization, and 12.5-59 minutes with test parallelization. Of course, these values also do not take into account the amount of time it takes to deploy or install a particular version of an application – which can take several minutes for large applications – as well as the fact that these tests have to be run multiple times for multiple versions of the application throughout the bisection. In this case, given a particular performance regression to localize, having a deeper understanding of the effectiveness of bisection and its contributing parameters will allow the developer to know whether bisection is an appropriate solution in the first place, and if so, what bisection metric and baseline to use that will maximize the effectiveness.

With regards to the second goal, it will allow us to identify information gaps in reported performance regressions that would prevent developers from making the above assessment. From these findings, recommendations on how to best fill these gaps can then be provided, based on the qualitative and quantitative analyses.

2.4 Scope and Running Example

To lay out the scope of this study and to help explain the effectiveness measure and its derivation in the next section, this subsection introduces a running example based on a real-world performance regression (Microsoft, 2021). This performance regression was found in .NET Runtime, with the response time of one of the tests (`System.Text.Encodings.Web.Tests.Perf_Encoders`) increasing from a mean of 82.3 ns to a mean of 87.0 ns, based on multiple iterations of the same test. Here, the historical data show the pre-regression standard deviation to be 0.42 ns, and after the regression, the standard deviation increased to 1.59 ns. Furthermore, there are a total of 15 commits between the good commit and the bad commit, inclusive. This regression is represented as an array in Figure 2.

In this study, we consider applications that run in variable execution environments. As seen in the running example above where the variance is comparatively high – especially after the regression is introduced – this can lead to variations in individual performance measurements, thereby necessitating multiple iterations of the same test. It is assumed that in such a system, either the aggregate measures (e.g., mean, 90th percentile, etc.) are stable enough for the developer or tester to manually recognize that a performance regression has occurred, or the development team has at their disposal some mechanism to automate this assessment (e.g., change point detection).

The running example also does not adhere to any performance specifications, and the same applies to all applications included as part of this study. In practice, this sitution arises when such specifications (e.g., service level objectives) do not exist; however, more commonly, it also arises when the development team is doing daily monitoring to ensure that specific performance measurements do not unnecessarily deviate from an established baseline, in order to remain as far away as possible from specified performance boundaries, if any.

With regards to bisection itself, note that the algorithm introduced in Sections 2.1 and 2.2 is the simplest type of bisection – i.e., naïve bisection. Performing the analysis on a version of bisection that is stripped to its bare essentials is beneficial, as it allows development teams to make practical decisions on what enhancements to the base bisection algorithm apply best for them, and therefore expend their resources on improving. For instance, suppose a team is working on an application that is more prone to large performance changes than smaller ones, but whose performance tests suffer from long execution times. In this scenario, it would be more reasonable for them to inherit the simple baseline comparison used in naïve bisection, as it makes the results easier to interpret, and then spend time trying to minimize the bisection runtime by using techniques such as selective bisection debugging (Saha and Gligoric, 2017). If, on the other hand, the test runtime is fast, but the application itself is more prone to small, gradual regressions, then it may be more worthwhile trying to enhance the baseline comparison, especially in light of the results presented in Section 5. In this case, the team can consider using more probabilistic algorithms such as noisy binary search (Karp and Kleinberg, 2007) or multisection (Keenan, 2019), at the cost of a potentially longer bisection. In summary, we offload these decisions to the development team, while providing the results as an aid to this practical assessment.

3 Effectiveness Measure

We will now derive a metric that measures the effectiveness of a bisection when applied to a performance regression, which we will refer to in this paper as the *effectiveness measure*. We first discuss some intuitions that directly lead to an informal definition of effectiveness. Thereafter, we will formalize the concepts of *performance regression* and *bisection*, which also naturally lead to a formal definition for *effectiveness* that is in line with the intuition presented. Finally, we will derive an algorithm to compute this effectiveness measure using Bayesian analysis.

3.1 Intuitions

In order to formulate a useful definition for effectiveness, we first go over two basic intuitions. The first intuition is that when we speak of a particular bisection being "effective", the context in which we make this claim is that the bisection has provided the correct bug-introducing commit; this context of course is in line with and follows directly from the goal of performance regression localization, which is precisely to find this bug-introducing commit. Based on this intuition alone, we can provisionally define the "effectiveness" of a bisection as *the probability that the bisection outputs the correct commit*. However, this definition remains ambiguous,

and we still need to define what it means for the bisection to "provide the correct commit". In order to arrive at this more refined definition, we turn to the second intuition.

In this case, the second intuition is that when localizing performance regressions, the bisection metric for each commit – which is a numerical metric as discussed in Section 2.1 – will follow a specific distribution given repeated measurements of that same metric, with some aggregate value of interest that describes this distribution – say, the mean, for illustrative purposes. Now, bisection assumes in the first place that everything before the bug-introducing commit has roughly the same mean M or smaller, and everything from the bug-introducing commit onwards will have a mean greater than M. Given that this property required by bisection holds, the goal of bisection would then be to find the first commit where the distribution changes – in particular, from one with a mean of M or smaller, to one with a mean greater than M.

Therefore, taking the above two intuitions together, we can refine our provisional definition for effectiveness as follows: *the probability that the bisection outputs the first commit (in the array of commits) where the distribution shifts.* The definition can be rewritten more concretely in terms of a conditional probability: *the probability that the first commit (in the array of commits) where the distribution shifts is commit C, given that bisection output C.* This informal definition will form the basis for our more formal definition presented in the remainder of the section.

3.2 Definitions

As is evident in the running example in Figure 2, a performance regression can be modeled as a tuple (i.e., array) of real numbers, with the good commit indexed at p and the bad commit indexed at q. We will denote such tuples as $\mathcal{T}(s, t, p, q)$, where s is the baseline value to compare against, stored at index p; and t is the performance measure at index q. The remaining elements in the middle are equal to real numbers of unknown value, as the commits represented by those elements are untested.

Definition 1 (Performance Regression) *A* performance regression *is an n-tuple $\mathcal{T}(s, t, p, q)$ such that $t > s$ and $q - p + 1 = n \geq 2$. Each element of a performance regression will also be referred to in this paper as a* commit.

In the running example, the baseline s can be set to any value less than 87.0 ns, but for illustrative purposes, we will set $s = 83.6$, which is around standard deviations away from the mean performance at the good commit. Furthermore, the value t is equal to 87.0, and n is equal to the number of commits, which is 15. The index p can be initialized to 1, and the index q can be initialized to 15.

Before we define bisection, we first make the following definitions to simplify our notation.

Definition 2 *Let $\mathcal{A} = \{\mathcal{T}(s, t, p, q) \mid t > s,\ q > p\} \cup \{[r] \mid r \in \mathbb{R}\}$. In other words, $\mathcal{A}$ is the set of all performance regressions* and *1-tuples.*

Definition 3 (Halving Function) *Let the* halving function *be a function $h\colon \mathcal{A} \to \mathcal{A}$, such that, for all $x \in \mathcal{A}$ where x is either a 1-tuple or a performance regression $\mathcal{T}(s, t, p, q)$:*

$$h(x) \equiv \begin{cases} \mathcal{T}(s, a_m, p, m) & a_m > s \text{ and } q - p > 1 \\ \mathcal{T}(s, t, m, q) & a_m \leq s \text{ and } q - p > 1 \\ [t] & q - p = 1 \\ x & x \text{ is a 1-tuple} \end{cases}$$

where a_m is the value of the element in the tuple $\mathcal{T}(s, t, p, q)$ at the middle index $m = \lfloor \frac{p+q}{2} \rfloor$.

Intuitively, one can think of the halving function as one iteration of a bisection. For instance, in the running example, if the middle element a_8 has a performance measurement greater than the baseline 83.6, the left half of the array remains; otherwise, the right half remains, with the value on the left-most index replaced with the baseline, to ensure that we are always comparing against the same baseline.

On the above note, we formally define bisection as follows.

Definition 4 (Bisection) *A* bisection *is a function $B \colon \mathcal{A} \to \mathbb{Z}^+$ such that for all $x \in \mathcal{A}$ where x is either a 1-tuple or a performance regression $\mathcal{T}(s, t, p, q)$:*

$$B(x) \equiv \begin{cases} B(h(x)) & q - p > 1 \\ q & q - p = 1 \\ i & x \text{ is a 1-tuple indexed at } i \end{cases}$$

In other words, bisection is basically a repeated application of the halving function on a performance regression x. This process continues recursively until the input to the bisection is either a 2-tuple (in which case it outputs the index of the second element) or a 1-tuple indexed at i (in which case it outputs i).

Lastly, based on our discussion in Section 3.1, we need to model the distribution (of a given bisection metric) for each commit in the performance regression. Recall from Section 3.1 that one of the underlying properties required by bisection, when applied to a performance regression, is that all commits prior to the bug-introducing commit follow roughly the same distribution with mean M (for example), and all commits from the bug-introducing commit onwards also follow another distribution with mean greater than M. Based on this intuition, given a performance regression $\mathcal{T}(s, t, p, q)$, we will model the distribution of the commits from index p to index $r - 1$ with the same continuous random variable X, and we will model the distribution of the commits from index r to index q with the same continuous random variable Y, where r is the index of the bug-introducing commit. However, note that the goal of bisection is to *find* the bug-introducing commit, so its location is initially unknown. Naturally, we can formally define the bug-introducing commit as a random variable, as follows:

Definition 5 (Bug-Introducing Commit) *Given a performance regression $x = \mathcal{T}(s, t, p, q)$, we define the* bug-introducing commit *D_x of x as a discrete random variable whose value represents the index of the first element in x where the distribution shifts from X to Y – i.e., the index of the first element in x that has distribution Y. This means that D_x has $\{p+1, ..., q-1, q\}$ as its sample space, as the first element of the performance regression (indexed at p) is assumed to follow distribution X.*

Once again, we turn to the running example in Figure 2 to concretize the definition just presented. In this example, the distribution from index 1 to index

4 is X, and the distribution shifts to Y at index 5. Thus, if we represent this performance regression with the variable x, then by Definition 5, $D_x = 5$ in this example. To reiterate, the bug-introducing commit is unknown to us initially, so $D_x = 5$ is simply one of several – in this case, fourteen – possible values for the random variable D_x.

Based on the above definitions, as well as the provisional definition presented in Section 3.1, we can formally define effectiveness as follows.

Definition 6 (Effectiveness Measure) *Given a performance regression* $x = \mathcal{T}(s, t, p, q)$*, the* effectiveness measure *(or simply,* effectiveness*)* $\epsilon_{x,c}$ *of* x *with respect to a bisection output* c *is defined as*

$$\epsilon_{x,c} \equiv P(D_x = c \mid B(x) = c)$$

As stated in Section 3.1, the above measure corresponds to the probability that the bug-introducing commit in the performance regression occurs at index c, given that bisection itself output index c. Note that this effectiveness measure is applicable only to a very specific output c of the bisection. In the next subsection, we will eventually define an aggregated measure that is averaged over all possible outputs of the bisection, which is useful as it allows us to measure the effectiveness of a particular bisection independent of the bisection output. However, we will use the above definition for effectiveness as the starting point of our derivation.

3.3 Derivation of the Effectiveness Measure

Now that we have formally defined the effectiveness measure, our next task is to derive an algorithm to compute it. As a preliminary observation, note that we can rewrite the effectiveness measure as follows, using Bayes' theorem:

$$\epsilon_{x,c} = \frac{P(B(x) = c \mid D_x = c)P(D_x = c)}{P(B(x) = c)} \tag{1}$$

This rewritten form provides us with a general strategy and framework for our derivation, as it suffices to compute the following individual probabilities, which are taken from the right-most side of the above identity.

$$\begin{cases} P(B(x) = c \mid D_x = c) & \text{Likelihood } (\lambda) \\ P(D_x = c) & \text{Shift Probability } (\sigma) \\ P(B(x) = c) & \text{Output Probability } (\omega) \end{cases}$$

Likelihood. First, we will compute $\lambda = P(B(x) = c \mid D_x = c)$ for a given performance regression $x = \mathcal{T}(s, t, p, q)$; here, we refer to λ as the *likelihood*. This value represents the probability that bisection outputs an index c, given that the bug-introducing commit is at that same index. To get an intuitive understanding of how to derive this value, we turn once again to the running example. Figure 3 shows a probability tree diagram of the running example, depicting the different possible branches that bisection can take, given that the bug-introducing commit is at index 5. Notice that at each step of the bisection, the branch taken depends on the value of the middle commit. For instance, in the first step, the left branch

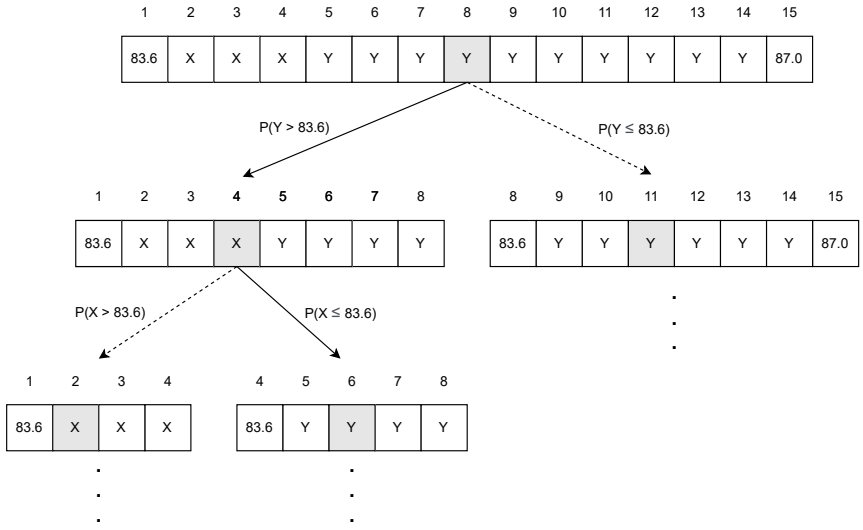


Fig. 3: Probability tree diagram showing different branches that the bisection algorithm can take, given that the bug-introducing commit is at index 5. Solid arrows represent branches that lead to index 5. Note that the value at index 1 is the baseline used in the comparison.

is taken if the middle commit at index 8 exceeds the baseline value – which has probability $P(Y > 83.6)$, since index 8 has distribution Y – while the right branch is taken otherwise, with probability $P(Y \leq 83.6)$. The tree diagram can be completed recursively using this same process, with leaves representing the final output of the bisection.

In this particular scenario, given that the bug-introducing commit is at index 5, we are interested in computing the probability that bisection also outputs index 5. To compute this value, we can take the branches in the tree that lead to this index, which are represented by solid arrows in Figure 3, and find the product of their corresponding probabilities – i.e., the product of $P(Y > 83.6)$, $P(X \leq 83.6)$, $P(Y > 83.6)$, etc. Thus, the likelihood λ will take on the form of a recursive product of cumulative distribution function (CDF) values of each subsequent middle commit.

Based on this intuition, it would be useful to have some way to refer to the distribution at each commit, so we introduce the following notation.

Definition 7 *Consider a performance regression $x = \mathcal{T}(s, t, p, q)$. We define $Z_i^{D_x=c}$ as the distribution at commit i of x, given that $D_x = c$ (i.e., given that the distribution shifts from X to Y at commit c, as per Definition 5), where $p \leq i \leq q$. In other words:*

$$Z_i^{D_x=c} \equiv \begin{cases} X & p \leq i < c \\ Y & c \leq i \leq q \end{cases}$$

We can now proceed with the derivation for the likelihood. First, from elementary probability, we can rewrite λ as follows, noting that $Z_m^{D_x=c} > s$ and $Z_m^{D_x=c} \leq s$ are mutually exclusive events, where $m = \left\lfloor \frac{p+q}{2} \right\rfloor$ is the midpoint.

$$\begin{aligned}\lambda \quad &= P(Z_m^{D_x=c} > s \mid D_x = c)P(B(x) = c \mid D_x = c, Z_m^{D_x=c} > s) \\ &\quad + P(Z_m^{D_x=c} \leq s \mid D_x = c)P(B(x) = c \mid D_x = c, Z_m^{D_x=c} \leq s)\end{aligned}$$

In essence, the above expression splits the probability into two mutually exclusive cases: one in which the result of the middle commit is greater than the baseline s, and another in which the result of the middle commit is less than or equal to s, as illustrated earlier in the running example.

Now, given that the value of the middle commit exceeds the baseline (i.e., $Z_m^{D_x=c} > s$), we know from Definitions 4 and 3 that the following holds, where a_m is the value at the middle commit.

$$B(x) = B(\mathcal{T}(s,t,p,q)) = B(h(\mathcal{T}(s,t,p,q))) = B(\mathcal{T}(s,a_m,p,m))$$

Similarly, given that the value of the middle commit is less than or equal to the baseline (i.e., $Z_m^{D_x=c} \leq s$), then we have the following case, by Definitions 4 and 3.

$$B(x) = B(\mathcal{T}(s,t,p,q)) = B(h(\mathcal{T}(s,t,p,q))) = B(\mathcal{T}(s,t,m,q))$$

Thus, in the expression for λ, we can expand $B(x) = c$ according to the relevant case.

$$\begin{aligned}\lambda \quad &= P(Z_m^{D_x=c} > s)P(B(\mathcal{T}(s,a_m,p,m)) = c \mid D_x = c) \\ &\quad + P(Z_m^{D_x=c} \leq s)P(B(\mathcal{T}(s,t,m,q)) = c \mid D_x = c)\end{aligned}$$

Note that some conditionals have been removed, as $Z_m^{D_x=c} > s$ and $Z_m^{D_x=c} \leq s$ are independent from $D_x = c$ (based on Definition 7), and the values of $B(\mathcal{T}(s,a_m,p,m))$ and $B(\mathcal{T}(s,t,m,q))$ no longer depend on the value of the commit at index m, as is evident from Definitions 4 and 3.

Finally, from Definition 4, we know that the bisection function always outputs a value between the two endpoints of the performance regression tuple, not including the left endpoint. This means that if $c \leq m$, then the index c would lie outside the possible values of $B(\mathcal{T}(s,t,m,q))$; hence, in this scenario, $P(B(\mathcal{T}(s,t,m,q)) = c \mid D_x = c) = 0$. Similarly, if $c > m$, then c would lie outside the possible values of $B(\mathcal{T}(s,a_m,p,m))$ and thus, in this scenario, $P(B(\mathcal{T}(s,a_m,p,m)) = c \mid D_x = c) = 0$. Given these observations, we can rewrite λ piecewise, with $P(Z_m^{D_x=c} > s)$ and $P(Z_m^{D_x=c} \leq s)$ written in terms of the cumulative distribution function $F_{Z_m^{D_x=c}}(s)$.

$$\begin{aligned}\lambda &= P(B(\mathcal{T}(s,t,p,q)) = c \mid D_x = c) \\ &= \begin{cases} (1 - F_{Z_m^{D_x=c}}(s))P(B(\mathcal{T}(s,a_m,p,m)) = c \mid D_x = c) & c \leq m \\ (F_{Z_m^{D_x=c}}(s))P(B(\mathcal{T}(s,t,m,q)) = c \mid D_x = c) & c > m \end{cases}\end{aligned} \tag{2}$$

Note how unrolling the above recursive expression leads to the product of probabilities encountered earlier when discussing the running example. Programmatically, Equation 2 allows us to compute λ recursively, taking either the first case or the second case depending on the value of the midpoint m.

Shift Probability. Next, we turn our attention to the *shift probability*, which we denote by $\sigma = P(D_x = c)$. In this case, given no additional information, we will model D_x as a uniformly distributed random variable, with each of its possible values having equal probability. As mentioned in Definition 5, the sample space of D_x is $\{p+1, ..., q-1, q\}$ for a given performance regression $\mathcal{T}(s,t,p,q)$. Thus, we can compute σ as follows.

$$\sigma = P(D_x = c) = \begin{cases} \frac{1}{q-p} & p < c \leq q \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Output Probability. Lastly, we need to compute the *output probability*, denoted by $\omega = P(B(x) = c)$. The main observation to make here is that, given $x = \mathcal{T}(s,t,p,q)$, the events $D_x = p+1$, $D_x = p+2$, ..., $D_x = q-1$, $D_x = q$ are all mutually exclusive and encompass the entirety of the sample space for D_x (Definition 5). Thus, we can write ω as follows.

$$\omega = P(B(x) = c) = \sum_{i=p+1}^{q} P(D_x = i) P(B(x) = c \mid D_x = i) \tag{4}$$

In the above equation, the value of $P(D_x = i)$ for each i can be computed using Equation 3. The value of $P(B(x) = c \mid D_x = i)$ can be computed using the following equation, whose derivation is omitted here as it is very similar to the derivation for the likelihood (i.e., Equation 2).

$$P(B(\mathcal{T}(s,t,p,q)) = c \mid D_x = i) = \begin{cases} (1 - F_{Z_m^{D_x=i}}(s)) P(B(\mathcal{T}(s,a_m,p,m)) = c \mid D_x = i) & c \leq m \\ (F_{Z_m^{D_x=i}}(s)) P(B(\mathcal{T}(s,t,m,q)) = c \mid D_x = i) & c > m \end{cases}$$

It is worth noting that if D_x is modeled to be uniform, as we have done in Equation 3, then $P(D_x = i)$ stays constant for all $p < i \leq q$. Therefore, combining Equations 1 and 4, we can also simplify the effectiveness measure calculation as follows.

$$\epsilon_{x,c} = \frac{P(B(x) = c \mid D_x = c)}{\sum_{i=p+1}^{q} P(B(x) = c \mid D_x = i)}$$

Average Effectiveness. Given Equations 2, 3, and 4, we can now compute the effectiveness $\epsilon_{x,c}$ for a given output index c. In the running example for instance, given an output index of 5 and a baseline of 83.6, the effectiveness $\epsilon_{x,5}$ is 95.3%. However, since we do not know the output of the bisection prior to running it, it would be more desirable to measure the overall effectiveness of bisection on a given performance regression x regardless of the output. In this paper, we will use the average effectiveness as our aggregate measure, and it is defined as follows.

Definition 8 (Average Effectiveness) *Given a performance regression $x = \mathcal{T}(s,t,p,q)$, we denote the* average effectiveness *of bisection $B(x)$ by $\epsilon_{x,avg}$, and define it as follows:*

$$\epsilon_{x,avg} \equiv \sum_{i=p+1}^{q} \epsilon_{x,i} P(B(x) = i)$$

The above definition is based on the expected value of the effectiveness, given the probability of each possible output of $B(x)$; note that $P(B(x) = i)$ can be computed for each i using the equation for the output probability (Equation 4). The following simplification, based on Equation 1 and the fact that D_x is uniform, can help speed up the computation of $\epsilon_{x,\text{avg}}$.

$$\epsilon_{x,\text{avg}} = \frac{1}{q-p} \sum_{i=p+1}^{q} P(B(x) = i \mid D_x = i)$$

From the above expressions, the average effectiveness for the running example computes to 96.9%, which happens to be a bit higher than the effectiveness only for output index 5.

In the empirical study presented in subsequent sections, we will be computing the effectiveness of a particular bisection based on the average effectiveness. Thus, in the remaining sections, we will use the terms "effectiveness" and "average effectiveness" interchangeably, whenever the context is clear.

3.4 Contributing Properties

With a formal definition of effectiveness in place, we can now identify the *contributing properties*, which are the properties of a performance regression that can potentially impact the effectiveness of a bisection. These contributing properties provide the skeleton for the empirical evaluation, whose methodology is described in Section 4 and whose results are presented in Section 5.

Contributing Property #1: Baseline. As seen in Equations 2 and 4, the values of the likelihood and the output probability depend on the cumulative distribution function (CDF) of the distribution at each commit (e.g., Equation 2 depends on $F_{Z_m^{D_x=c}}(s)$). The CDF calculations have, as their input, the value of the baseline s. This dependency indicates that the baseline value is a contributing property.

Contributing Property #2: Distributions. Since the likelihood and output probability both depend on the CDFs as just mentioned, then the value of effectiveness is also potentially impacted by the characteristics of the distribution before the regression (i.e., X) and the distribution after the regression (i.e., Y). In turn, this observation implies that the distributions X and Y are contributing properties. Note that the characteristics of the distribution that are primarily analyzed in this study are the mean and the standard deviation, as these are used to compute the CDF.

Contributing Property #3: Commit Range Length. The length of the commit range is also a contributing property as it affects the value of the likelihood and the output probability. In particular, note that Equation 2 is computed recursively, and the number of recursive iterations depends on the commit range length. The commit range length also affects the value of the shift probability, as is evident in Equation 3; however, as discussed earlier, the shift probability gets cancelled out from the effectiveness measure computation when D_x is uniform.

Contributing Property #4: Transition Index. Lastly, from Equations 2 and 4, we can see that the value of the bisection output c also has an impact when computing the likelihood and the output probability. We will call this value the

transition index. Note that this value will *not* affect the average effectiveness, as the bisection output is abstracted out from the average effectiveness calculation as per Definition 8. However, it is still worthwhile studying the effect that this value has on the "per bisection output" effectiveness $\epsilon_{x,c}$, as $\epsilon_{x,c}$ can still be used for post-facto analysis. In particular, while $\epsilon_{x,\text{avg}}$ can help developers answer, "How effective will this bisection be?" prior to running the bisection, $\epsilon_{x,c}$ can help developers answer, "How effective was the bisection that just ran?"

4 Experimental Methodology

This section describes in detail the methodology used to achieve the goals enumerated in Section 2.3. We will start by going over the overall strategy, and then a description of the research questions follows thereafter.

4.1 Overall Strategy

For each of the contributing properties identified in the previous section, we ask the following two overarching questions, which correspond to the two goals discussed in Section 2.3. These overarching questions are further broken down into more granular research questions in Section 4.2.

1. What impact does this contributing property have on the effectiveness of the bisection?
2. What characteristics does this contributing property have in practice?

To answer the first question, a more thorough analysis of the effectiveness measure is performed, studying the effects of varying the value of the contributing property. In this particular study, we use a one-at-a-time (OAT) sensitivity analysis to determine the impact of each individual contributing property. This analysis entails choosing nominal values for each contributing property; thus, when analyzing a particular contributing property, the value for that contributing property is varied, and the other contributing properties are set to their nominal values. These values are chosen based on (1) common values encountered in practice, using ranges seen by our performance and reliability team at SAP as a guideline, and (2) computation speed, as larger values tend to make the computation of the effectiveness measure take longer. The nominal values are listed below.

- For the *commit range length*, we choose a nominal value of **100**. As a point of comparison, the twenty most recent bisections conducted in our team for which we still have data on the commit range had a median commit range length of 136.5, which is very close to the chosen nominal value. These bisections were run by seven members of our team, four of whom have at least 5 years of experience each in performance engineering, one with about 2 years of experience, and two with 7 months of experience.
- For the distributions, the nominal value for the *mean* is randomly chosen **between 1000 and 10000**, and the nominal value for the *standard deviation* is randomly chosen **between 100 and 1000**. Of the 32 core operations that our team is primarily monitoring, 25 have means that fall within the specified

Table 1: Software applications from which bug reports were collected, listed alphabetically

Application Name	Application Type	# of Bug Reports Collected	Search Term	Source
ansible	IT Automation System	10	`performance regression is:closed`	Red Hat (2021a)
cockroach	Database	26	`performance regression is:closed`	Cockroach Labs (2021)
elasticsearch	Search Engine	20	`performance regression is:closed`	Elastic NV (2021)
flutter	Mobile SDK	30	`performance regression is:closed`	Google (2021b)
kubernetes	Container Management	30	`performance regression is:closed`	Google (2021c)
moby	Container Management	8	`performance regression is:closed`	Moby Project (2021)
nixpkgs	Software Packages	6	`performance regression is:closed`	NixOS (2021)
node	Runtime Environment	30	`performance regression is:closed type:issue`	Dahl (2021)
origin	Container Management	10	`is:closed label:area/performance`	Red Hat (2021b)
roslyn	Compiler	7	`performance regression is:closed type:issue`	Microsoft (2021b)
runtime	Runtime Environment	30	`performance regression is:closed`	Microsoft (2021a)
rust	Programming Language	30	`performance regression is:closed`	The Rust Foundation (2021)
servo	Web Browser Engine	8	`is:closed label:I-perf-slow`	Mozilla Corporation (2021)
tensorflow	Machine Learning Framework	21	`performance regression is:closed`	Google (2021d)
tgstation	Role Playing Game	15	`is:closed label:Performance type:issue`	Exadv1 (2021)
vscode	Source Code Editor	24	`performance regression is:closed`	Microsoft (2021c)
wp-calypso	Web Application Front-End	5	`is:closed label:Performance type:issue`	Automattic (2021)

range (in units of milliseconds), with the remaining 7 just above or just below the boundaries, based on a week's worth of data. Similarly, 26 of the 32 core operations fall within the range chosen for the standard deviation, with the remaining 6 falling just below 100, within the same time span. These values were observed by 11 members of our team, six of whom have at least 5 years of experience each in performance engineering, two with 2-3 years of experience, and three with 3-7 months of experience.

Note that randomly choosing the distribution parameters from a range of values instead of fixing them simulates what one would normally encounter in practice, where the distribution shapes are quite varied; with that said, to increase generalizability, we consider multiple pairs of distributions in our analysis. We do not need to set a nominal value for the transition index for any of our analyses, as most of the research questions consider average effectiveness, where this property is abstracted out. In addition, we do not need to set a nominal value for the baseline, because in most research questions, we only consider the baseline that produces the *maximum* effectiveness value; the set of baseline values considered when trying to ascertain this maximum ranges from the mean of distribution X to the mean of distribution Y, in intervals of 10.

For the second question, we base our answers on an analysis of performance regression bug reports for real-world software applications. These software applications are taken from the list of the top 100 most valuable GitHub repositories, as compiled by Hacker Noon and ranked based on a reputation algorithm by Gaviar (2019). More specifically, the highest ranked applications were taken from this list, omitting any applications that did not contain an "Issues" tab in GitHub, as well as any applications that did not have any performance regressions reported, based on the search criteria described shortly. A maximum of 30 bug reports were collected for each application, and the collection was halted once the total exceeded 300 bug reports. The number of bug reports was capped, as analyzing each one required significant effort, with each bug report requiring 10-20 minutes to be analyzed on average; thus, a balance between the effort involved and the number of samples analyzed needed to be made. Following this process, a total of 310 bug reports were collected from 17 applications, which are listed in Table 1. To be clear, the goal of this second overarching question is not to evaluate the effectiveness of bisection on real-world bug reports, but to understand the contributing properties

that would help developers assess how effective a bisection will be for their specific use case.

In most cases, the search term used to find the bug reports was `performance regression is:closed`. However, in cases where this search term did not provide too many results, it was altered either by relaxing some of the keywords or including labels or tags as part of the search. Table 1 shows the search terms used for each application's repository. Furthermore, only closed bug reports that are acknowledged by the developers as a performance regression are considered; this acknowledgement can either be explicitly made in the comments, or implicitly made by the presence of a fix or a recommended resolution that still implies a regression (e.g., fixed in a later version, accepting the performance hit, etc.).

4.2 Research Questions

We break down the two overarching questions from Section 4.1 into the following research questions.

RQ1 (Impact of Baseline): What impact does the choice of baseline have on the effectiveness?
RQ2 (Baselines in Practice): Do real-world performance regression bug reports provide enough information to assess a proper baseline value?
RQ3 (Impact of Distributions): What impact do the distributions have on the effectiveness?
RQ4 (Distributions in Practice): How close (or far apart) are the pre-regression distribution X and the post-regression distribution Y from each other in real-world performance regressions?
RQ5 (Impact of Commit Range Length): What impact does the length of the commit range have on the effectiveness?
RQ6 (Commit Range Lengths in Practice): How long are the commit ranges provided in real-world performance regression bug reports?
RQ7 (Impact of the Transition Index): Does the position of the transition index affect the effectiveness?
RQ8 (Transition Indices in Practice): How much does effectiveness vary per transition index in real-world performance regressions?
RQ9 (Effectiveness in Practice): How effective will bisection be when applied to real-world performance regressions?

The first eight questions are based on the strategy laid out in Section 4.1. In this case, the real-world analysis of a particular contributing property is asked immediately after the analysis on its impact; the research questions are organized in this manner in order to make it easier to link the theoretical analysis with its corresponding real-world analysis. The last research question allows us to assess the overall effectiveness of the real-world performance regressions from a black-box perspective (i.e., without looking at the contributing properties).

Note that in this study, we consider three different types of distributions for the pre-regression distribution X and the post-regression distribution Y, namely (1) normal, (2) log-normal, and (3) Pareto. We consider normal distributions in this list as they are well-studied, potentially making the analysis simpler and more interpretable. Furthermore, normal distributions are often used in scenarios where

the distribution is not known, which applies to this experiment as none of the bug reports provide raw performance data, but only aggregated metrics (e.g., this applies to the running example). Having said this, prior research has shown that performance measurements tend to be more heavy-tailed than normal (Crovella et al, 1998; Sasaki et al, 2017; Crovella, 2000; Chen et al, 2014). For this reason, we also consider two common heavy-tailed distribution types as part of the analysis, namely log-normal and Pareto. The latter, in particular, has been observed in various workload scenarios (Harchol-Balter, 2013; Akinshin, 2019). While the experiments were conducted on all three distribution types for all pertinent research questions, the full results for log-normal and Pareto are presented only for RQ1, as the three distribution types yielded very similar results for the remaining research questions.

Impact of Baselines (RQ1). To analyze the impact of the choice of baseline, we keep the commit range length and the distributions to their nominal values, as described in Section 4.1, while varying the baseline from the mean of X to the mean of Y, which allows us to plot a graph of the baseline value versus the average effectiveness. We consider 1000 different pairs of distributions, applied to each of the three distribution types.

Baselines in Practice (RQ2). In order to identify the choice of baseline that maximizes the effectiveness (the properties of which are analyzed in the preceding research question), developers need to be provided enough information to compute the effectiveness in the first place. Thus, the goal of this research question is to determine if the information provided in bug reports is sufficient to make a proper baseline assessment, and if not, find ways to either fill in or mitigate these gaps. To answer this research question, we first categorize the commit range provided in each bug report as follows:

- **Full (F)**: Version information (i.e., commit hash, version number, or test number) is provided for both the good commit (i.e., the first commit in the performance regression) and the bad commit (i.e., the last commit in the performance regression);
- **Half (H)**: Version information is provided only for the good commit or the bad commit, but not both;
- **None (N)**: Version information is not provided at all

Correspondingly, we also categorize the distribution provided in each bug report as follows:

- **Full (F)**: Complete distribution information (i.e., mean and standard deviation) is provided for both the good commit and the bad commit
- **Half (H)**: Complete distribution information is provided only for the good commit or the bad commit, but not both;
- **None (N)**: Complete distribution information is not provided at all

Based on the above categories, we classify a bug report based on the nine possible combinations, each labeled as **AB**, where **A** is the commit range category and **B** is the distribution category. For example, a bug report classified as **FF** means its commit range is categorized as "Full" and its distribution is categorized as "Full"; the running example falls under this classification, as complete information is included. On the other hand, a bug report classified as **HN** means its commit range is categorized as "Half", and its distribution is categorized as "None".

Lastly, note that version information or distribution information is considered "provided" if the information comes from the reporter of the bug, either by explicitly including the information in the report, or by including data that allows the developer to easily infer the information (i.e., links that contain the information, or raw numbers and graphs that allow the developer to easily compute the mean and/or standard deviation).

Impact of Distributions (RQ3). Here, we perform three analyses. First, we perform the OAT analysis on the distance between the means. To do this, we keep the commit range length and the standard deviation to their nominal values (which include a randomly chosen standard deviation as described in Section 4.1). Further, we fix the mean of X at 1000, and we vary the mean of Y from 1000 to 10000. Based on these values, we can generate a graph of the distance between the means of the distributions versus the maximum effectiveness (i.e., the largest effectiveness value from any baseline). In total, we analyze the results for 1000 randomly chosen pairs of standard deviation values, for each of the three distribution types.

Second, we perform another OAT analysis, this time on the standard deviations. In this case, we keep the commit range length and the means at their nominal values (with a distance of 1000 between the mean of X and the mean of Y, for simplicity), with the standard deviation varied in three ways: (1) vary only the standard deviation of X; (2) vary only the standard deviation of Y; (3) vary the standard deviations of both X and Y. In each case, the standard deviation value will vary between 100 and 1000, similar to its range of nominal values.

Third, we keep the commit range length at its nominal value, and we randomly generate the means and the standard deviations for both X and Y; from these values, we then calculate both the maximum effectiveness and the overlapping coefficient. The overlapping coefficient is defined as the area of overlap between two distributions (Inman and Bradley Jr, 1989; Weitzman, 1970) – in this case, between the distribution X at the good commit and the distribution Y at the bad commit. This procedure is repeated 200 times, after which we compute the correlation between the maximum effectiveness and the overlapping coefficient.

Distributions in Practice (RQ4). For this question, we gather all the bug reports whose distributions are categorized as "Full", and we compute the overlapping coefficient for each one. Doing so allows us to measure the "closeness" of the distributions, which we can interpret in the context of the results from RQ3.

Impact of Commit Range Length (RQ5). To assess the impact of the length of the commit range, we randomly generate the parameters for the distributions X and Y according to their range of nominal values, and vary the length of the commit range from 2 to 300, after which we compute the maximum effectiveness. This procedure is carried out for 1000 different pairs of distributions for each of the three distribution types.

Commit Range Lengths in Practice (RQ6). The goal of this research question is twofold. First, we would like to determine typical commit range lengths in real-world performance regressions. Second, we would like to analyze the various ways in which developers provide commit ranges, and determine whether they correlate in any way with the commit range length. To perform this analysis, we gather all bug reports whose commit range is categorized as "Full", and categorize them as follows.

- **Hash**: The commit range is given by the reporter as GitHub commit hashes;

- **Version**: The commit range is given by the reporter as software version numbers;
- **Test Run Number**: The commit range is given by the reporter as build numbers of a regression test;
- **Mix**: The commit range is given by the reporter as any combination of the above types

In addition, for each bug report whose commit range is "Full", we also record the commit range length wherever it is possible to do so. For the running example, the commit range categorization is "Hash" as the GitHub commit hashes are provided, and the commit range length is 15. Note that in some cases, the commit range length can no longer be computed due to lost information resulting from repository migration, version no longer being available, etc. Thus, only bug reports for which the commit range length can be computed are included in this specific analysis.

Impact of the Transition Index (RQ7). Unlike the other contributing properties, the sensitivity analysis on the transition index focuses on its impact on the "per bisection output" effectiveness $\epsilon_{x,c}$, instead of the average effectiveness $\epsilon_{x,\text{avg}}$. To carry out this analysis, we will keep the commit range length and the distributions to their nominal values, while varying the value of the transition index from 2 to 100 (i.e., the range of possible transition index values, according to the sample space of D_x). In addition, the baseline is set to the optimal baseline – i.e., the baseline that produces the maximum average effectiveness. As with previous research questions, this procedure is carried out for 1000 different pairs of distributions for each of the three distribution types, thereby producing different graphs showing the transition index c versus the value of $\epsilon_{x,c}$.

Transition Indices in Practice (RQ8). To conduct this analysis, we gather all the **FF** bug reports (i.e., bug reports with full commit range and distribution information provided by the reporter) whose commit range length can be inferred, and generate the "transition index versus $\epsilon_{x,c}$" graphs for each one, similar to what is done for RQ7.

Effectiveness in Practice (RQ9). Finally, for this research question, we gather all the **FF** bug reports whose commit range length can be inferred and compute the maximum (average) effectiveness for each one. The results are presented in a table, and any interesting patterns observed are identified. In addition, to demonstrate the real-world applicability of the effectiveness measure, the bisections conducted in our performance and reliability team are gathered; as of writing, there are 40 bisections for which we still have corresponding data on the effectiveness measure. The effectiveness measure is then compared against the success (or failure) of the bisection, and the comparison is presented in a chart.

5 Results

The results of the analyses described in the preceding section are now presented. The focus of this section is to present observations from the results, with a more thorough discussion on their implications in Section 6. Raw data for the results have also been made available.[2]

[2] `https://people.ece.ubc.ca/frolino/projects/perf-bisect-effectiveness/`

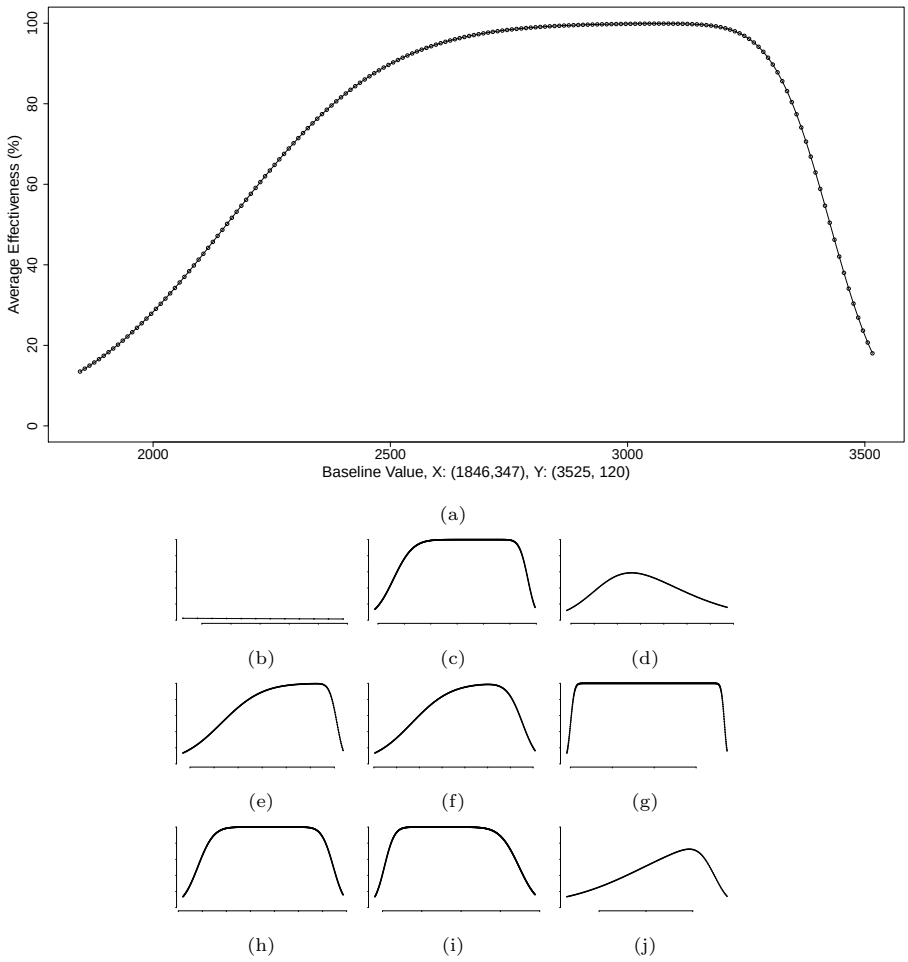


Fig. 4: Results of the OAT analysis for the impact of the baseline on the average effectiveness, for ten randomly chosen pairs of normal distributions X and Y. The horizontal axis refers to the baseline, ranging from the mean of X to the mean of Y. The vertical axis refers to the effectiveness measure, ranging from 0% to 100%. The axis labels for all graphs are similar to those in (a).

5.1 Impact of Baseline

Figure 4 shows the results of the OAT sensitivity analysis on the baseline (RQ1); in particular, it shows the graph of the baseline versus the average effectiveness for the first ten randomly chosen pairs of distributions, as described in Section 4.2. Only the results for normal distributions are shown here, as the graphs for log-normal and Pareto yielded similar ranges of results, with the Pareto graphs making sharper turns at the peak. From these graphs, we can clearly see that the effectiveness is very sensitive to the choice of baseline, with a large range of effectiveness measures.

Table 2: Range of effectiveness measures over different baseline values, for ten randomly chosen pairs of normal distributions X and Y. Each distribution is presented in the first two columns as (μ, sd), where μ is the mean and sd is the standard deviation. For the third and fourth columns, the numbers in parentheses are the baseline values corresponding to the effectiveness measures; percentage values that round up to 100 in two decimal places are shown as ∼100. The last column shows how far the optimal baseline is from the mean of X, in terms of the number of standard deviations of X.

Distribution of Good Commit (X)	**Distribution of Bad Commit (Y)**	**Minimum Effectiveness**	**Maximum Effectiveness**	**Standard Deviations Away**
(8627, 931)	(8743, 336)	1.43 (8737)	2.30 (8627)	0
(1846, 347)	(3525, 120)	13.50 (1846)	99.90 (3066)	3.52
(3884, 772)	(9955, 328)	13.50 (3884)	∼100 (8114)	5.48
(3765, 309)	(5148, 707)	12.14 (3765)	58.78 (4325)	1.81
(4352, 886)	(7687, 159)	13.50 (4352)	99.63 (7112)	3.12
(4523, 896)	(8050, 342)	13.50 (4523)	98.66 (7003)	2.77
(1835, 195)	(9477, 153)	13.50 (1835)	∼100 (3415)	8.10
(3194, 685)	(9855, 491)	13.50 (3194)	∼100 (7064)	5.65
(1615, 533)	(9774, 993)	13.50 (1615)	∼100 (4515)	5.44
(4656, 918)	(6369, 181)	13.50 (4656)	72.57 (5966)	1.43

Table 3: Comparing the effectiveness at the midpoint with the maximum effectiveness, for each corresponding distribution type. Each difference is relative to the effectiveness at the midpoint.

	Normal	**Log-Normal**	**Pareto**
Avg. Percentage Point Difference Compared to Midpoint Effectiveness	2.07	2.23	5.85
# of Pairs with At Most 5 Percentage Point Difference	867	867	578
# of Pairs with Greater Than 5 Percentage Point Difference	133	133	422
# of Pairs with Greater Than 10 Percentage Point Difference	65	67	187
# of Pairs with Greater Than 20 Percentage Point Difference	9	3	54

This behaviour is made more evident by Table 2, which shows the exact range of effectiveness measures for each of the ten distribution pairs. In 9 out of the 10 pairs listed in this table, the effectiveness ranges from a value below 14% to a value above 58% (and above 98% in 7 out of the 10 scenarios).

Looking more closely at the graphs in Figure 4, we can observe that the effectiveness monotonically increases in value until it reaches a peak; thereafter, the effectiveness plateaus in some graphs, and then monotonically decreases in value. When considering all 1000 distribution pairs, the minimum effectiveness occurs at one of the endpoint mean values in all of the pairs (this applies to all three distribution types). Intuitively, this result makes sense – for example, most measurements of the bisection metric when testing a pre-regression commit will be very close to the mean of X, and choosing a baseline that is too close to that mean increases the chances of the measurement going *just above* the chosen baseline, thereby causing it to be mislabelled as a post-regression commit.

While the above result is expected, a more interesting question to ask is whether the midpoint of the means of distributions X and Y constitutes a reasonable estimate for the optimal baseline value. Table 3 shows the results. On average,

Table 4: Comparing the maximum effectiveness when assuming a normal distribution with the maximum effectiveness when assuming heavy-tailed distributions. Each difference is relative to the maximum effectiveness when assuming a normal distribution.

	Log-Normal	**Pareto**
Avg. Percentage Point Difference Compared to Normal	0.97	9.48
Maximum Percentage Point Difference Compared to Normal	12.06	50.14
# of Pairs with At Most 5 Percentage Point Difference	978	616
# of Pairs with Greater Than 5 Percentage Point Difference	22	384
# of Pairs with Greater Than 10 Percentage Point Difference	2	319
# of Pairs with Greater Than 20 Percentage Point Difference	0	237

Table 5: Comparing the effectiveness values when using the optimal baseline of the normal distribution case as the baseline for each distribution type. Each difference is relative to the maximum effectiveness when assuming a normal distribution.

	Log-Normal	**Pareto**
Avg. Percentage Point Difference Compared to Normal	1.03	7.29
Maximum Percentage Point Difference Compared to Normal	11.27	49.03
# of Pairs with At Most 5 Percentage Point Difference	973	628
# of Pairs with Greater Than 5 Percentage Point Difference	27	372
# of Pairs with Greater Than 10 Percentage Point Difference	2	216
# of Pairs with Greater Than 20 Percentage Point Difference	0	137

the effectiveness at the midpoint comes within around 5 percentage points of the maximum effectiveness for all three distribution types, with a Mann-Whitney U Test p-value very close to 0% due to the large sample size. The small average percentage point difference provides some evidence that the midpoint value can be used as a rough estimate of the optimal baseline. However, many deviations are still observed, with 13.3% deviating by more than 5 percentage points for normal distributions, 13.3% for log-normal distributions, and 42.2% for Pareto distributions. There are also several scenarios where the difference exceeds 10 percentage points, or even 20 percentage points, especially for Pareto.

***Finding 1**: The choice of baseline has a significant impact on the effectiveness of a bisection. The midpoint between the means can be used as a rough estimator of the optimal baseline, but it can be inaccurate especially for some heavy-tailed distributions*

Lastly, Table 4 compares the maximum effectiveness at the heavy-tailed distributions (i.e., log-normal and Pareto) with the maximum effectiveness at normal distributions. As seen in this table, the effectiveness for the log-normal case is very similar to the effectiveness for the normal case, differing by only around 1 percentage point on average. On the other hand, the effectiveness for the Pareto

Table 6: Number of bug reports that belong to each classification on the information provided, as defined in Section 4.2 (RQ2)

Application	FF	HF	FH	HH	FN	NF	HN	NH	NN	Total
ansible	1	-	-	-	9	-	-	-	-	10
cockroach	3	1	-	-	12	1	4	1	4	26
elasticsearch	2	-	-	-	13	-	2	-	3	20
flutter	-	-	-	-	13	-	12	-	5	30
kubernetes	-	-	1	1	15	-	8	-	5	30
moby	-	-	-	-	7	-	1	-	-	8
nixpkgs	-	-	-	-	3	-	3	-	-	6
node	5	-	1	-	19	-	5	-	-	30
origin	-	-	-	-	1	-	3	-	6	10
roslyn	-	-	-	-	1	-	2	-	4	7
runtime	15	1	-	-	10	-	4	-	-	30
rust	3	-	-	-	22	-	3	-	2	30
servo	-	-	-	-	-	-	5	-	3	8
tensorflow	1	-	-	-	15	-	5	-	-	21
tgstation	-	-	-	-	-	1	4	-	10	15
vscode	2	-	1	-	9	-	8	-	4	24
wp-calypso	-	-	-	-	-	-	3	-	2	5
Overall	**32**	**2**	**3**	**1**	**149**	**2**	**72**	**1**	**48**	**310**

case differs considerably more on average (i.e., by 9.48 percentage points), and sometimes differs by more than 50 percentage points. Nonetheless, most of the Pareto results are very close to the results from the normal case, with 61.6% coming within 5 percentage points and 68.1% coming within 10 percentage points. Table 5 shows a similar comparison, but this time, we examine the effectiveness for each distribution type with the baseline fixed to the optimal baseline of the normal case. The results are very similar to those seen in Table 4, except with considerably fewer Pareto results exceeding the normal results by 10 percentage points. Taken together, these numbers provide some evidence that setting X and Y as normal distributions may be a reasonable simplification to make when assessing the effectiveness, especially in the absence of raw performance data.

***Finding 2**: When assessing the effectiveness of a bisection and computing an optimal baseline, estimating the pre-regression and post-regression distributions as normal may be a suitable simplification, particularly if the real distribution is log-normal or Pareto. However, some differences are to be expected, especially if the true distribution is closer to Pareto*

5.2 Baselines in Practice

Based on Finding 1, it is important to choose a suitable baseline that will yield high effectiveness. In order to make this assessment, the developer – or whoever is investigating the performance regression – needs to know the full commit range, as well as the distribution parameters (both pre-regression and post-regression), as required by the effectiveness measure calculation. Here, we go over the results on how much of this information is provided to real-world developers when performance regression bug reports are filed.

Table 6 shows the classification of each bug report, based on the classification scheme defined in Section 4.2 (RQ2). As the table shows, only 32 out of 310 bug

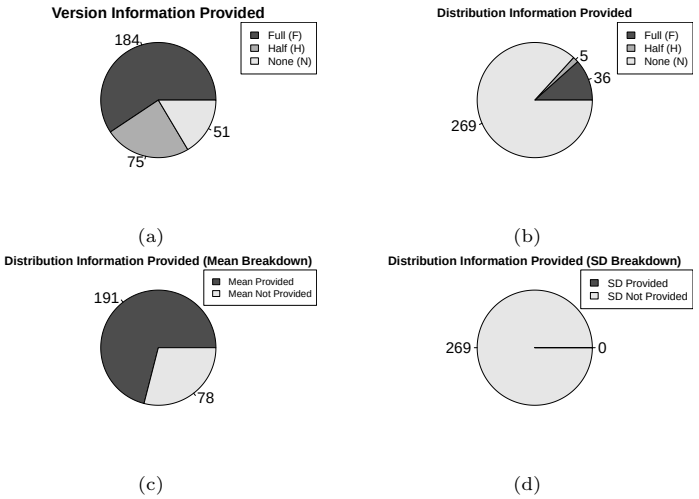


Fig. 5: Breakdown of the information provided: (a) Breakdown of version information provided; (b) Breakdown of complete (i.e., both mean and standard deviation) distribution information provided; (c) Breakdown of mean provided, among all bug reports labelled as "None" for complete distribution information; (d) Breakdown of standard deviation (SD) provided, among all bug reports labelled as "None" for complete distribution information.

reports belong to the **FF** category, which means that only around 10% of the bug reports analyzed provide the full version and distribution information needed to compute an optimal baseline. Looking more closely at the bug reports demystifies this result a bit, as in most cases, the person reporting is an end user who is not running thorough performance regression tests, as one would expect from a tester or a developer. As a matter of fact, the only application where reporters provided full distribution information in the majority of bug reports is `runtime`, where two qualities immediately stand out: (1) The `runtime` developers use a reporting system that automatically sends bug reports to an external repository, which a human can then validate and thereafter report as a performance regression in the main repository; and (2) the majority of performance regression bug reports sent to `runtime` are sent by developers themselves, and are based on the reports generated by the reporting system.

***Finding 3**: Performance regression bug reports often do not contain sufficient information to compute an optimal baseline*

Going back to Table 6, we can likewise see that the classification that takes up the largest percentage is **FN** (i.e., full version information provided, but no complete distribution information provided). This classification also takes up the largest percentage in 11 out of the 17 applications. These observations seem to indicate that performance regression bug reports are more likely to include full version information than they are to include full distribution information. To corroborate this claim, Figure 5 shows a breakdown of the classifications, with Figure 5a show-

ing the version classifications, and Figure 5b showing the distribution classifications. From these charts, it is clear that while the majority of reporters provide full version information (around 59%), an even more overwhelming majority (around 86%) provides no complete distribution information at all. Furthermore, of the 269 bug reports that provide no complete distribution information (i.e., those labelled as "None"), around 71% provide the mean for at least one commit; however, 100% of these 269 bug reports provide no information at all on the standard deviation. Once again, this observation meshes with the fact that most of the bug reports are reported by end users who are more motivated to send out a bug report quickly than to make thorough performance measurements.

Finding 4*: Most performance regression bug reports provide full version information; however, most of these bug reports do* not *provide complete distribution information at all, primarily due to missing standard deviations*

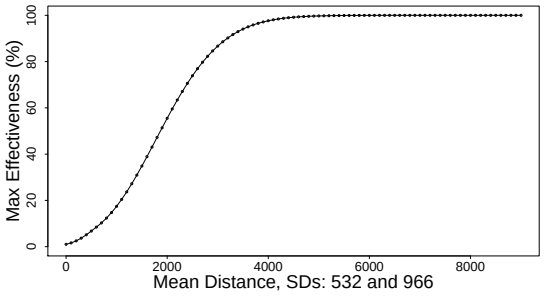
In the discussion section, some suggestions on ways to help developers fill the information gap are presented, as observed in the above results.

5.3 Impact of Distributions

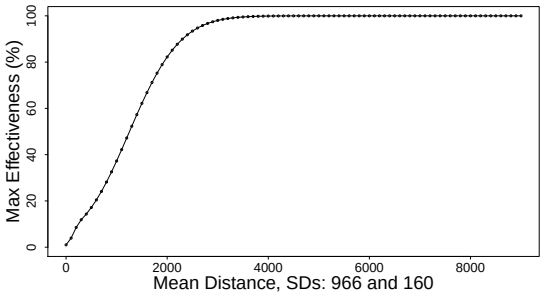
We start our analysis of the distributions by conducting the OAT sensitivity analysis on the means. Note that it suffices to conduct the analysis on the *distance* between the means, as the CDF functions (i.e., in Equations 2 and 4) evaluate to the same probability for distributions of the same shape, for all inputs with corresponding standardized scores (i.e., z-scores). Figure 6 shows the maximum effectiveness for a range of distance values between the means; while the results for 1000 randomly chosen pairs of standard deviations were generated, only two graphs are shown based on a sample of ten results, as the remaining graphs are very similar to Figure 6a, with the same sigmoid shape. The only exception among the pairs that were graphed is Figure 6b, which has two different inflection points – one between distance 0 and distance 300, and the other between distance 300 and distance 9000. Further, only the graphs for the normal distribution case are shown, as the log-normal and Pareto graphs were found to be very similar. Nonetheless, in all scenarios, the graph is monotonically increasing, which suggests that the maximum effectiveness increases as the distance between the means of X and Y increases.

Next, we vary the standard deviation values as described in Section 4.2. The results are shown in Figure 7; note that in these graphs, the base standard deviation value used for the distribution that is *not* varied in Figures 7a and 7b is 100, but similar results are observed for other base values. In general, regardless of the type of variation method, it is evident from these graphs that the maximum effectiveness decreases as the standard deviation increases. Note that in Figure 7, the range of effectiveness values is larger when the standard deviation of distribution X is varied, than when the standard deviation of distribution Y is varied. However, this pattern seems to be specific to normal and Pareto distributions, as the log-normal results are more senstive to the value of Y.

The above results are consistent with earlier findings on a noisy variant of binary search, in which the oracle that decides whether to take the left or right



(a)



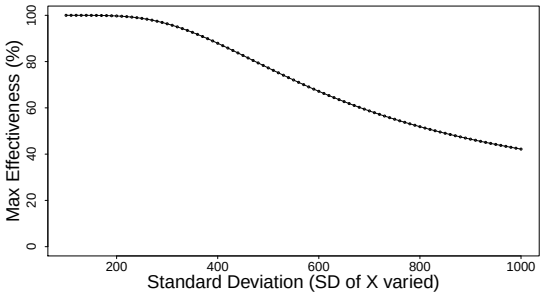
(b)

Fig. 6: Impact of the mean on the maximum effectiveness. The horizontal axis represents the distance between the means, while the vertical axis represents the maximum effectiveness attained. These results are computed using normal distributions, but the log-normal and Pareto results are very similar. Results were computed for 1000 different pairs of standard deviations, and graphs were generated for the first 10 pairs; the ones not shown here have the same sigmoid shape as the first graph.

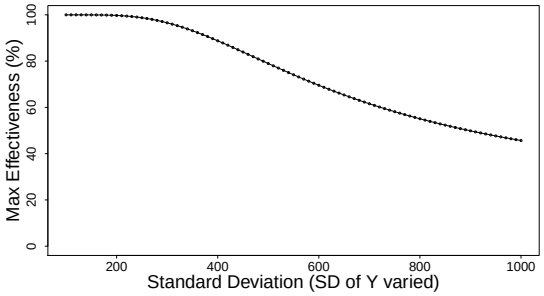
side of the array is incorrect with a known and fixed probability. For example, Pelc (1989) theoretically demonstrated that in the discrete bounded version of this noisy binary search – which is the closest analogue to performance regression bisection – an error probability less than 50% is needed in order to guarantee that the correct target element is found. Our results above can be seen as a corollary to this, as intuitively, smaller mean gaps and larger standard deviations would increase the error probability.

Finding 5*: In line with prior findings, the maximum effectiveness increases with the mean, and decreases as the standard deviation of any of the distributions increases*

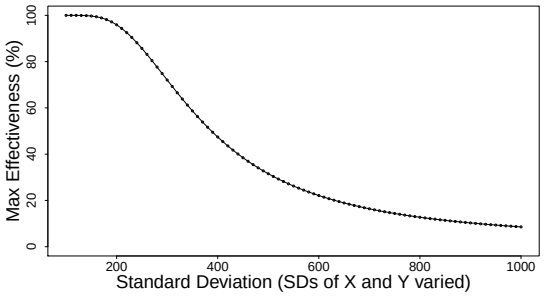
Lastly, Figure 8 shows a scatter plot of the maximum effectiveness versus the overlapping coefficient, which is a measure of the amount of overlap between distributions X and Y; this graph only shows the normal distribution results, but similar results are observed for log-normal and Pareto. In this case, the Pearson correlation coefficient is -0.4799, with a p-value less than 0.01 ($p = 6.44 \times 10^{-13}$), indicating a significant negative correlation. While the correlation value itself is



(a)



(b)



(c)

Fig. 7: Impact of the standard deviation on the maximum effectiveness: (a) Only the standard deviation of X is varied; (b) Only the standard deviation of Y is varied; (c) The standard deviations of X and Y are varied simultaneously. In all three graphs, the horizontal axes represent the standard deviation, and the vertical axes represent the maximum effectiveness. Note that for (a) and (b), the base value of the standard deviation that is *not* varied is 100.

quite weak when considering the data holistically, a couple of important observations can be made.

– The *only* distribution pairs that attained a maximum effectiveness score greater than 80% are those that have less than 30% overlap;

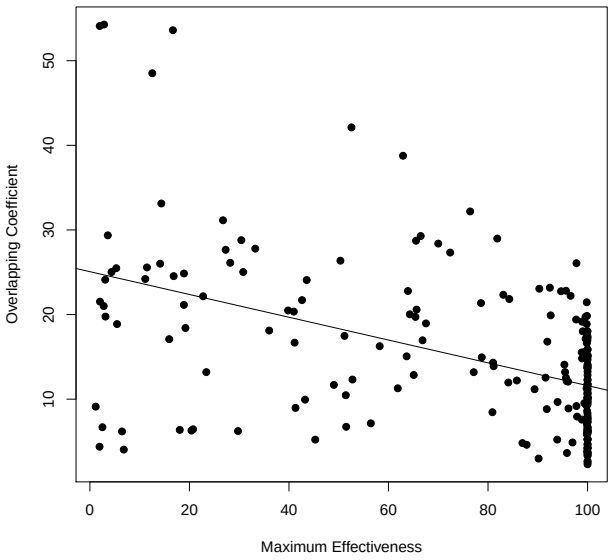


Fig. 8: Scatter plot of the maximum effectiveness versus the overlapping coefficient, for 200 randomly chosen pairs of distributions.

– Distribution pairs that have an overlap greater than 40% have low maximum effectiveness, with almost all of these having a maximum effectiveness of less than 20%

Thus, Figure 8 suggests that even though the overlapping coefficient is not an exact predictor of the maximum effectiveness even for a fixed commit range length value, it can nonetheless be used as a rough indicator. For example, in this particular setup where the commit range length is 100, the two observations stated above strongly indicate that distribution pairs with greater than 40% overlap is almost guaranteed to yield low maximum effectiveness, and that the distribution pairs should have less than 30% overlap in order for the bisection to even have a strong chance of yielding a maximum effectiveness that exceeds 80%.

***Finding 6**: The overlapping coefficient has a significant negative correlation with the maximum effectiveness, and can be used as a rough predictor of the effectiveness measure*

5.4 Distributions in Practice

Having analyzed the impact of the distributions on the effectiveness, we will now look at distribution pairs from real-world performance regressions. In total, full distribution information is provided in 36 bug reports, and for each one, the overlapping coefficient was computed. The results are shown in Figure 9. From this

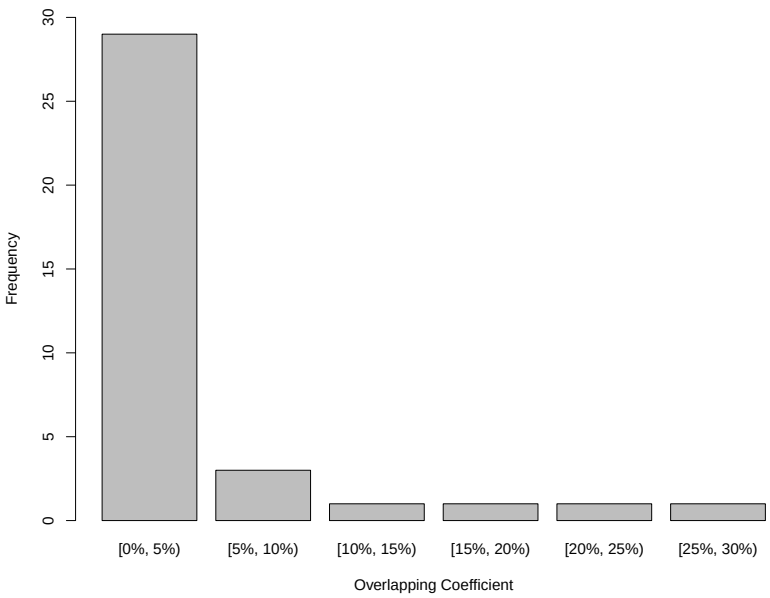


Fig. 9: Number of bug reports with a particular overlapping coefficient, among those where full distribution information is provided.

figure, we can see that the vast majority of distribution pairs overlap in the 0-5% range (29 out of 36). In addition, all 36 distribution pairs overlap by less than 30%; as per the results from Section 5.3, this means that all of these distribution pairs have a much greater chance of yielding a high maximum effectiveness measure.

***Finding 7**: Distribution pairs in reported performance regressions tend to have small overlap, making them more amenable to bisection*

Care must be exercised when interpreting this particular finding, with a more thorough discussion presented in Section 6.

5.5 Impact of Commit Range Length

Figure 10 shows graphs of the maximum effectiveness per commit range length value for four randomly chosen distributions; the remainder of the distributions that are not shown are very similar to the ones shown in this figure. Looking at these graphs, two features can be readily observed. First, the maximum effectiveness decreases as the commit range length increases. This behaviour makes sense intuitively, as longer commit range lengths imply a larger search space, thereby making it more difficult to find the bug-introducing commit. It also meshes with the intuition behind prior research on commit reduction (Najafi et al, 2019; An and Yoo, 2021; Saha and Gligoric, 2017), although these prior works were done in the

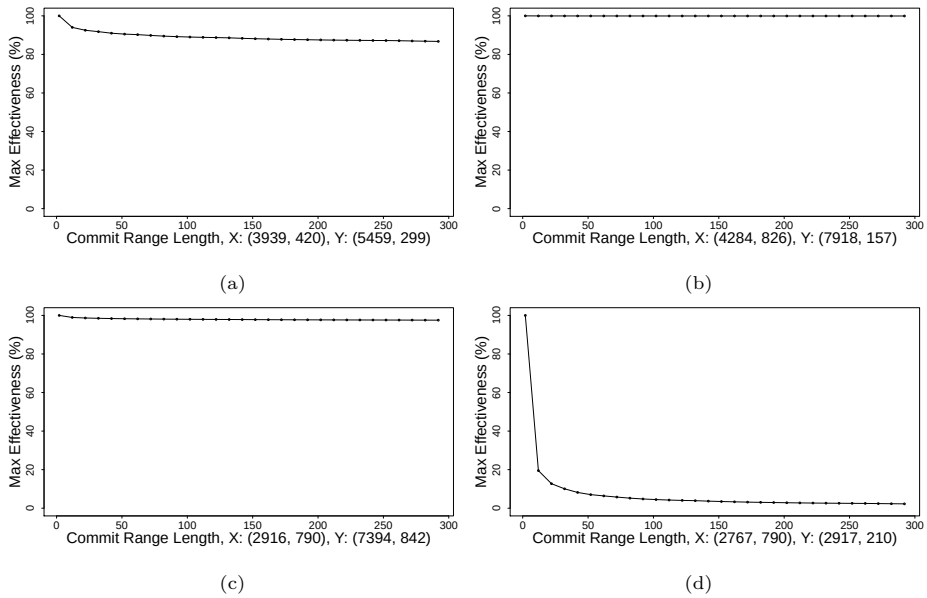


Fig. 10: Impact of the commit range length on the maximum effectiveness, for four randomly chosen distribution pairs. The horizontal axis represents the commit range length (ranging from 2 to 300), while the vertical axis represents the maximum effectiveness.

context of functional debugging, where bisection runtime was the main motivator for reducing the commits, as opposed to effectiveness. Second, the rate of decrease in the maximum effectiveness as the commit range length increases gets slower with longer lengths, which aligns with the logarithmic nature of binary search.

From the results, it is also evident that the decrease rate of the maximum effectiveness depends on the other inputs, notably the distribution pair. For instance, the distribution pair for Figure 10d is $X : (2767, 790)$ and $Y : (2917, 210)$, which consists of two means that are very close to each other and an overlapping coefficient of 44.34%; this particular feature translated to a larger overall rate of decrease for the maximum effectiveness, as the graph makes apparent.

***Finding 8**: The maximum effectiveness decreases as the commit range length increases, at a rate that gets slower with longer lengths*

5.6 Commit Range Lengths in Practice

Table 7 contains information about the number of bug reports where a particular type of commit range is provided ("Count") and the average length of the commit range for each of these types ("Avg. Length"). Looking at the "Count" columns, we can see that among those bug reports where full version information is provided, the majority of these reporters provided the version number (106 out of 184). After

Table 7: Type of version information provided among those where full version information is provided (see Section 4.2), and the average length of the commit range for each type. In some cases, the length can no longer be inferred, for reasons outlined in Section 4.2; those for which none of the lengths can be inferred are marked with a dash.

	Commit Range Type									
	Hash		Version		Test Run Number		Mix		Overall	
Application	**Count**	**Avg. Length**	**Count**	**Avg. Length**	**Count**	**Avg. Length**	**Count**	**Avg. Length**	**Count**	**Avg. Length**
ansible	0	-	10	3097.50	0	-	0	-	10	3097.50
cockroach	7	254.14	7	4151.29	1	-	0	-	15	2202.71
elasticsearch	5	7	10	3007	0	-	0	-	15	2007
flutter	12	2.22	0	-	0	-	1	-	13	2.22
kubernetes	6	80	2	1382	8	-	0	-	16	514
moby	0	-	7	2099.14	0	-	0	-	7	2099.14
nixpkgs	1	300	2	12642	0	-	0	-	3	6471
node	2	7	23	3296.83	0	-	0	-	25	3033.64
origin	0	-	1	3439	0	-	0	-	1	3439
roslyn	0	-	1	3154	0	-	0	-	1	3154
runtime	13	88.54	12	144	0	-	0	-	25	92.5
rust	15	120.20	6	2161.60	3	-	1	2350	25	712.43
servo	0	-	0	-	0	-	0	-	0	-
tensorflow	0	-	14	3782.14	0	-	2	1152	16	3453.38
tgstation	0	-	0	-	0	-	0	-	0	-
vscode	1	621	11	1767.50	0	-	0	-	12	1663.27
wp-calypso	0	-	0	-	0	-	0	-	0	-
Total	*62*	*106.02*	*106*	*3089.14*	*12*	-	*4*	*1551.33*	*184*	*1940.12*

this, the second-most frequently provided type of commit range is the commit hash value, with 62. The remaining bug reports either had the test run number provided, or some mix of the three types.

Furthermore, not including `servo`, `tgstation`, and `wp-calypso`, which had no bug reports that had full version information, the bug reports with version numbers provided comprise the majority in 9 applications, while those with hashes provided comprise the majority in only 4 applications (one application – `cockroach` – has 7 of each of these two types). Additional analysis of the four applications for which the majority had hashes provided reveals the following:

- For `flutter`, most of the bug reports link to a publicly available test result querying system that contains commit hashes;
- For `runtime`, a large number of bug reports are based on an automatically-generated report, where the reporting system includes hash information;
- For `kubernetes` and `rust`, reporters either simply provided the hashes directly, or provided information that contain hash information (e.g., link to a GitHub diff page, test logs, etc.)

***Finding 9**: In cases where full version information is provided, most reporters of performance regressions provide the version number for the commit range*

Now, looking at the "Avg. Length" columns, we can readily observe that the average commit range length when hashes are provided is around 106, while the average commit range length when version numbers are provided is aroud 3089 (the average length when these are mixed is around 1551, although this number is only based on three bug reports). This pattern also holds on a per-application basis, where the average length of the version numbers exceeds the average length of the hashes in all applicable scenarios. It is therefore clear from this result that the commit range length tends to be much higher when version numbers are provided

than when commit hashes are provided. Intuitively, this result can be understood in light of the fact that version numbers are conceptually less granular than commit hashes. It also strongly implies that most of the commit hashes provided by the reporters in the subject applications are not simply based on the version numbers themselves, but are based on commits that occur between version numbers.

***Finding 10**: The commit range length tends to be much shorter when hashes are provided compared to when version numbers are provided*

5.7 Impact of Transition Index

As mentioned in Section 4.2, this analysis on the transition index will focus on its impact, not on the average effectiveness measure, but on the "per bisection output" effectiveness measure – i.e., the value of $\epsilon_{x,c}$ as c is varied. Figure 11 shows the results for this analysis, from a sample of ten distributions out of the 1000 analyzed. There are a few interesting observations that can be made. First, the graphs are fractal-like, with each one having an almost symmetrical shape. Second, and more importantly, these graphs show that the effectiveness can actually vary significantly for different transition indices; for instance, in Figure 11e, the effectiveness is over 90% for several transition indices, but goes as low as 30.29% when the transition index is 51. At first glance, this result may seem counter-intuitive – i.e., it seems odd that bisection would have a high probability of success given that it outputs one particular suspected bug-introducing commit, but has a low probability of success given it outputs another. However, the effectiveness measure computation can provide some insight on why we are observing this result. For one, some transition indices require more recursive iterations than others; as an example, with 100 commits, a transition index of 2 requires only 6 iterations, while a transition index of 3 requires 7 iterations. From this standpoint, the variation of the effectiveness makes more sense intuitively, as more recursive iterations implies more tests to run, which in turn implies more room for error. In addition, for the same reason just stated, certain output probabilities can also be larger than others, and since the effectiveness is inversely proportional to the output probability (Equation 1), this can cause the effectiveness measure to lower.[3]

***Finding 11**: The effectiveness of a bisection varies with different transition indices*

5.8 Transition Indices in Practice

The third to fifth columns of Table 8 show the minimum and maximum "per bisection output" effectiveness $\epsilon_{x,c}$, as well as the difference between these two values, for all **FF** bug reports where the commit range length is known. What immediately stands out is that there are 21 bug reports whose minimum and maximum effectiveness round up to 100 in two decimal places. Of the remaining nine bug reports, eight have a minimum and maximum that differ by more than one

[3] In fact, this turns out to be the main reason the effectiveness measure tends to be low when the transition index is somewhere in the middle, for all three distribution types

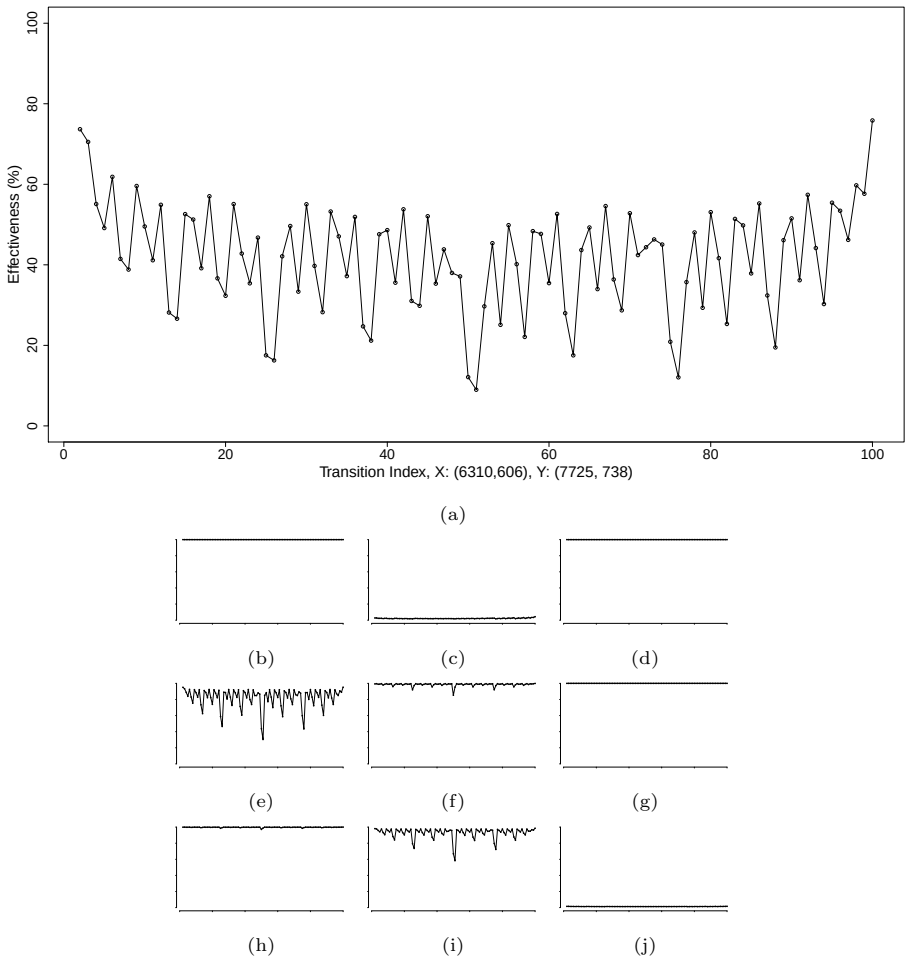


Fig. 11: Results of the OAT analysis for the impact of the transition index on the effectiveness, for a sample of ten randomly chosen pairs of normal distributions X and Y, out of 1000 pairs analyzed. The results for log-normal and Pareto exhibited similar behaviour. The horizontal axis refers to the transition index, and the vertical axis refers to the "per bisection output" effectiveness measure. The axis labels for all graphs are similar to those in (a).

percentage point, with most of these differing by more than 20 percentage points; the largest difference is seen in ELASTICSEARCH-8, with a 94.54 percentage point difference.

***Finding 12**: Reported performance regressions often have low sensitivity to the transition index, but a significant number are still very sensitive to this parameter*

Table 8: Bug reports with full information provided (i.e., those classified as **FF**), and whose commit range length can still be inferred. This table shows the minimum and maximum "per bisection output" effectiveness (i.e., $\epsilon_{x,c}$), as well as the difference in percentage points between the minimum and the maximum. It also shows the average effectiveness $\epsilon_{x,\text{avg}}$. Percentage values that round up to 100 in two decimal places are shown as ∼100 (for RUST-6, the average effectiveness is exactly 100, as there are only two commits).

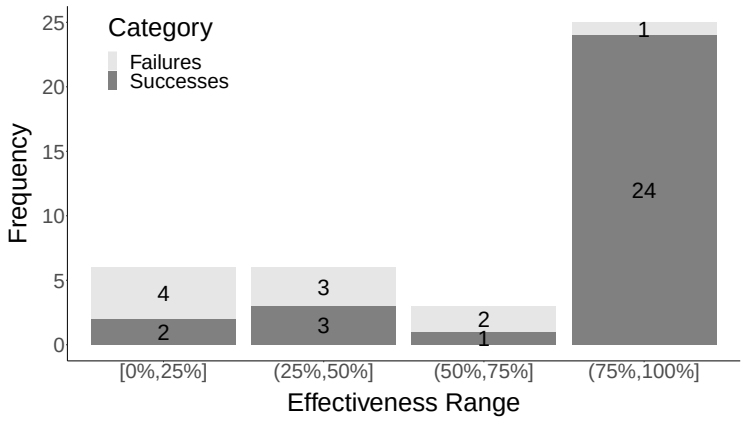
Application	Bug Report ID	Min $\epsilon_{x,c}$	Max $\epsilon_{x,c}$	Diff in Percentage Points	Avg. Effectiveness ($\epsilon_{x,\text{avg}}$)
ansible	ANSIBLE-1	∼100	∼100	∼0	∼100
cockroach	COCKROACH-1	99.85	99.87	0.02	99.86
	COCKROACH-10	9.63	88.85	79.22	34.14
	COCKROACH-23	∼100	∼100	1.90e-12	∼100
elasticsearch	ELASTICSEARCH-2	33.16	99.91	66.75	98.43
	ELASTICSEARCH-8	0.48	95.02	94.54	45.91
node	NODE-6	∼100	∼100	∼0	∼100
	NODE-12	∼100	∼100	6.77e-11	∼100
	NODE-13	∼100	∼100	∼0	∼100
	NODE-21	∼100	∼100	∼0	∼100
	NODE-23	∼100	∼100	∼0	∼100
runtime	RUNTIME-1	91.94	99.53	7.59	97.43
	RUNTIME-2	∼100	∼100	7.38e-7	∼100
	RUNTIME-6	∼100	∼100	3.60e-11	∼100
	RUNTIME-8	∼100	∼100	3.63e-5	∼100
	RUNTIME-10	∼100	∼100	∼0	∼100
	RUNTIME-11	∼100	∼100	∼0	∼100
	RUNTIME-12	∼100	∼100	9.95e-14	∼100
	RUNTIME-20	∼100	∼100	∼0	∼100
	RUNTIME-21	∼100	∼100	∼0	∼100
	RUNTIME-25	∼100	∼100	8.19e-8	∼100
	RUNTIME-26	∼100	∼100	1.57e-9	∼100
	RUNTIME-27	97.24	99.90	2.66	99.33
	RUNTIME-28	∼100	∼100	∼0	∼100
	RUNTIME-29	∼100	∼100	1.14e-13	∼100
rust	RUST-6	100	100	0	100
	RUST-15	70.80	99.79	28.99	95.86
	RUST-23	∼100	∼100	∼0	∼100
tensorflow	TENSORFLOW-18	0.39	75.47	75.09	11.13
vscode	VSCODE-2	0.70	40.27	39.56	4.06

5.9 Effectiveness in Practice

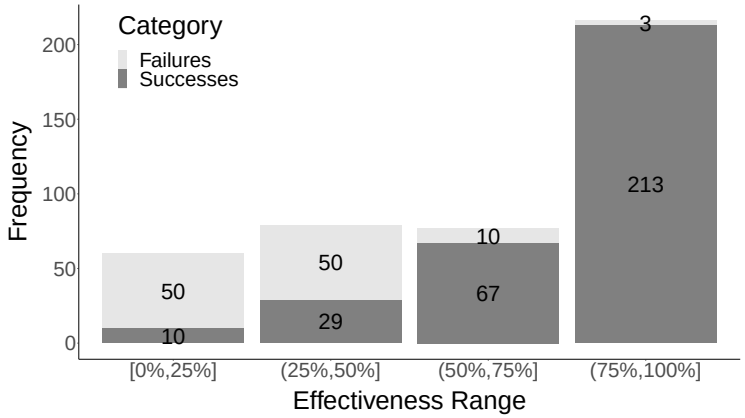
The last column of Table 8 shows the average effectiveness for all **FF** bug reports for which the commit range length can still be inferred. Of the 30 such bug reports, 26 exceed 95%. The main insight that can be gathered from this observation is that bisection is often a highly effective approach to localizing real-world performance regressions. It is, however, not a perfect approach, as 4 of the 30 bug reports fell well below 50% in terms of effectiveness; given how costly a bisection can be as discussed in Section 2.3, the fact that these performance regressions for which bisection is generally ineffective exist in real-world applications implies that a pre-assessment would still be useful prior to running the bisection.

***Finding 13**: Reported performance regressions often have high bisection effectiveness measures, although the effectiveness can still be quite low for some*

We also corroborate the above results with an analysis of real-world bisections, which also serves as an evaluation of the effectiveness measure itself. Figure 12a shows the breakdown of successful and failed bisections per effectiveness range, for



(a)



(b)

Fig. 12: Frequency of successful and failed bisections for different effectiveness ranges. The data is taken from the 40 most recent bisections conducted by the performance and reliability team at SAP, for which bisection results data are still available. The first graph shows the results when only considering these 40 bisections. The second graph shows the results for the forty bisections, with results extrapolated based on the path taken by four successful bisections for which we still have full data.

40 recent bisections conducted by our team for which full or partial results data are still available. There are two main observations that stand out from this graph. First, the vast majority (75%) of the bisections we conducted were successful, which aligns with expectations set by Finding 13. Second, all but one of the bisections with effectiveness over 75% succeeded, while the majority of bisections with effectiveness 25% or below failed; this result suggests at a rather coarse-grained level that the effectiveness measure can accurately predict the success or failure of a bisection.

In order to better appreciate the second observation above, we extrapolate the successful bisections for which we still have the full path to the correct commit, where a path consists of successive results at each bisection step. In particular, note that the 40 bisections above all used the optimal baseline; thus, to extrapolate the results, we take all other possible baselines (in increments of 10), compute the effectiveness at each of these baselines, and then check whether the bisection would *still* succeed had these alternative baselines been used, based on the full path to the correct commit. Any deviation from this full path will be counted as a failure. The results are shown in Figure 12b, and here, the accuracy of the effectiveness measure becomes more pronounced. More specifically, note how the success to failure ratio increases with higher effectiveness ranges.

***Finding 14**: The effectiveness measure is a reliable predictor of the success or failure of real-world bisections*

6 Discussion

We now discuss the implications of our findings. The discussion is organized based on action items that developers can take, based on the findings in Section 5.

Compute the effectiveness before and after the bisection. Finding 1 implies the usefulness of computing an optimal baseline prior to bisecting a performance regression, and Finding 14 provides empirical evidence that the effectiveness measure derived in Section 3 can be useful in this assessment. There are two notable decision points that developers need to take when implementing this computation. First is the type of distribution to use for X and Y. In general, the distribution type will be difficult to ascertain as the intermediate commits are, by definition, untested. The distribution types can be estimated if raw performance measurements are available for the two endpoints (i.e., the good commit and the bad commit). If these measurements are not available, Finding 2 suggests that normal distributions may be a suitable estimate, even in scenarios where the real distribution is not normal.

The second decision point is the manner in which optimal baseline search is conducted. Given n commits, and given that m possible baselines are considered (depending on the interval chosen), a linear search for the optimal baseline will take $\mathcal{O}(mn \log n)$. However, since the effectiveness increases towards a peak and then decreases (Figure 4), a binary search can be carried out, which will reduce the time complexity to $\mathcal{O}(n \log(m + n))$; this binary search approach is therefore the suggested way to compute the effectiveness measure.

Lastly, while most of the reported performance regressions have low sensitivity to the transition index (Finding 12), a significant number of these reported regressions are still quite sensitive to this contributing property. In such a case, the average effectiveness measure will not always be completely representative of the actual effectiveness based on the actual bisection output. Thus, in light of Finding 11, it would also help to compute the "per bisection output" effectiveness measure at the *end* of the bisection. Note also that the effectiveness measure defined in this paper is an *external* measure, in that it only takes into account input parameters to the bisection, and not what actually happens during the bisection

(e.g., How significant are the results of the individual tests? Are any of the results exhibiting multimodal behaviour?). An *internal* effectiveness measure that takes these execution characteristics into account may also be a helpful metric to measure at the end of a bisection, and devising such a metric can therefore be a useful problem for researchers to tackle. Having said all of this, it is of course still recommended that the average effectiveness be computed prior to running the bisection to help assess apriori the probability that it will provide the correct result; this is especially relevant given that bisection, while typically highly effective, can still have low effectiveness in practice (Finding 13).

Fill information gap through pre-emptive measures. Regarding the gap for the information needed to compute the effectiveness and ascertain the optimal baseline (Finding 4), two types of mitigative measures are suggested: (1) pre-emptive measures, whose goal is to increase the chance that reporters of performance regressions will provide the information needed, and (2) post-facto measures, which accept the information gap and attempt to fill that gap after the fact. Based on an analysis of the bug reports in the empirical study, the following pre-emptive measures are recommended:

- *Create a bug report template that is customized for performance regressions.* Many of the repositories already provide bug report templates; however, almost all of these are generic templates applicable to *any* regression, which explains why full version information is provided for a large number of bug reports (since this information is needed not just for performance regressions, but any type of regression), whereas complete distribution information is very scarcely provided. This suggestion is supported by data from the results, as the only repository that uses customized templates is `runtime`, which is also the only repository where full distribution information is provided in the majority of bug reports;
- *Provide an easy-to-use test command* that a potential reporter can run on a particular component of the application, on a specific commit. This test can be designed so that it runs a small number of measurements from which the mean and the standard deviation can be estimated. The reporter can then copy the results of this run and paste it into the bug report; the template suggested above can even be designed to ask for this information. Of course, it is infeasible to create a test for every component of an application, particularly for large applications, but it would be useful to create some for at least a representative set. This feature appears to be available to some degree in some of the applications that were analyzed in this study, including `cockroach` and `node` (both of which had several bug reports providing full distribution information).

Fill information gap through post-facto measures. In terms of post-facto measures, the following are recommended:

- *Run regular performance regression tests.* Doing this is good practice to begin with as it allows developers to regularly monitor performance changes in the product. It also has an additional benefit from the standpoint of bisection: if the reporter of a performance regression only specifies commit range information, developers can map that information with the test history to infer the distribution, thereby allowing the developer to compute the effectiveness. Ad-

ditionally, if the commit range provided is not granular enough (cf. Findings 9 and 10), test information can also reduce the search space;
- *Do a baseline run of each end of the commit range.* Doing this will allow the developers to fill the information gap on the distributions as the baseline runs will naturally provide that information. At the same time, it is also a required step in scenarios where the performance numbers are sensitive to certain hardware specifications; in particular, note that in most cases, the developers will be testing the application and trying to replicate the performance regression on a different set of hardware compared to what was used by the reporter.
- *Use rough estimators for effectiveness.* If means are provided but standard deviations are not, the midpoint may be sufficient in estimating the maximum effectiveness (Finding 1). On the other hand, if distribution information is provided, but the commit range is not, the overlapping coefficient exhibits a fairly significant negative correlation with the effectiveness, and can therefore also be used as an estimator (Finding 6).

Note that the pre-emptive and post-facto measures suggested above are not mutually exclusive, but instead complement each other. The pre-emptive measures can help developers get a quick sense of the effectiveness of the bisection prior to running any tests, while the post-facto measures will help developers get more concrete numbers applicable to the environment being used to run the bisection, and thus make a final assessment on the effectiveness.

Implement a mechanism to find the best bisection metric. The goal here is to find a bisection metric that maximizes the distance between the pre-regression and post-regression means, and minimizes the standard deviations (Finding 5). This recommendation is feasible, as regressions in one metric (e.g., response time) often correspond to regressions in other metrics (e.g., memory usage, CPU utilization, throughput, etc.), thereby providing the developer a choice on which bisection metric to use. Development teams that keep track of a large number of metrics (e.g., componentized performance logs) can consider automating this search process by computing the effectiveness for different metrics and finding the one yielding the highest effectiveness, although the time complexity considerations above should be taken into account.

As per Finding 6, it would also be useful to use the overlapping coefficient to pre-assess whether a particular bisection metric has a good chance of yielding high maximum effectiveness, and ultimately assess which bisection metric to use. This information will come in handy in a couple of scenarios: (1) in cases where the reporter provides complete distribution information, but incomplete version information (which, admittedly, does not happen too often based on the empirical study results, but nonetheless is still applicable to a few of the bug reports analyzed); and (2) in cases where the developer notices a regression between two test runs, and wants a quick way to predict what the effectiveness of a potential bisection might be before making the effort to gather all the other necessary inputs to compute the final effectiveness value.

On the surface, Finding 7 seems to render the advice just provided practically moot, as the finding states that distribution pairs tend to have small overlap in practice. However, this is not at all the case. In particular, recall that the distribution pairs considered in Finding 7 are all based on *reported* performance regressions; since reporters tend to be users of the application, who are more

likely to report human-detectable performance changes as opposed to very small performance changes, it makes sense that the distributions from these reported regressions are usually far apart. However, bisection is not only run to localize user-reported bugs, but also performance regressions observed in, for instance, automated regression tests, where the trend graphs can make even small regressions noticeable. To give an anecdotal example, our performance team at SAP typically investigates performance regressions in the order of milliseconds, oftentimes as low as 100 milliseconds, even for operations that take over a second to complete. In such a case, the overlap can vary significantly, and the pre-assessment described above will be useful.

Use variants of performance regression bisection. Depending on which requirements are not met, development teams can consider enhancing the version of bisection they are using, although note that there is often a tradeoff between accuracy and test execution time. Performance test systems that are prone to large distribution overlaps (cf. Findings 5 and 6) should consider using algorithms based on "noisy binary search", while systems that are prone to long commit ranges (cf. Finding 8) should consider implementing commit reduction techniques. Both of these areas are well-studied in prior research, and some existing techniques are described in Section 8.1.

7 Threats to Validity

We now discuss the internal, external, and construct threats to the validity of the present research.

7.1 Internal Threats

One limitation of the effectiveness measure is that it inherently assumes that everything before the bug-introducing commit has roughly the same distribution X and everything from the bug-introducing commit onwards also has roughly the same distribution Y. From experience, this assumption holds true in many practical situations, and it is precisely what makes it useful to monitor trend graphs to detect performance regressions. In addition, for large applications with a large number of components, many of the commits that happen around the bug-introducing commit touch unrelated components, and will usually have minimal impact on the component that regressed. Furthermore, the fact that around 75% of all bisections (and 89% of all bisections with at least 50% effectiveness) successfully found the bug-introducing commits for real-world performance regressions encountered by our team – as shown in Section 5.9 – provide some evidence that this assumption is warranted when applied to bisection. Having said these, there can of course still be situations where this assumption does not hold, as exhibited by prior research (Bezemer et al, 2014; Alcocer and Bergel, 2015; Sandoval Alcocer et al, 2016). As part of future research, it would be interesting to see how the effectiveness measure behaves given alternative permutations of the distributions at each commit.

Another internal threat to validity is that the effectiveness measure models the list of commits as an array, although in general, commits can take the form of any

directed acyclic graph (DAG). Nonetheless, this simplification is warranted, for a couple of reasons. First, even if the commits were modeled as a DAG, the computations are not expected to be drastically different, since DAGs can be topologically sorted as an array based on the midpoints taken during the bisection (i.e., take the midpoint chosen by the bisection in the DAG, then recursively sort the subgraph to its left and the subgraph to its right). Granted, the midpoints chosen in the DAG are not always guaranteed to have a balanced number of commits to its left and to its right in the topological sorting, but they usually do, as the midpoints are chosen such that the two sides are as balanced as possible (Couder, 2009). Second, this simplification allowed us to make more intuitive judgments about the sensitivity of the effectiveness to the contribution properties, as the data structure is simpler.

Finally, note that all of the bug reports were analyzed and categorized by one person (i.e., the author), which can also be seen as an internal threat. To minimize bias, two mitigative measures were put in place. First, categorization rules were clearly laid out, as outlined in Section 4, to ensure consistency. Second, the analysis was carried out over two separate passes, to correct any mistakes and to clear up any ambiguities that may have arisen in the first pass. Note that the present author has over five years of experience in software performance engineering, and has previously published empirical research that involved bug report categorization (Ocariza et al, 2017).

7.2 External Threats

As described in the experimental methodology, the number of applications and bug reports considered in this study is limited. While this is acknowledged as an external validity threat, it was necessary to make a tradeoff between the number of bug reports analyzed and the effort it takes to analyze them. The number of bug reports in this study (310) is reasonable, as it is large enough to make statistically significant claims, and small enough to allow for a reasonably thorough analysis of each bug report. Indeed, there is also precedence from prior research, including one conducted by the present author, where 317 bug reports were analyzed (Ocariza et al, 2013); and those conducted by Nistor et al (2013) and Selakovic and Pradel (2016), each of which analyzed fewer than 300 performance-related bug reports.

In the same vein as above, the sample sizes of the data collected to motivate the choice for the nominal values and to evaluate the accuracy of the effectiveness measure are relatively small. Unfortunately, our team regularly deletes old data in order to save costs, which consequently limited this sample size. While our team can anecdotally attest to the representativeness of the data presented if extrapolated towards a longer time period, it is nonetheless necessary to acknowledge this limitation as an external threat on the basis of the amount that actually is available and shown in the paper.

Limiting the analysis to three parametric distributions (normal, log-normal, and Pareto) can also be seen as an external validity threat. Narrowing down the scope in this manner is useful as it allows for more focused analyses of the results. In addition, the distributions that were chosen are reasonable to analyze, with the rationales provided in Section 2.4. Nonetheless, it would be interesting to see

how similar (or different) the results may be for other distribution types, including non-parametric ones.

7.3 Construct Threats

Lastly, in terms of construct validity threats, the OAT sensitivity analysis inherently does not capture interactions between the contributing properties. This limitation does not in itself invalidate the results, but it highlights the possibility that there are important insights that may have been missed, and could provide additional context to the findings in Section 5. Some of these interactions did make their way into our analysis, such as the impact of the distributions on the rate of decrease of the effectiveness when varying the commit range length (Section 5.5).

8 Related Work

Prior work related to the current research is now discussed. In addition to research already known to the current author, a literature search was conducted using keywords pertinent to the topic, discussed shortly. Each keyword is used as a search term in Google Scholar, and for each keyword, at least the first five pages of results are scanned for any related papers.

We divide related work according to three main topics, with the first two topics relating to the non-empirical aspects of the paper, and the last topic relating to its empirical aspects. First, we discuss prior research on bisection, gathered in part using the keywords "bisection debugging", "noisy binary search", and "git bisect". Second, we discuss work previously done on the more general problem of performance regression localization, gathered in part using the keywords "performance regression", "software performance regression localization", and "performance regression debugging". Third, we discuss work related to empirical studies conducted on performance issues, gathered in part using the keywords "software performance bug reports" and "software performance regression empirical study".

8.1 Bisection

To begin, we discuss prior work on bisection in both research and industry. We start by outlining work on bisection in the context of regression localization. Thereafter, we discuss more general techniques that perform binary search in the presence of noise.

There are many blog posts and unpublished work online that discuss the topic of bisection, including those that discuss it in the context of performance regressions (Couder, 2009; Murphy, 2018). However, these posts primarily discuss the functionality of bisection, as opposed to its effectiveness. A column that appears in ACM Queue (Neville-Neil, 2021) very briefly addresses certain concerns and limitations regarding the use of bisection, although its concern is not so much regarding effectiveness, but regarding the blind usage of bisection, and the possibility that developers will take its results at face value without attempting to understand *why* the regression was introduced.

In terms of tooling, version control systems such as Git (2009), Mercurial (2005), and Fossil (2006) natively support bisection, and some third-party software and plugins exist that take advantage of these existing technologies. Perhaps the most relevant to this present work is Phoronix Test Suite (Larabel, 2009a), which contains a module that uses `git bisect` to try to isolate performance regressions. Documentation is lacking for this feature, but based on other articles in their website (Larabel, 2009b), it appears to be testing the statistical significance of the results from each recursive iteration and making decisions based on the outcome of this significance test. If so, this significance test pertains more to an *internal* effectiveness measure similar to the one alluded to in Section 6, as opposed to an external effectiveness measure, which is what this paper focuses on, and is more relevant in pre-assessing whether to run a bisection in the first place.

Although a very popular approach in practice, research publications pertaining to bisection are quite scarce. The concept of bisection – in the context spoken of in this paper – was first introduced by Gross (1997). Twenty years later, Saha and Gligoric (2017) would propose an approach that reduces the amount of recursive iterations (i.e., what the authors call "bisection steps" in their paper) using test and commit selection. A technique that can be seen as a generalization of bisection is delta debugging (Zeller, 1999; Artho, 2011), which localizes at different levels of granularities, and not just commits. Unlike this present work, these papers do not analyze the theoretical and practical effectiveness of bisection – whether it be in its more specific form or in a generalized form – when applied to performance regressions.

Bisection is a specialized case of the binary search algorithm, and here, we focus on discussing prior work on a variant of this algorithm called "noisy binary search". Similar to performance regression bisection, the goal of noisy binary search is to find a target element using the regular binary search algorithm, but with each query having a certain probability of being incorrect. This is a well-studied domain (Nowak, 2009; Ben-Or and Hassidim, 2008; Dereniowski et al, 2021; Bittner et al, 2018; Epa et al, 2019; Rivest et al, 1980). Of particular note is the work done by Waeber et al (2013) that analyzes certain theoretical properties of this noisy variant; however, unlike performance regression bisection, the analysis is carried out for bisection over a continuous domain, and the focus is primarily on understanding the residual convergence of the algorithm. Many studies have been conducted on discretized variants of noisy binary search (Jedynak et al, 2012; Karp and Kleinberg, 2007; Tsiligkaridis, 2016), but these studies focus primarily on understanding theoretical bounds (e.g., bounds on the error probability at each step of the search) as opposed to providing an overall effectiveness measure. In addition, many of these studies assume a known per-iteration error probability, which does not apply to performance bisections where the intermediate probability distributions are not known in advance. Lastly, they also do not explore any practical considerations regarding the specific use-case of regression localization, which the present study does.

8.2 Performance Regression Localization

Next, we discuss approaches that have been previously proposed to simplify the process of localizing performance regressions. These include automated localization techniques, as well as techniques that act as aids in the localization process.

In prior work by our performance and reliability team (Ocariza and Zhao, 2021), we proposed a tool that automatically localizes performance regressions in client-side JavaScript by comparing execution call trees (i.e., timelines). Along the same lines, Bezemer et al (2015) proposed a tool for these timelines that outputs a visualization diff. Various other techniques have been proposed in research to automate this localization process, including general techniques (Malik et al, 2010; Heger et al, 2013; Nguyen et al, 2014; Luo et al, 2016; Shang et al, 2015; Sandoval Alcocer et al, 2016), as well as techniques applied to specific domains such as database systems and machine learning systems (Tizpaz-Niari et al, 2018; Jung et al, 2019; Tizpaz-Niari et al, 2020).

There have also been prior research on techniques to facilitate the process of localization, including Gregg (2016), whose article discusses the usefulness of performance timelines in this process. Similarly, Rogora et al (2020) proposed techniques to add probabilistic performance annotations to help debug detected performance issues. Additionally, Alcocer et al (2019) suggest a way to visualize performance variations along software versions.

Even though the goal of the above techniques is the same as that of bisection at a high level, they are not panaceas. In particular, they are either localized to work for specific types of performance regressions, or make some limiting assumptions about the code base (e.g., the existence of a repository of previous performance regression root causes). Bisection, in contrast, is a general-purpose approach applicable to any repository with a commit history; its wide usage, along with the costs associated with running it, makes bisection warrant its own study regarding its effectiveness, which this paper has demonstrated.

8.3 Empirical Study of Performance Issues

Finally, we discuss empirical studies that have been conducted on performance-related issues. Some of these prior works studied performance issues based on an analysis of bug reports, while others based their study on other components, such as developer commits.

Many researchers have previously conducted these empirical studies, including Selakovic and Pradel (2016), who analyzed the characteristics of fixed performance issues in popular JavaScript projects; Linares-Vásquez et al (2015), who studied performance bottlenecks in Android apps based on how they are detected and fixed; Chen et al (2019), who extracted performance bug patterns based on commits; Sánchez et al (2020), who proposed a taxonomy of performance issues based on real-world performance bugs; and Leitner and Bezemer (2017), who studied how performance testing is conducted on open source software. Han et al (2019) studied performance bug reports with the goal of analyzing their replicability from a resesarch standpoint; this study was a follow-up to another empirical study conducted by the same authors on highly configurable software systems (Han and Yu, 2016). Harkening back to the variance in performance measurements mentioned

earlier, Arif et al (2018) studied the discrepancy between performance test results. These works, however, do not specifically study performance regressions, which is the main fault model of interest in this current work.

Chen and Shang (2017) conducted an exploratory study of code changes that introduced performance regressions. This study focused exclusively on analyzing commits, as opposed to bug reports. In addition, it was more interested in understanding how performance regressions come about, as opposed to studying techniques to localize them post-facto.

Lastly, Nistor et al (2013) looked at performance bugs from different code bases, and analyzed how they are detected, reported, and fixed, in comparison to non-performance bugs. For instance, the authors found that performance bugs are generally more difficult to fix compared to non-performance bugs, and conclude the need for better tool support for the former. In a similar vein, Zhao et al (2020) studied how performance issues are caused and resolved. While their goal of understanding the characteristics of performance bugs is shared with this paper, this current work differs in many respects. First, the focus of this paper is on performance regressions, which entail different localization requirements compared to other types of performance issues. Further, the bug reports in this current study are analyzed within the context of bisection, which is outside of the scope of these other studies.

9 Conclusion

In the preceding sections, we discussed the need to understand the effectiveness of bisection when applied to performance regressions. After introducing a metric to quantify this effectiveness, an empirical study was conducted to help us understand the sensitivity of this effectiveness measure to its contributing properties, as well as the real-world characteristics of these contributing properties in reported performance regressions. The results presented in Section 5 and the discussion that follows it should help developers better assess the suitability of bisection for specific performance regressions, make more informed decisions about what baseline value and bisection metrics to use, and put measures in place to fill the information gap currently present in bug reports. The results should also help researchers better understand key properties of bisection for performance regressions, and thereby formulate solutions to improve or extend it. Future work includes improving the performance of the effectiveness measure computation, deriving internal effectiveness measures and studying their properties, and understanding how to make bisection work best when used with detection techniques (e.g., change-point detection).

Acknowledgements Many thanks go out to Jacques Buchholz and the rest of the performance and reliability team at SAP Vancouver for supporting this project. Special thanks to Parry Fung, Roger Zhao, Mahbubur Shihab, and the anonymous EMSE reviewers for their feedback, which have served to improve the quality of the paper.

Declarations

Conflict of interests Not Applicable

Competing Interests Not Applicable

References

Ahmed TM, Bezemer CP, Chen TH, Hassan AE, Shang W (2016) Studying the effectiveness of application performance management (apm) tools for detecting performance regressions for web applications: an experience report. In: Proceedings of the International Conference on Mining Software Repositories (MSR), ACM, pp 1–12

Akinshin A (2019) Pro. NET Benchmarking: The Art of Performance Measurement. Springer

Alcocer JPS, Bergel A (2015) Tracking down performance variation against source code evolution. ACM SIGPLAN Notices 51(2):129–139

Alcocer JPS, Beck F, Bergel A (2019) Performance evolution matrix: Visualizing performance variations along software versions. In: Proceedings of the Working Conference on Software Visualization (VISSOFT), IEEE, pp 1–11

An G, Yoo S (2021) Reducing the search space of bug inducing commits using failure coverage. In: Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM, pp 1459–1462

Arif MM, Shang W, Shihab E (2018) Empirical study on the discrepancy between performance testing results from virtual and physical environments. Empirical Software Engineering 23(3):1490–1518

Artho C (2011) Iterative delta debugging. International Journal on Software Tools for Technology Transfer (STTT) 13(3):223–246

Automattic (2021) Automattic WordPress Calypso. `https://www.github.com/Automattic/wp-calypso` (Accessed: July 20, 2021)

Ben-Or M, Hassidim A (2008) The bayesian learner is optimal for noisy binary search (and pretty good for quantum as well). In: Proceedings of the IEEE Symposium on Foundations of Computer Science, IEEE, pp 221–230

Bezemer C, Milon E, Zaidman A, Pouwelse J (2014) Detecting and analyzing I/O performance regressions. Journal of Software: Evolution and Process (JSEP) 26(12):1193–1212

Bezemer CP, Pouwelse J, Gregg B (2015) Understanding software performance regressions using differential flame graphs. In: Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, pp 535–539

Bittner DM, Sarwate AD, Wright RN (2018) Using noisy binary search for differentially private anomaly detection. In: Proceedings of the International Symposium on Cyber Security Cryptography and Machine Learning (CSCML), Springer, pp 20–37

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

Chen T, Guo Q, Temam O, Wu Y, Bao Y, Xu Z, Chen Y (2014) Statistical performance comparisons of computers. IEEE Transactions on Computers 64(5):1442–1455

Chen Y, Winter S, Suri N (2019) Inferring performance bug patterns from developer commits. In: Proceedings of the International Symposium on Software Reliability Engineering (ISSRE), IEEE Computer Society, pp 70–81

Cockroach Labs (2021) CockroachDB. https://www.github.com/cockroachdb/cockroach (Accessed: July 20, 2021)

Couder C (2009) Fighting regressions with git bisect. https://git-scm.com/docs/git-bisect-lk2009 (Accessed: August 9, 2021)

Crovella ME (2000) Performance evaluation with heavy tailed distributions. In: International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS), Springer, pp 1–9

Crovella ME, Taqqu MS, Bestavros A (1998) Heavy-tailed probability distributions in the world wide web. A Practical Guide to Heavy Tails: Statistical Techniques and Applications 1:3–26

Dahl R (2021) Node.js. https://www.github.com/nodejs/node (Accessed: July 20, 2021)

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

Dereniowski D, Łukasiewicz A, Uznański P (2021) An efficient noisy binary search in graphs via median approximation. In: Proceedings of the International Workshop on Combinatorial Algorithms, Springer, pp 265–281

Dynatrace (2018) Dynatrace. https://www.dynatrace.com/ (Accessed: January 8, 2018)

Elastic NV (2021) Elasticsearch. https://www.github.com/elastic/elasticsearch (Accessed: July 20, 2021)

Epa NS, Gan J, Wirth A (2019) Result-sensitive binary search with noisy information. In: Proceedings of the International Symposium on Algorithms and Computation (ISAAC), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik

Exadv1 (2021) Space Station 13. https://www.github.com/tgstation/tgstation (Accessed: July 20, 2021)

Fossil (2006) Fossil bisect command documentation. https://www.fossil-scm.org/fossil/help/bisect (Accessed: August 11, 2021)

Gaviar A (2019) GitHub's Top 100 Most Valuable Repositories Out of 96 Million. https://hackernoon.com/githubs-top-100-most-valuable-repositories-out-of-96-million-bb48caa9eb0b (Accessed: July 19, 2021)

Git (2009) Git Bisect Documentation. https://git-scm.com/docs/git-bisect (Accessed: August 11, 2021)

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

Google (2021a) Bisecting Performance Regressions. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/speed/bisects.md (Accessed: November 29, 2021)

Google (2021b) Google Flutter. https://www.github.com/flutter/flutter (Accessed: July 20, 2021)

Google (2021c) Kubernetes. https://www.github.com/kubernetes/kubernetes (Accessed: July 20, 2021)

Google (2021d) TensorFlow. https://www.github.com/tensorflow/tensorflow (Accessed: July 20, 2021)

Graham SL, Kessler PB, Mckusick MK (1982) Gprof: A call graph execution profiler. In: Proceedings of the SIGPLAN Symposium on Compiler Construction, ACM, pp 120–126

Gregg B (2016) The flame graph: This visualization of software execution is a new necessity for performance profiling and debugging. Queue 14(2):91–110

Gross T (1997) Bisection debugging. In: Proceedings of the International Workshop on Automatic Debugging (AADEBUG), Linkøping University Electronic Press, 001, pp 185–191

Han X, Yu T (2016) An empirical study on performance bugs for highly configurable software systems. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement, ACM/IEEE, pp 1–10

Han X, Carroll D, Yu T (2019) Reproducing performance bug reports in server applications: The researchers' experiences. Journal of Systems and Software 156:268–282

Harchol-Balter M (2013) Performance modeling and design of computer systems: queueing theory in action. Cambridge University Press

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

Inman HF, Bradley Jr EL (1989) The overlapping coefficient as a measure of agreement between probability distributions and point estimation of the overlap of two normal densities. Communications in Statistics-Theory and Methods 18(10):3851–3874

Jedynak B, Frazier PI, Sznitman R (2012) Twenty questions with noise: Bayes optimal policies for entropy loss. Journal of Applied Probability 49(1):114–136

Jung J, Hu H, Arulraj J, Kim T, Kang W (2019) Apollo: Automatic detection and diagnosis of performance regressions in database systems. Proceedings of the VLDB Endowment 13(1):57–70

Karp RM, Kleinberg R (2007) Noisy binary search and its applications. In: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, ACM, pp 881–890

Keenan JE (2019) Multisection: When Bisection Isn't Enough to Debug a Problem – The Perl Conference 2019. https://www.youtube.com/watch?v=05CwdTRt6AM (Accessed: November 18, 2021)

Larabel M (2009a) Autonomously finding performance regressions in the linux kernel. https://www.phoronix.com/scan.php?page=article&item=linux_perf_regressions&num=2 (Accessed: August 11, 2021)

Larabel M (2009b) Phoromatic tracker launches to monitor linux performance. https://www.phoronix.com/scan.php?page=article&item=phoromatic_tracker&num=2 (Accessed: August 11, 2021)

Leitner P, Bezemer CP (2017) An exploratory study of the state of practice of performance testing in java-based open source projects. In: Proceedings of the International Conference on Performance Engineering (ICPE), ACM, pp 373–384

Linares-Vásquez M, Vendome C, Luo Q, Poshyvanyk D (2015) How developers detect and fix performance bottlenecks in Android apps. In: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME),

IEEE Computer Society, pp 352–361

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

Malik H, Adams B, Hassan AE (2010) Pinpointing the subsystems responsible for the performance deviations in a load test. In: Proceedings of the International Symposium on Software Reliability Engineering (ISSRE), IEEE Computer Society, pp 201–210

Mercurial (2005) Mercurial bisect command documentation. `https://www.selenic.com/mercurial/hg.1.html` (Accessed: August 11, 2021)

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

Microsoft (2018) Startup Performance regressed. `https://github.com/microsoft/vscode/issues/42513` (Accessed: June 20, 2021)

Microsoft (2021a) Microsoft .NET CoreFX. `https://www.github.com/dotnet/runtime` (Accessed: July 20, 2021)

Microsoft (2021b) Microsoft .NET Roslyn. `https://www.github.com/dotnet/roslyn` (Accessed: July 20, 2021)

Microsoft (2021c) Microsoft Visual Studio Code. `https://www.github.com/microsoft/vscode` (Accessed: July 20, 2021)

Microsoft (2021) [Perf -6%] Regression in System.Text.Encodings.Web.Tests.Perf_Encoders. `https://github.com/dotnet/runtime/issues/48519` (Accessed: June 20, 2021)

Moby Project (2021) Moby. `https://www.github.com/moby/moby` (Accessed: July 20, 2021)

Mozilla Corporation (2021) Servo. `https://www.github.com/servo/servo` (Accessed: July 20, 2021)

Murphy W (2018) Investigating performance changes with git bisect. `https://willmurphyscode.net/2018/02/07/investigating-performance-changes-with-git-bisect/l` (Accessed: August 11, 2021)

Najafi A, Rigby PC, Shang W (2019) Bisecting commits and modeling commit risk during testing. In: Proceedings of Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM, pp 279–289

Neville-Neil GV (2021) Divide and conquer: The use and limits of bisection. Queue 19(3):37–39

Nguyen TH, Nagappan M, Hassan AE, Nasser M, Flora P (2014) An industrial case study of automatically identifying performance regression-causes. In: Proceedings of the International Working Conference on Mining Software Repositories (MSR), ACM, pp 232–241

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

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

NixOS (2021) NixOS Package Collection. https://www.github.com/NixOS/nixpkgs (Accessed: July 20, 2021)

Nowak R (2009) Noisy generalized binary search. In: Proceedings of Advances in Neural Iinformation Processing Systems, pp 1366–1374

Ocariza F (2020) Web Application Debugging – UBC Guest Lecture. https://www.youtube.com/watch?v=gNa247IaaGM (Accessed: June 20, 2021)

Ocariza F, Zhao B (2021) Localizing software performance regressions in web applications by comparing execution timelines. Software Testing, Verification and Reliability (STVR) 31(5):e1750

Ocariza F, Bajaj K, Pattabiraman K, Mesbah A (2013) An empirical study of client-side JavaScript bugs. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE Computer Society, pp 55–64

Ocariza FS, Bajaj K, Pattabiraman K, Mesbah A (2017) A study of causes and consequences of client-side javascript bugs. IEEE Transactions on Software Engineering 43(2):128–144

Olianas D, Leotta M, Ricca F, Biagiola M, Tonella P (2021) STILE: a tool for parallel execution of e2e web test scripts. In: Proceedings of the International Conference on Software Testing, Verification and Validation (ICST), IEEE Computer Society, pp 460–465

Pelc A (1989) Searching with known error probability. Theoretical Computer Science 63(2):185–202

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

Red Hat (2021a) Ansible. https://www.github.com/ansible/ansible (Accessed: July 20, 2021)

Red Hat (2021b) Red Hat OpenShift. https://www.github.com/openshift/origin (Accessed: July 20, 2021)

Rivest RL, Meyer AR, Kleitman DJ, Winklmann K, Spencer J (1980) Coping with errors in binary search procedures. Journal of Computer and System Sciences 20(3):396–404

Rogora D, Carzaniga A, Diwan A, Hauswirth M, Soulé R (2020) Analyzing system performance with probabilistic performance annotations. In: Proceedings of the European Conference on Computer Systems (EuroSys), pp 1–14

Saha R, Gligoric M (2017) Selective bisection debugging. In: Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE), Springer, pp 60–77

Sánchez AB, Delgado-Pérez P, Medina-Bulo I, Segura S (2020) Tandem: A taxonomy and a dataset of real-world performance bugs. IEEE Access 8:107,214–107,228

Sandoval Alcocer JP, Bergel A, Valente MT (2016) Learning from source code history to identify performance failures. In: Proceedings of the International Conference on Performance Engineering (ICPE), ACM, pp 37–48

Sasaki H, Su FH, Tanimoto T, Sethumadhavan S (2017) Why do programs have heavy tails? In: Proceedings of the International Symposium on Workload Characterization (IISWC), IEEE, pp 135–145

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

Shang W, Hassan AE, Nasser M, Flora P (2015) Automated detection of performance regressions using regression models on clustered performance counters. In: Proceedings of the International Conference on Performance Engineering (ICPE), ACM, pp 15–26

The Rust Foundation (2021) Rust. `https://www.github.com/rust-lang/rust` (Accessed: July 20, 2021)

Tizpaz-Niari S, Cerny P, Chang BYE, Trivedi A (2018) Differential performance debugging with discriminant regression trees. In: Proceedings of the AAAI Conference on Artificial Intelligence, AAAI

Tizpaz-Niari S, Černỳ P, Trivedi A (2020) Detecting and understanding real-world differential performance bugs in machine learning libraries. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), ACM, pp 189–199

Tsiligkaridis T (2016) Asynchronous decentralized algorithms for the noisy 20 questions problem. In: Proceedings of the International Symposium on Information Theory (ISIT), IEEE, pp 2699–2703

Waeber R, Frazier PI, Henderson SG (2013) Bisection search with noisy responses. SIAM Journal on Control and Optimization 51(3):2261–2279

Weitzman MS (1970) Measures of overlap of income distributions of white and Negro families in the United States, vol 3. US Bureau of the Census

YourKit (2018) YourKit. `https://www.yourkit.com/` (Accessed: July 2, 2018)

Zaman S, Adams B, Hassan AE (2012) A qualitative study on performance bugs. In: Proceedings of the IEEE Working Conference on Mining Software Repositories (MSR), IEEE Computer Society, pp 199–208

Zeller A (1999) Yesterday, my program worked. today, it does not. why? In: Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM, pp 253–266

Zhao Y, Xiao L, Wang X, Sun L, Chen B, Liu Y, Bondi AB (2020) How are performance issues caused and resolved?-an empirical study from a design perspective. In: Proceedings of the International Conference on Performance Engineering (ICPE), ACM, pp 181–192

Author Biography

Frolin S. Ocariza, Jr. received his BASc degree (with honours) from the University of Toronto in 2010, and his MASc and PhD degrees from the University of British Columbia (UBC) in 2012 and 2016, respectively. He currently works in SAP Vancouver with the performance and reliability (P&R) team. His main area of research is in software engineering, with emphasis on software performance engineering, web applications, code analysis, fault localization and repair, and fault detection. He received a nomination for the Best Paper

Award at the IEEE International Conference on Software Testing, Verification and Validation (ICST), 2012. He was awarded the NSERC Canada Graduate Scholarship (CGS-D) in 2014, and he received two ACM CAPS-GRAD travel grants from 2014 to 2015 to attend and present his papers at the International Conference on Software Engineering (ICSE). He has served in the program committees of the International Conference on Dependable Systems and Networks (DSN'19) and the International Symposium on Software Reliability Engineering (ISSRE'17) among others, and has reviewed for many reputable software engineering journals, including JSS, EMSE, TDSC, SPE, and STVR. He is a member of the IEEE Computer Society.