Perphecy: Performance Regression Test Selection Made Simple but Effective

Augusto Born de Oliveira*, Sebastian Fischmeister*, Amer Diwan†, Matthias Hauswirth‡ and Peter F. Sweeney§
*Dept. of Electrical and Computer Eng., University of Waterloo, Waterloo, ON, Canada, {a3olivei,sfischme}@uwaterloo.ca
†Google Inc., Mountain View, CA, USA, diwan@google.com
‡Faculty of Informatics, Università della Svizzera italiana, Lugano, Switzerland, Matthias.Hauswirth@usi.ch
§Turbonomic (work done while at IBM Research), New York, NY, USA, peter.sweeney@turbonomic.com

***Abstract*—Developers of performance sensitive production software are in a dilemma: performance regression tests are too costly to run at each commit, but skipping the tests delays and complicates performance regression detection. Ideally, developers would have a system that predicts whether a given commit is likely to impact performance and suggests which tests to run to detect a potential performance regression. Prior approaches towards this problem require static or dynamic analyses that limit their generality and applicability. This paper presents an approach that is simple and general, and that works surprisingly well for real applications.**

I. Introduction

The performance of a computer system determines its cost and usability (e.g., [1]). Consequently, many software projects maintain both a *test suite* for exposing correctness bugs and a *benchmark suite* for exposing performance bugs. Accurately measuring the performance of real systems is expensive, because many performance improvements or degradations only show up once the system is warm (e.g., once all the software and hardware caches are warm and the system is under full load) and, to account for run-to-run variations, each benchmark needs to be run multiple times [2]. As a consequence, it is often impractical to run a benchmark suite multiple times each time a developer commits code to the software repository; therefore, developers often check for changes in performance only at significant milestones (e.g., periodically at the end of the week or on each public release of the system).

Infrequently running the benchmark suite may expose changes in performance; however, infrequent runs will not identify which of the multiple commits, since the last benchmark suite run, is responsible for that performance change. Worse, the performance impact of different commits may cancel each other out and thus the developers may not realize that some commits improve performance while others degrade performance.

The problem thus is that *running the benchmark suite at each commit is too costly, but skipping it complicates performance detection*. This paper describes and evaluates a system that predicts if a code commit will affect the performance of a benchmark in the benchmark suite; in this way developers can determine the performance impact of their commit without performance testing the entire benchmark suite.

Our approach is *light-weight*, *general*, and *effective*. It is light-weight because it does not require expensive white-box program analyses, and does not require executing the entire benchmark suite when a performance change is predicted to occur. It is general because it works for arbitrary systems written in arbitrary programming languages. Finally, to demonstrate its effectiveness, we implemented our approach in a tool, Perphecy, and used it to predict change in performance on real open-source applications.

Through extensive evaluation, we found that Perphecy reduces the number of benchmarks that must be run at a code commit by up to 83%. Thus, Perphecy enables developers to catch performance changes early by enabling them to run performance tests more frequently than only at significant milestones.

The remainder of this paper is structured as follows: Section II introduces the background and related work, Section III presents our approach and its implementation in Perphecy, Section IV experimentally evaluates our ideas, Section V discusses threats to validity, and Section VII concludes.

II. Background and Related Work

This section provides the context for our contributions, introducing regression benchmarking, continuous integration systems, regression test selection, and performance bug detection in the field.

a) Regression Benchmarking: Detecting performance regressions is expensive for two main reasons. (1) Unlike functional regressions, performance regressions tend to manifest themselves especially in long-running code and for large input sizes. (2) The non-determinism prevalent in modern systems severely complicates the reliable detection of performance regression [3], and further drives up the regression benchmarking cost. While frameworks exist that reduce the complication of benchmarking, they do not reduce the cost. BEEN [4] is a general infrastructure for automated regression benchmarking in a heterogeneous distributed environment. BEEN compiles software and benchmarks, takes care of deployment, runs benchmarks, and collects, evaluates, and visualizes results. DataMill [5] extends these ideas into a community-based infrastructure for reliable performance evaluation. It allows varying several environmental factors known to affect performance [6], which enables the production of robust, reliable, and reproducible results.

Many larger-scale systems projects implement their own performance regression testing infrastructures. For example, the Jikes RVM [7] project used to run performance benchmarks every 12 hours, checking out the most recent version of their VM, building it, and running a suite of benchmarks including DaCapo [8]. They visualized the resulting evolution of benchmark performance on a public web page [9]. The Jikes RVM project did not run performance benchmarks after every commit. On days without commits, this constituted a waste of resources. On days with many commits, it meant a loss of coverage.

Mozilla's Talos performance regression detection system [10] runs performance tests "every time a change is pushed to the Firefox source repository" [11], and detects a performance regression if the performance changed significantly from before and after that push. They provide an online visualization of all the collected performance test results [12]. Talos uses performance tests, not necessarily full-fledged benchmarks. Running such tests can cost much less time than running complete representative workloads; however, testing in this manner also means that developers have to write a specific performance test for every relevant aspect of the application, and that they may not detect performance issues that only manifest themselves in realistic situations with large inputs.

The Linux Kernel Performance project [13] runs and reviews performance regression tests on a weekly basis, and for each major kernel release [14]. They reformat the disk, reboot the system, and run a warm-up load before each benchmark run. To minimize measurement variation, they use long benchmark running times and multiple repetitions, which drives up the cost. Despite the significant size of their test suite, they find that their tests only cover a portion of performance regressions, and they call for volunteers to contribute additional resources to enable more extensive benchmarking.

b) Continuous Integration Systems: Continuous integration systems perform fully automated builds and regression tests of a software. Traditional systems focus on *correctness* tests, which either are run at regular intervals or are triggered by individual commits. Systems like Jenkins [15] provide plug-ins to also support *performance* testing and reporting of performance numbers. Continuous integration is now also offered as cloud-based services, such as drone.io [16], Travis CI [17], or Coveralls [18]. Our approach could be particularly helpful in such scenarios by significantly reducing performance testing costs.

c) Regression Test Selection: Our approach is a form of regression test selection [19]. However, traditional regression test selection techniques focus on correctness tests, not on performance regressions.

For example, Ekstazi [20] uses run-time analysis to determine the dependencies of each test in a suite at file resolution, i.e., which files contain code reachable by each test, and which files are opened during each test (e.g., plaintext configuration files). If a file changes and a test depends on that file, (file changes are checked through checksum comparisons), then Ekstazi marks that test for execution. In experimental evaluation, Ekstazi has more than halved test suite execution times. While this approach is effective for correctness tests, it ignores that most changes to code will cause no performance change at all, and thus we believe that Perphecy is better suited for performance test selection.

One exception among regression test selection approaches is "performance risk analysis" (PRA), which is close in spirit to our work. Huang et al. [21] motivate their approach by showing that performance regressions are prevalent in commonly-used software, that performance tests can run for hours or days, and that common software projects evolve rapidly (e.g., Chrome and Linux with over 100 commits each day), making it almost impractical to run the complete suite of performance tests for every commit. They present a "white-box" approach to performance regression test selection that requires a static analysis to determine the "expensiveness" and the "frequency" of the code affected by a commit. Their analysis requires inter-procedural control and data dependence information that can be difficult to determine with adequate precision. This is especially problematic for modern languages where programs are dominated by heap data and polymorphic calls. They present PerfScope, an implementation of PRA on top of LLVM. Their evaluation of PerfScope on a set of large projects shows that by testing 14-22% of the commits, developers will catch 87-95% of the performance regressions.

To address the imprecision of PRA for programs written in dynamic languages, Sandoval Alcocer et al. [22] propose "horizontal profiling". They profile a *prior* version of the application to determine precise execution counts for each code block. They present LITO, an implementation of that approach for Pharo, a Smalltalk-like dynamic language. Their evaluation of LITO on a set of 17 Pharo programs, ranging in size between 9 and 404 classes, shows that by profiling 17% of the commits, developers will catch 83% of the performance regressions (where regression is defined as at least 5% increase in execution counts).

Perphecy addresses the regression test selection problem in a different way. We use a generic approach to gather dynamic information about prior application versions, and combine this with static information about the current and prior version. We then train an application-specific prediction model that uses the available information to predict which performance tests to run for a given commit. Our approach is light-weight and language-independent, and it can automatically learn accurate prediction models for arbitrary applications.

d) Performance Bug Detection in the Field: End user applications also can collect performance data in the field, send that data back to the developers, allowing them to determine performance issues in the environments in which they are actually used [23], [24], [25]. Open-source developers like the Mozilla Foundation even provide publicly accessible dashboards showing the aggregate performance data as measured in deployed instances of their applications (e.g., for Firefox [26]).

The advantage of such approaches is that performance is measured where it actually matters, in the users' context. The disadvantage is that performance problems are only detected

after they have affected a user. While we are not aware of any such systems, one could envision approaches to run performance tests in the deployed applications. These could detect performance issues, in the users' contexts, before the users encounter them. In such a system, our approach could possibly be extended to predict which tests to run in which contexts.

III. PERPHECY

We now present our approach to performance regression test selection and its implementation in the form of Perphecy. Our approach has three key parts: (A) Perphecy collects static data for new commits and collects dynamic data from benchmark runs; (B) Perphecy derives performance change indicators from that data, and (C) Perphecy uses these indicators to predict whether or not a new commit will impact performance for each benchmark in the benchmark suite.

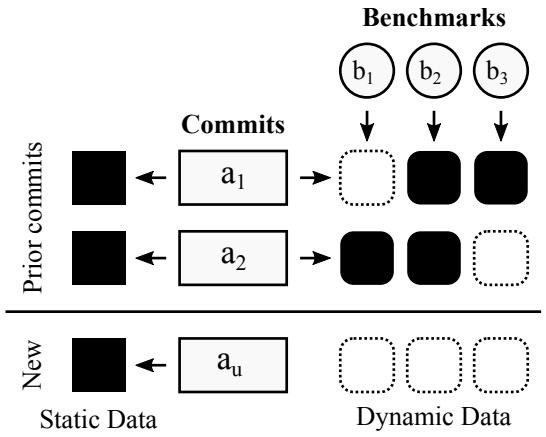


Fig. 1: Static and dynamic data collected by Perphecy.

When a developer commits code to the software repository, Perphecy collects static data without running any benchmark against the new commit. Whenever developers run one or more benchmarks (e.g., because Perphecy predicted a performance change or because of a periodic milestone) Perphecy collects dynamic data from those benchmarks. Perphecy uses both static and dynamic data to predict if a performance change might occur for a benchmark. For a new commit, a_u, and benchmark b, Perphecy (i) finds the closest ancestor to a_u (call it a_i) for which a_i has dynamic data for b; (ii) plugs the static data for the a_u and static and dynamic data for a_i into our indicators; and finally (iii) uses the instantiated indicators to predict whether or not a_u may change the performance of b.

Figure 1 provides an example. The repository contains two prior commits, represented by labeled rectangles (a_1 and a_2), and we use three benchmarks, represented by circles (b_1, b_2, and b_3) for performance regression testing. Perphecy has dynamic data from four prior benchmark runs, represented by the filled-in black boxes with rounded corners (a_1 with b_2 and b_3, as well as a_2 with b_1 and b_2). Missing dynamic data is represented by dashed boxes with rounded corners (a_1 with b_1, as well as a_2 with b_3). When a new commit, a_u comes in, Perphecy has to predict for which of the three benchmarks the new commit a_u may cause a performance change. To predict whether the performance of benchmark b_1 will change (and thus whether one should run a_u with b_1), Perphecy uses static data (represented by the filled-in black squares to the left of the commits) from a_u, and static and dynamic data from the closest ancestor commit a_2. Perphecy uses the same data to predict whether the performance of benchmark b_2 will change. However, to predict whether the performance of benchmark b_3 will change, there is no dynamic data available for a_2's performance on b_3. Thus, in addition to the static data from a_u, Perphecy uses static and dynamic data from the older commit a_1, because a_1 is the closest ancestor *for which dynamic data for b_3 is available*.

The rest of this section describes how Perphecy derives a predictor.

A. Collecting Data

As developers commit code to the software repository, Perphecy collects and stores data for use in predictions. Because predicting which benchmark's performance will change is on the critical path of a commit, Perphecy uses only cheaply obtained static data to make the prediction. Perphecy obtains static data by analyzing the compiled code (the binary) of the program. Using the binary, Perphecy trivially ignores non-semantic changes to the application (such as comments).

Whenever a benchmark is run to determine performance changes, due to a prediction or a milestone, Perphecy collects, from a separate background run, dynamic data to use in future predictions. Specifically, Perphecy runs a benchmark only once against a commit and this run is off the critical path of making a decision. Perphecy uses readily obtained dynamic data that can be obtained via multiple means: indeed we have experimented with both Pin [27] and Perf [28] for collecting this data. Using either Pin or Perf, Perphecy collects only the number of times each function is called, and how many instructions are executed in each function.

We decided against collecting more complex dynamic data such as full call graphs or instruction-level traces because (i) we did not want our dynamic runs to be expensive (even though they run off the critical path) and (ii) we did not want to reduce the applicability of Perphecy by requiring language or environment specific tools.

We evaluate Perphecy on four projects (Section IV-2). Perphecy uses Pin for Git, glibc and HotSpot, and Perf for MongoDB. We use different methods for different projects because some benchmarks crashed with either Pin or Perf. Fortunately, because Perphecy uses simple dynamic data, we can readily substitute tools for collecting the dynamic data.

B. Indicators

Perphecy "boils down" the collected data into indicators. This process reduces data for commits (e.g., a list of reached functions for each benchmark) to a Boolean value related to

Indicator template	Description	Rationale	Source Data
Del Func $\geq X$	True if the number of deleted functions (i.e., functions in a_i that are not in a_u) is greater or equal to X	Deleted functions indicate refactoring, which can lead to overall performance changes	static
New Func $\geq X$	True if the number of new functions (i.e., functions in a_u that are not in a_i) is greater than or equal to X	New functions indicate new functionality or refactoring, which can lead to overall performance changes	static
Reached Del Func $\geq X$	True if the number of deleted functions, which are called by b, is greater than or equal to X	Deleted functions indicate refactoring, and when reached in execution can lead to overall performance changes	static + dynamic
Top Chg by Call $\geq$X%	True if one of the X% most called functions changed from a_i to a_u	Performance-affecting changes in a function are multiplied by the number of times that function is called	static + dynamic
Top >X% by Call Chg by $\geq$10%	True if one of the X% most called functions changed by at least 10% of its static instructions from a_i to a_u	Performance-affecting changes in a function are multiplied by the number of times that function is called and the magnitude of the change	static + dynamic
Top Chg by Instr $\geq X\%$	True if one of the X% longest running functions (by total dynamic instructions) changed from a_i to a_u	Changes in the longest-running functions are more likely to correlate with overall performance changes	static + dynamic
Top Chg Len $\geq X\%$	True if any function's static instruction length changed by at least X%	Large changes to functions are more likely to affect performance than small ones	static
Top Reached Chg Len $\geq X\%$	True if any function, which is called by b, has its static instruction length changed by at least X%	Same as above, with the addition that the function is called by b	static + dynamic

TABLE I: Indicator template definitions for Pin profile data, where a_u is the new commit which only has static data and a_i is an older commit which has both static and dynamic data (where dynamic data was computed for a specific benchmark b).

the difference between a new commit and an ancestor commit (e.g., how many reachable functions have changed).

Table I lists the *indicator templates* for Pin profiling data, the rationale for why they would correlate with performance changes, and if they are derived from static data alone or from a mix of static and dynamic data. An *indicator* is a concrete instantiation of an indicator template using a specific threshold value for its parameter X. We use analogous indicators for data from Perf. Note that while Perphecy as a strategy is not language or environment dependent, indicators may be (e.g., ranking changed functions by call count is only possible in languages that compile to binary functions, and for which a profiler is available).

Each indicator converts a complex signal into a Boolean. For example, the number of deleted functions between commits can range from zero to the number of functions in the old commit (i.e., all were deleted). The "Del Func" indicator reduces this value to a Boolean by using a threshold X. In particular, "Del Func > X" is $true$ if the number of deleted functions between the two commits is greater than X, otherwise it is $false$. To obtain a predictor for an application we must (i) learn the threshold value X for each indicator template to generate an indicator; and (ii) combine the indicators into a Boolean expression to form a predictor.

C. Evaluating indicators

We evaluate indicators using two metrics: *hit rate* and *dismiss rate*. Let H be the set of (a_u, a_i, b) tuples such that the performance of benchmark b is different between commits a_u and a_i. We call these tuples *hits*. H is the ground truth. H_p is the set of (a_u, a_i, b) tuples for which an indicator returns $true$; thus, H_p is a prediction. An indicator's hit rate is defined as follows:

a) Indicator Hit Rate: $|H_p \cap H|/|H|$

Intuitively, the hit rate is a value between 0.0 and 1.0 that indicates the fraction of performance changes the indicator *correctly* predicted.[1] An indicator that simply returns a $true$ value independent of input will, by definition, have a perfect hit rate of 1.0. We say that an indicator *covers* a hit if it correctly predicts the hit.

Let D be the set of (a_u, a_i, b) tuples with no measurable performance change. We call these tuples *misses*. D is the ground truth. Let D_p be the set of (a_u, a_i, b) tuples for which an indicator indicates $false$ (i.e., no performance change for benchmark b). D_p is the prediction. Dismiss rate, is defined as:

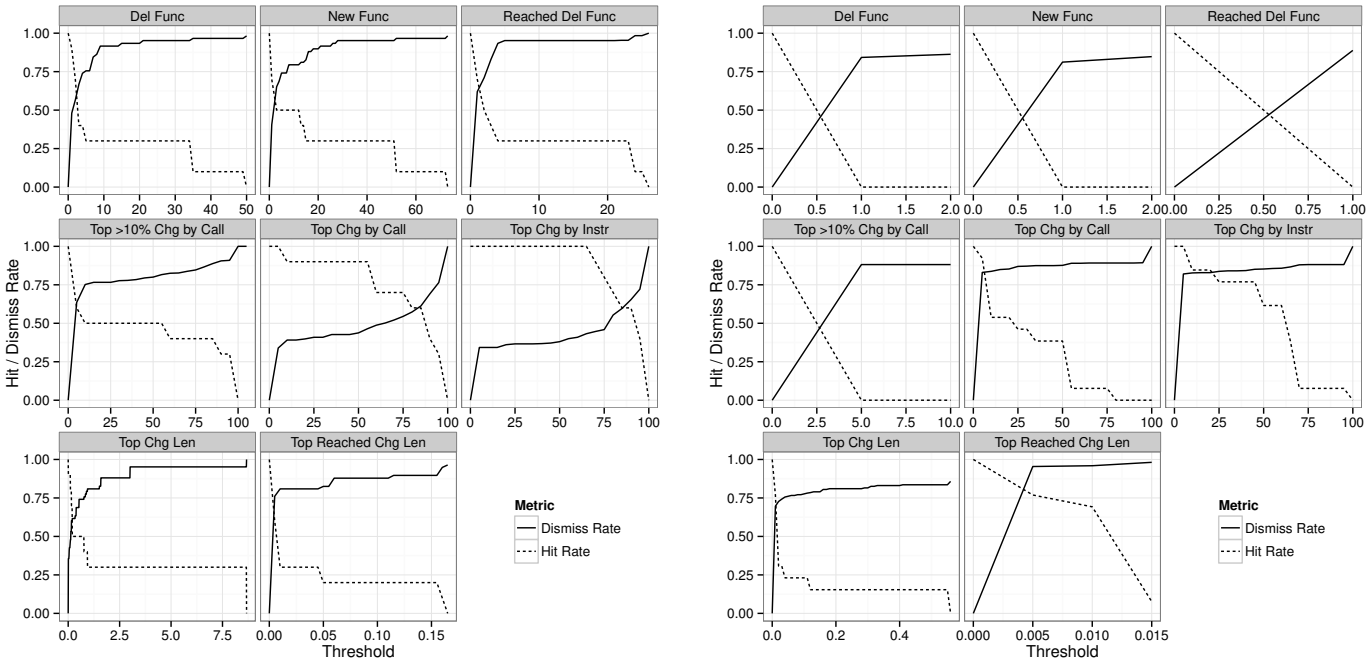
b) Indicator Dismiss Rate: $|D_p \cap D|/|D|$

The dismiss rate is a value between 0.0 and 1.0 that indicates the fraction of benchmarks that the indicator correctly dismissed. An indicator that always returns $false$ will have a perfect dismiss rate of 1.0. An optimal indicator will have *both* a hit rate and a dismiss rate of 1.0; however, such an indicator will be difficult to create for any software project. An effective indicator will have high hit and dismiss rates.

Figure 2a shows the hit rate (dashed lines) and dismiss rate (solid lines) of these indicator templates for the HotSpot program; we evaluate each indicator template with a range of values for the threshold X. For example, an X of 10 for the "Del Func" indicator gives the hit and dismiss rates when the threshold for the number of deleted functions is ten. As expected, as thresholds increase the indicator becomes more selective and thus the hit rate decreases, while the dismiss rate increases. We see that some indicators are more effective at prediction than others.

The best individual indicator for HotSpot, "Top Chg by Instr" with its threshold set at 65, has a 1.0 hit rate (i.e., its value is $true$ for all commit pairs with performance changes), and a dismiss rate of 0.57 (i.e., only 43% of commits would be unnecessarily sent for benchmarking by this indicator). "Top Reached Chg Len", a less effective indicator individually, has an immediate drop-off in hits as the threshold increases. Nevertheless, this indicator is not without merit, because up to a 15% threshold, the dismiss rate is high (> 75%) and, although the hit rate is low (< 25%). As we will see in Section III-D, by combining indicators with separate strengths,

[1] This is equivalent to the recall metric in pattern recognition.



(a) HotSpot

(b) Git

Fig. 2: Indicator performance at various thresholds for HotSpot and Git.

we can derive a predictor with high aggregate hit and dismiss rates.

Figure 2b shows the hit and dismiss rates of the same indicators for Git. The metrics for each indicator differ from the HotSpot results, which suggests that different software projects will require different thresholds for different indicators. Also "Reached Del Func" is effectively useless for Git because it has a hit rate of 0.0 as soon as its threshold increases over zero.

D. Making Predictions

From Figures 2a and 2b, we know that different indicators have different strengths and that the same indicator performs differently for different programs. The weakness of any individual indicator is not surprising, because the indicators are individually trivial. Thus, Perphecy combines indicators to construct more complex predictors and learns thresholds that are specific to each program and each benchmark.

Pair	Dels	Adds	Δ_b	Del Func≥ 5	Del Func$\geq 10 \vee$ New Func≥ 9
P_1	5	9	T	T	$F \vee T = T$
P_2	5	3	F	T	$F \vee F = F$
P_3	10	1	T	T	$T \vee F = T$
Hit				1.0	1.0
Dismiss				0.0	1.0

TABLE II: Combining indicators to construct predictors.

For example (Table II), consider three commit pairs, P_1, P_2, and P_3, and the new commit in each of these pairs deletes (adds) 5 (9), 5 (3), and 10 (1) functions respectively. Furthermore assume that for a given benchmark b there is a performance difference Δ_b in P_1 and P_3, but not in P_2. If we use the "Del Func" indicator template only, a threshold of 5 gives us the good hit rate (of 1.0), but it yields a poor dismiss rate (of 0.0), because it incorrectly predicts a performance change for P_2. Analogously, if we use just the "New Func" indicator template, we have to either sacrifice the hit rate or the dismiss rate. By using the disjunction of both indicators with thresholds of 10 (for "Del Func") and 9 (for "New Func"), we get a predictor with perfect hit and dismiss rates.

We can use many off the shelf techniques for by picking thresholds for our indicator templates to construct indicators. In this paper, we use a straightforward greedy approach and show that even this simple approach is effective. We leave exploring more sophisticated predictors to future work.

To further simplify our approach, we only consider predictors that are a trivial disjunction of indicators. As with indicators, we use the hit and dismiss rate metrics to evaluate our predictors.

While using disjunctions of indicators gives us better predictors than just using individual indicators, it is easy to construct examples where the disjunctions are inadequate (e.g., if two indicators both have to be $true$ for a prediction). To handle such cases, we can either support more complex combinations (e.g., also including conjunctions) or add new indicators (which effectively give us the conjunctions that we deem interesting). We have explored some alternatives and found, at least for the alternatives we considered, that

they offered little benefit in practice. Thus, we consider only simple predictors (indicators combined with disjunctions) in this paper, but recognize that there is room for improvement.

```
   Data: H= {(a_u, a_i, b) : hit},
   Data: D= {(a_u, a_i, b) : dismiss},
   I= {i_k : 1 ≤ k ≤ n_ind}
   Result: T= {(i_k, t_k) : i_k ∈ I, t_k ∈ Integer}
1  T= ∅;
2  for h ∈H do
3  |  min_price_thresh[h] = null;
4  |  min_price[h] = ∞;
5  |  min_price_ind[h] = null;
6  |  for i ∈I do
7  |  |  thresh_for_hs = maxthresh(h, i);
8  |  |  price_for_hs = | allhits(i, thresh_for_hs, D) |;
9  |  |  if price_for_hs < min_price[h] then
10 |  |  |  min_price[h] = price_for_hs;
11 |  |  |  min_price_thresh[h] = thresh_for_hs;
12 |  |  |  min_price_ind[h] = i;
13 |  |  end
14 |  end
15 end
   /* Invariant: for each hit, the min_price
      structures identify the indicator
      template and threshold that can cover
      the hit at the lowest price          */
16 C= H;
17 while C≠ ∅ do
18 |  max_min_price = 0;
19 |  target_ind = null;
20 |  target_thresh = null;
21 |  for h ∈ C do
22 |  |  if min_price[h] > max_min_price then
23 |  |  |  target_thresh = min_price_thresh[h];
24 |  |  |  max_min_price = min_price[h];
25 |  |  |  target_ind = min_price_ind[h];
26 |  |  end
27 |  end
28 |  T = T ∪ {(target_ind, target_thresh)};
29 |  C = C \ {allhits(target_ind, target_thresh, C)};
30 end
```

Algorithm 1: Greedy heuristic to pick indicator thresholds.

Algorithm 1 presents our greedy heuristic to pick indicator thresholds. Our algorithm attempts to come up with a predictor that *covers all the hits* in a training set, while maximizing dismiss rate (minimizing unnecessary benchmarking). This means that our predictor will likely be conservative. The training set is made up of two sets of triples. H are triples that give a pair of commits and a benchmark that performs differently for the two commits. D are triples that give a pair of commits and a benchmark that performs the same for the two commits. In addition to the training set, our algorithm takes as input a set of indicator templates I. The goal of the algorithm is to learn thresholds for each indicator template in I. The output of the algorithm is a set of indicators, T, which we use for prediction.

We can use our algorithm in two ways: (i) to learn a predictor for each benchmark, one benchmark at a time; in this case the H and D are all for the same benchmark; and (ii) to learn a predictor for a benchmark suite; in this case H and D will contain tuples for different benchmarks. Even if we learn a single predictor for the entire benchmark suite, our predictions are still specific to individual benchmarks, because the indicators that include dynamic state will be different for different benchmarks. For example, two benchmarks will differ in how often they (dynamically) exercise a particular function and thus their "Top Chg Len by Instr" indicator will be different even if they use the same threshold. This is essential in order to minimize benchmarking time, because it allows executing only small subsets of the benchmark suite at each commit.

The algorithm uses two auxiliary functions: (i) $maxthresh(h, i)$ returns the maximum threshold for the indicator template, i, that still covers the hit, h; and (ii) $allhits(i,\ t,\ \{(a_u, a_i, b)\})$ returns the tuples in the set, $\{(a_u, a_i, b)\}$, for which the indicator template, i, evaluates to $true$ at the given threshold, t.

The first stage of the algorithm (lines 3-14) iterates over all the hits and finds the indicator that covers the hit at the lowest price (i.e., it returns true for the fewest tuples in the dismiss set). It stores this information in the "min_price" structures. If, for each indicator template we pick the lowest threshold recorded in the min_price structures, we obtain a *trivial predictor* by ORing the indicators together which will return $true$ for all the hits; however, this predictor will also return $true$ for some of the dismisses.

The second stage of the algorithm (lines 20-26) greedily refines the output of the first stage so that the overall predictor still covers all the hits, but may have an overall lower price than the trivial predictor suggested above. At each iteration, this stage finds the hit, h, with the highest price (using the min_price structures) and adds its indicator to the output. From stage 1, we know that we cannot do any better for h; thus we *have* to pay this price to cover h. By picking the hits with the highest price first, our algorithm hopes that the associated indicator may cover not just the current hit, but also other hits, eliminating the need for other indicators, which may increase the overall price. The algorithm uses the $allhits$ function to removes all the hits covered by this indicator. The algorithm is done when no more hits need to be covered. Alternative algorithms, which allow conjunctions or miss some hits, could do better.

This algorithm depends on a non-empty set of hits to operate correctly. If $H = \emptyset$, such as for the glibc project (Section IV-2), the resulting predictor will be trivial: the thresholds for each of its indicators will be at their maximum possible value, such that its dismiss rate equals 1.0 over the training set, and no matter what code change the predictor is used on, it will return $false$. It is essential, therefore, that developers ensure their training sets include as many hits as possible to avoid such overfitting.

IV. Experiments

In this section we study two questions: (1) What is the potential for any performance regression test selection technique? (2) And how much of that potential can Perphecy realize?

Software	Description	SLOCs	Commits/Day	Tests	Benchmarks	Commits	Hits
Git	Version control system	312,398	11.78	init, add, commit, diff, clone	5	201	13
glibc	C library	1,174,764	4.94	libc-bench suite	6	98	0
HotSpot	Java virtual machine	738,720	4.99	DaCapo suite	10	50	10
MongoDB	Database management system	1,657,728	10.18	insert, insert then select	2	80	27

TABLE III: Software projects used in experiments.

We study those questions using the repositories of the four open-source projects in Table III: git, a popular version control system; glibc, a runtime library used by most applications on Linux; HotSpot, the dominant Java virtual machine; and MongoDB, a database used in many modern web applications. We selected those projects for their widespread use, high degree of complexity, and extensive development history. For each project, we selected a continuous sequence of commits going back in time from when we started our experimentation. In total, we checked out and built 429 commits. For each project, we then determined a benchmark suite.

Table III shows the name and description for each software project. The SLOCs column shows the number of source lines of code for the oldest commit investigated[2]. The Commits/Day column shows the average number of daily commits for the year 2013. The Tests column shows the the benchmark sets we executed on each of them. The Benchmarks column shows the number of individual benchmarks in each set[3]. The Commits column shows the number of commits compiled. Finally, the Hits column shows the number of (a_u, a_i, b) tuples, where a_u is a direct child of a_i, and where there is a performance change.

1) General Potential: To evaluate the general potential for performance regression test selection, we determine what fraction of commits and benchmarks do *not* encounter a statistically significant change in performance. This fraction represents the potential for reducing wasted work, because running those benchmarks on those commits will find no performance change.

For our experimental methodology, we compiled all the commits for all of the software projects and then executed each benchmark on each commit at least five times. Executing a benchmark multiple times allows us to estimate variance and then perform a t-test with $\alpha = 0.05$ to determine if there was a difference in performance[4]. The metric for each benchmark was its total execution time for all cases except DaCapo, which reports its own metric. Two machines executed all experiments, each with a four-core 3.60GHz Core i7-3820 CPU, one with 24GB of RAM and the other with 32GB of RAM. To ensure independence between samples, we rebooted the machines after each run of a benchmark and a commit, and randomized the order in which we executed the benchmarks and commits. These experiments took over five weeks to run and generated gigabytes of decompiled binaries and trace data.

The first notable result is that the number of significant performance changes, shown in the "Hits" column of Table III, is low. In the case of the HotSpot dataset, we found only 10 out of 570 (a_u, a_i, b) tuples[5] to have a performance change. This means that the opportunity to reduce wasted work for HotSpot is large: 98.2%. Unfortunately, this also means that missing a single performance change would represent missing 10% of all significant changes.

For glibc we did not find a single commit that incurred a performance change on *any* of its benchmarks. In the 285 individual (a_u, a_i, b) tuples we investigated, only two reachable binary functions ever changed, `__init_cpu_features` and `_IO_file_open`, and neither code change caused a significant performance effect on the benchmarks. Because glibc is by far the oldest software project investigated here — its initial release was in 1987 — it is to be expected that its user-facing, performance-sensitive functions (i.e., the ones exercised by our benchmark suite) will not change frequently; however, having no hits poses a challenge to our prediction strategy.

In conclusion, the rarity of performance changes supports Algorithm 1 bias to cover all hits in the training set.

2) Effectiveness of Perphecy: After having established the general potential for performance regression test selection, we can evaluate to what degree Perphecy realizes that potential. Our evaluation addresses the following questions: (a) How effective is Perphecy in an idealized scenario where we train and evaluate on the same commits and benchmarks? (b) How effective is Perphecy in regression test selection? (c) What is the effect of the training set size on predictor performance? (d) Is it possible to use a predictor trained on one project to predict for another project? (e) What is the effect of using dynamic information from older commits? (f) What is the effect of training the predictor using synthetic training sets of non-adjacent commit pairs?

a) Effectiveness in an idealized scenario: When Perphecy trains and evaluates on the full set of commits and benchmarks, we get a sense for how well our approach works in the ideal case. This provides a baseline for what we can possible achieve using our algorithm.

[2]Lines of code were counted using the SLOCCount tool.

[3]Although the DaCapo 9.12 suite contains 14 benchmarks, we were not able to include four of the benchmarks (eclipse, h2, tradebeans and tradesoap), because they did not run on all of the commits; that is, the benchmarks exposed some bugs. Therefore, we only used 10 of the DaCapo benchmarks in our experiments.

[4]Normality of the data was ascertained graphically. Bonferroni correction was not performed, since each commit's performance data was used only once per commit graph edge (i.e., twice in most cases).

[5]How we came up with 570 tuples for HotSpot requires some explaining. Although Table III identifies 50 commits for HotSpot, there are 57 parent-child commit pairs due to merges in the commit history. Therefore, we evaluated 570 tuples by comparing the results of 10 benchmarks on 57 parent-child commit pairs.

Software	Full Hit	Full Dismiss	K-fold/Individual Hit	K-fold/Individual Dismiss	K-fold/All Hit	K-fold/All Dismiss
HotSpot	1.0	0.89	0.40	0.88	0.70	0.47
Git	1.0	0.83	0.77	0.84	0.85	0.83
MongoDB	1.0	0.46	0.74	0.46	1.0	0.46
glibc	n/a	1.00	n/a	1.00	n/a	1.00

TABLE IV: Effectiveness.

The "Full" columns Table IV show the average hit and dismiss rates achieved by the algorithm when applied to each benchmark. For HotSpot, Git and MongoDB, the hit rate is 1.0 by construction (because the predictors cover all hits). The dismiss rate column shows that, for HotSpot and Git, the algorithm is able to dismiss more than 83% of the performance-neutral commits, while the predictor for MongoDB has a comparatively low dismiss rate of 46%. Because the glibc training set contains no hits, the resulting trivial predictor has a 1.0 dismiss rate.

b) Realistic Effectiveness: We now measure the effectiveness of Perphecy in actually *predicting* performance changes by using k-fold cross-validation [29] with $k = 10$. k-fold cross-validation trains on 90% of the commits and evaluates on the remaining 10%; it repeats this 10 times, each time for a different 10%.

We use two different strategies: (1) training a separate predictor for each benchmark (the "K-fold/Individual" columns), and (2) training a common predictor across all benchmarks (the "K-fold/All" columns). The second case learns a single set of thresholds for all benchmarks; as discussed in Section III the prediction is still specific to each benchmark because the predictor also considers dynamic data which vary across benchmarks.

When we have a separate predictor for each benchmark (the "K-fold/Individual" columns), the predictor's hit rates range from 0.4 to 0.77, and dismiss rates range between 0.46 and 0.88. When we have a common predictor across all the benchmarks (the "K-fold/All" columns), the predictor's hit rates improve, ranging from 0.70 to 1.0, and dismiss rates stay about the same, ranging from 0.46 to 0.83. Although appearing to be counter intuitive, the common predictor has a higher hit rate than a per-benchmark predictor, because the per-benchmark predictor over fits the indicators to individual benchmarks. In contrast, a common predictor must find a threshold for each indicator that results in a hit for all the benchmarks, thereby the threshold computed by $maxthreshold$ in Algorithm 1 is lower than the threshold computed for each benchmark individually. From the graphs in Figure 2a and 2b, we know that the hit rate diminishes with higher thresholds; therefore, the per-benchmark predictors have lower hit rates than the common benchmark predictors.

c) Training Set Size: To determine the size of training set needed to generate a good predictor, we use k-fold cross-validation once again. In particular, we use the common predictor across all the benchmarks. Because the size of the training set grows as the value of k grows, we can investigate the effect of the training set size on the quality of the

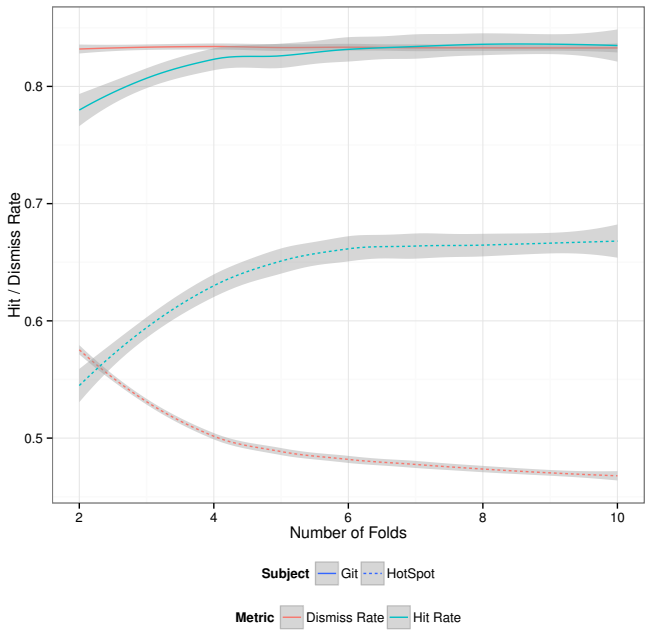


Fig. 3: Predictor hit and dismiss rates by size of training dataset.

predictors. Figure 3 shows smoothed curves for the hit and dismiss rates for each project by k, where $2 \leq k \leq 10$. The shaded area surrounding each line represents the confidence interval, because we partitioned the commits multiple times into different training sets. When $k = 2$, the training sets are 50% of the data, and predictions are made for the other 50%, and when $k = 10$, the training sets are 90% of the data and predictions are made for the remaining 10%. The figure shows that, as the training set grows, there is a marked improvement on the hit rate, accompanied by a minor decrease in the dismiss rate. Note that the y-axis ranges from 0.45 to 0.85. In the case of Git, the improvement to the hit rate levels off at $k = 6$, or approximately 171 commits in the training set.

d) Cross-Project Prediction: To evaluate if a single predictor can be used effectively for all software projects, we test the transferability of predictors between the projects (MongoDB is excluded, because its indicators are derived from Perf instead of Pin, and glibc is excluded, because its data set not containing any hits). A predictor trained on Git data and used for HotSpot results in a hit rate of 1.0 and a dismiss rate of 0.34, which means that lots of wasted work will be done. A predictor trained on HotSpot and used for Git results in a 0.0 hit rate and a dismiss rate of 0.88, which covers none of the hits but does little wasted work. These results show that using a Perphecy predictor trained on one software project to make predictions about another is ineffective at best. More generally, these results also show that a one-size-fits-all approach such as the one presented by Huang et al. [21] will miss opportunities to skip benchmarks, and that it is essential to learn each particular applications' and benchmarks' dynamic properties

in order to make the most effective predictions possible.

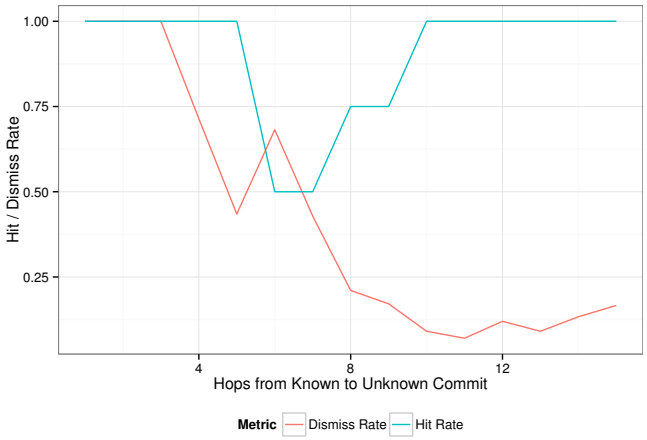


Fig. 4: Predictor hit and dismiss rates by distance between a two commits, for Git.

e) Distance to Last Known Commit: As development continues, if the predictor indicates that none of the incoming code changes will incur a performance change, developers may make multiple subsequent commits without Perphecy collecting dynamic data. In such a scenario, the last commit with known performance has a continuously growing distance from the newest commit with unknown performance. To determine the effect of this distance on predictor performance, Figure 4 shows the hit and dismiss rates of a predictor trained on the 25% oldest (by time stamp) commits of the Git dataset, making a series of predictions $(a_u,\ a_i,\ b)$ where a_u is the newest commit in the training set, to each a_i in the 75% of commits not in the training step. A *hop* is the distance between a_u and a_i counting a_i and intervening commits. The figure shows that there is a direct correlation between hops and dismiss rate. Between one and three hops, *all* commits are correctly dismissed (hit and dismiss rates equal 1.0), but after that the dismiss rate decreases nearly monotonically with hops until reaching 0.16 at 15 hops. This happens because the longer the distance between a_i and a_u, the larger the difference between the resulting binaries (all changes in the intervening commits are accumulated), and the more indicators will have their thresholds surpassed. No such effect is observed for hit rate, which varies between 0.5 and 1.0 with no pattern relating to hops. This result suggests that there is a natural effect limiting the length of a string of commits for which the predictor will return `false`. This effect is desirable because it bounds the number of hops a missed performance change will go unnoticed. In the scenario shown in Figure 4, the predictor starts returning $true$ as soon as four hops from a_i, which would have "reset" the latest known commit.

f) Synthetic Training Sets: Because there is no restriction on the relationship between commits involved in a prediction, we can generate "synthetic commits" to achieve a larger training set. We test this by exhaustively combining commit pairs: from v commits, we create $\binom{v}{2}$ commit pairs, greatly expanding the inputs to the predictor generation algorithm. Table V presents the results for k-fold cross validation ($k = 10$) on this exhaustive set of commit pairs for Git, HotSpot and glibc. The data shows that the predictors trained on such synthetic training inputs have nearly optimal hit rates; however, dismiss rates are low, less than 0.3. Because these artificial training sets include even the most distant commit pairs (in all metrics of distance: hops, commit date, *and* code delta size), the frequency and nature of hits they contain is different from what occurs in regular development. As the hit set grows, the threshold selection algorithm can only make the predictor more conservative, which suggests developers should limit the size of their training set once their dismiss rate levels off (as in Figure 4) or starts decreasing. Interestingly, the artificially generated training set for glibc included 398 hits in 14,259 prediction tuples, allowing a non-trivial predictor to be derived. The generation of synthetic commits is, therefore, a viable option for augmenting training sets with an insufficient amount of hits.

Software	Hit	Dism.
Git	1.00	0.08
HotSpot	0.99	0.03
glibc	1.00	0.29

TABLE V: K-fold cross validation for synthetic training sets.

g) Summary: This section showed that, for Git, Perphecy can predict as much as 85% of the performance-changing commits, while avoiding executing 83% of benchmarks. In the case of HotSpot, which complicates analysis by generating dynamic binary code during run-time, the predictor detects 70% of performance-changing commits, and dismisses nearly half of performance-neutral code changes. These predictors are based on inexpensive language and architecture independent static analysis of binary code, and lightweight profiling of benchmarking runs, that collects nothing but (1) the number of times each function is called, and (2) how many instructions are executed in each function. In the case of Git, for example, all processing operations needed to make a prediction executed in under $350ms$. The predictor for MongoDB, which used a different set of indicators based on hardware performance counters, detected 100% of performance-affecting changes, while dismissing 46% of performance neutral changes, suggesting that these indicators are effective in conservative predictors.

As indicated by the experimental data, Perphecy is not a substitute for running all the benchmarks against the latest commit at periodic intervals, for example, nightly or even weekly, because Perphecy does not have a perfect hit rate. Nevertheless, the hit and dismiss rates are good enough to use Perphecy at check-in time of a new commit to quickly decide what performance tests should be run. Any performance regressions Perphecy misses will be caught at the next periodic interval of running all benchmarks against the latest commit.

V. Threats to Validity

This study reports data for (i) four open-source projects; and (ii) a set of indicator templates. Thus, the main threat to validity is: how does our approach generalize to other projects or to different indicators. We intentionally picked projects that are actively developed, complex, and widely used. We picked only four projects due to the cost of analyzing each project; we spent more than a month of CPU time to analyze our four projects. While we have some evidence that the approach generalizes across at least these projects and across two methods of collecting dynamic data, more work is needed before we can be confident that our approach is broadly applicable.

VI. Discussion

Perphecy presents the first exploration using easily-obtained static and dynamic data for reducing the cost of performance benchmarking. It opens up many avenues for future research.

First, we explore only simple indicators and even simpler predictors. While we show that these predictors work well for four open-source applications, we do see there is clear room for improvement. We believe that exploring more powerful indicators, allowing conjunctions in predictors, and applying machine learning algorithms to learn predictors could give us more effective and robust predictors.

Second, programming languages that compile code dynamically or load code dynamically could present new challenges for our approach. For such languages, there is often not a single (or even a set of) binaries that contains the entire code of an application. This produces challenges for traditional static analyses.

VII. Conclusion

Because performance can make or break a software project, developers must detect performance regressions immediately. To avoid the overhead and turnaround time involved in checking for performance changes in every benchmark in a benchmark suite on every commit, we present Perphecy, a lightweight, general, and effective strategy to predict which commits will cause performance changes on which benchmarks. Although Perphecy is not a substitute for running all the benchmarks against the latest commit at periodic intervals, for example, nightly or even weekly, we have shown, by applying our approach to four widely used performance-sensitive software projects, that Perphecy can save as much as 83% of benchmarking time while still detecting 85% of performance-affecting code changes, which allows performance regression tests to be run at check in time.

References

[1] S. Stefanov, "Yslow 2.0." Presented at the CSDN Software Development 2.0 Conference, 2008. [Online]. Available: http://www.slideshare.net/stoyan/yslow-20-presentation

[2] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.

[3] T. Kalibera, L. Bulej, and P. Tuma, "Automated detection of performance regressions: The mono experience," in *MASCOTS*. IEEE Computer Society, 2005, pp. 183–190.

[4] T. Kalibera, J. Lehotsky, D. Majda, B. Repcek, M. Tomcanyi, A. Tomecek, P. Tuma, and J. Urban, "Automated benchmarking and analysis tool," in *VALUETOOLS*, ser. ACM International Conference Proceeding Series, L. Lenzini and R. L. Cruz, Eds., vol. 180. ACM, 2006, p. 5.

[5] A. Oliveira, J.-C. Petkovich, T. Reidemeister, and S. Fischmeister, "Datamill: Rigorous performance evaluation made easy," in *Proc. of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, Prague, Czech Republic, April 2013, pp. 137–149.

[6] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 265–276. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508275

[7] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen, "Implementing jalapeño in java," in *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '99. New York, NY, USA: ACM, 1999, pp. 314–324. [Online]. Available: http://doi.acm.org/10.1145/320384.320418

[8] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[9] DaCapo, http://dacapo.anu.edu.au/regression/perf/head.html, 2014.

[10] Talos, https://wiki.mozilla.org/Buildbot/Talos, 2014.

[11] M. Brubeck, "Better automated detection of firefox performance regressions," http://limpet.net/mbrubeck/2013/11/10/improving-regression-detection.html, November 2013.

[12] Mozilla, http://graphs.mozilla.org/, 2014.

[13] LKP, https://01.org/lkp/, 2014.

[14] T. Chen, L. I. Ananiev, and A. V. Tikhonov, "Keeping kernel performance from regressions," in *Proceedings of the Linux Symposium*, ser. OLS'07, June 2007, pp. 93–102.

[15] Jenkins, http://jenkins-ci.org/, 2014.

[16] Drone, http://drone.io, 2014.

[17] Travis, https://travis-ci.org/, 2014.

[18] Coveralls, https://coveralls.io/, 2014.

[19] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1002/stv.430

[20] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight test selection," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, May 2015, pp. 713–716.

[21] P. Huang, X. Ma, D. Shen, and Y. Zhou, "Performance regression testing target prioritization via performance risk analysis," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 60–71. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568232

[22] J. P. Sandoval Alcocer, A. Bergel, and M. T. Valente, "Learning from source code history to identify performance failures," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '16. New York, NY, USA: ACM, 2016, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/2851553.2851571

[23] A. Adamoli and M. Hauswirth, "Trevis: A context tree visualization & analysis framework and its use for classifying performance failure reports," in *Proceedings of the 5th International Symposium on Software Visualization*, ser. SOFTVIS '10. New York, NY, USA: ACM, 2010, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/1879211.1879224

[24] M. Jovic, A. Adamoli, and M. Hauswirth, "Catch me if you can: Performance bug detection in the wild," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New

York, NY, USA: ACM, 2011, pp. 155–170. [Online]. Available: http://doi.acm.org/10.1145/2048066.2048081

[25] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 145–155. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337241

[26] Telemetry, http://telemetry.mozilla.org/, 2014.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[28] Perf, https://perf.wiki.kernel.org/, 2014.

[29] D. C. Montgomery, *Design and Analysis of Experiments*. John Wiley & Sons, 2006.