Performance Regression Testing Target Prioritization via Performance Risk Analysis

Peng Huang, Xiao Ma†, Dongcai Shen, and Yuanyuan Zhou
Computer Science and Engineering
Univ. of California at San Diego
La Jolla, CA, USA
†Computer Science
Univ. of Illinois at Urbana-Champaign
Urbana, IL, USA
{ryanhuang,x1ma,doshen,yyzhou}@cs.ucsd.edu

ABSTRACT

As software evolves, problematic changes can significantly degrade software performance, i.e., introducing performance regression. Performance regression testing is an effective way to reveal such issues in early stages. Yet because of its high overhead, this activity is usually performed infrequently. Consequently, when performance regression issue is spotted at a certain point, multiple commits might have been merged since last testing. Developers have to spend extra time and efforts narrowing down which commit caused the problem. Existing efforts try to improve performance regression testing efficiency through test case reduction or prioritization.

In this paper, we propose a new lightweight and white-box approach, performance risk analysis (PRA), to improve performance regression testing efficiency via testing target prioritization. The analysis statically evaluates a given source code commit’s risk in introducing performance regression. Performance regression testing can leverage the analysis result to test commits with high risks first while delaying or skipping testing on low-risk commits.

To validate this idea’s feasibility, we conduct a study on 100 real-world performance regression issues from three widely used, open-source software. Guided by insights from the study, we design PRA and build a tool, PerfScope. Evaluation on the examined problematic commits shows our tool can successfully alarm 91% of them. Moreover, on 600 randomly picked new commits from six large-scale software, with our tool, developers just need to test only 14-22% of the 600 commits and will still be able to alert 87-95% of the commits with performance regression.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—Testing tools

General Terms

Performance, Experimentation

Keywords

Performance regression, performance risk analysis, cost modeling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE ’14, May 31–June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-5/14/05 ...$15.00.

Figure 1: A real-world performance regression (80 times) issue in MySQL: the change causes an expensive function call find_table_for_trigger to be executed many times.

1. INTRODUCTION

1.1 Performance Regression Testing

Performance is a vital quality metric for software system. It can directly affect user experience and job efficiency. For example, a 500 ms latency increase could cause 20% traffic loss for Google [44]. As another example, the Colorado Benefits Management System is designed to make social welfare accessible. But it runs so slowly that the system is virtually unable to accept assistance applications [18].

On the other hand, software today is evolving rapidly. Code commits for feature enhancement, bug fixing or refactoring are frequently pushed to the code repository. Some of these commits, while preserving the software’s functionality, may significantly degrade performance, i.e., introducing performance regression.

Figure 1 shows a real-world performance regression issue from MySQL. The added code commit was written without much consideration for performance. When the code gets indirectly executed inside a loop, it can incur 80 times slowdown.

To provide motivating evidence, Figure 2 shows the releases containing performance regression issue(s) in a snapshot of the evo...
1.2 Performance Regression Testing Challenges

An effective way to combat performance regression is to employ systematic, continuous performance regression testing. This is widely advocated in academia [28, 57, 24, 45], open source community [45, 15, 36, 21] and industry [6].

Ideally, the testing should be carried out as comprehensively and intensively as possible, i.e., on every source commit basis. This can eliminate the “last-minute surprise” wherein performance issues are exposed too late to be fixed before release [16]. More importantly, this would avoid the tedious and lengthy diagnosis process to figure out which commit is responsible for the observed performance regression.

Unfortunately, despite the obvious benefits, real practices often cannot afford performance regression testing for every commit due to the combination of two factors: the high performance testing overhead and the rapid software evolution pace.

Performance testing is supposed to measure systems under representative and comprehensive workloads. It can take hours to days to complete even with a number of dedicated machines [3, 21, 23]. Table 1 lists the typical per run cost of several popular benchmarks. A comprehensive performance testing often includes not only these benchmarks but also load and stress testing suites. In a leading, public US data warehousing company that we collaborate with (anonymized as required), some internal test cases can take almost one week to just load the data.

Table 2: Estimated commit and performance testing frequency in popular software.

<table>
<thead>
<tr>
<th>Software</th>
<th>Avg. Rev. per Day</th>
<th>Regular Perf. Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>~ 6</td>
<td>every release [49]</td>
</tr>
<tr>
<td>Chrome</td>
<td>~ 140</td>
<td>every 4 rev.</td>
</tr>
<tr>
<td>Linux</td>
<td>~ 140</td>
<td>every week [21]</td>
</tr>
</tbody>
</table>

What’s more, performance testing results are subject to external factors such as caching and testing environment load. To minimize experimental errors, performance testing should be carried out in a clean environment for a period long enough and repeated several times until the performance becomes stable [35, 5, 41]. Consequently, performance testing guide often advocates to “never believe any test that runs for only a few seconds” [9, 46]. In this sense, performance testing is by nature time and resource consuming.

On the other hand, the increasingly favored rapid development methodologies such as agile development [34] are catalyzing software revision speed. As Table 2 shows, Chrome and Linux have more than 100 revisions per day merged into the code base. With the high revision rate and high testing cost, it is almost impractical to run comprehensive performance testing on every commit.

1.3 Current Practices

The above factors often cause performance regression testing frequency to be compromised. Testing is carried out on daily or per release basis (Table 2). When performance regression is exposed, developers have to spend extra efforts bisecting which commit among the recently committed changes causes the problem. For example, in diagnosing a Chrome performance regression issue [2] revealed during testing, there were six revisions included since last testing. The developers had to conduct regression testing on the problematic commit, which already took hours.

With performance test case prioritization, a technique as used in feature regression testing [51, 25], the testing frequency can be increased but at the cost of reduced comprehensiveness. This scheme, test cases are divided into multiple levels based on their overhead and ability to catch performance issues. Then lightweight test cases with high detection rate are run more frequently while costly tests are run infrequently. Our industry collaborator adopted this practice. Such prioritization is effective to capture easy-to-trigger performance regression. But the detection of complicated issues that are manifested only under the comprehensive test cases are delayed because of reduced comprehensiveness.

1.4 Our Contributions

Given that performance testing frequency and the commit speed are not in synchrony, it is important to fully utilize the testing cost. To this end, it is desired to devote testing on exactly the regressing commits and skip non-regressing commits. But existing practices as above all treat the testing target—code commits—as black-box, ignoring the valuable information in commit content that may be exploited to direct performance testing on the right target. Consequently, the testing is carried out blindly even for commits that are unlikely to introduce performance regression.
Table 3: Studied software.

<table>
<thead>
<tr>
<th>Software</th>
<th>Description</th>
<th>LOC</th>
<th># of Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>DBMS</td>
<td>1.2M</td>
<td>50</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>DBMS</td>
<td>651K</td>
<td>25</td>
</tr>
<tr>
<td>Chrome</td>
<td>Web browser</td>
<td>6.0M</td>
<td>25</td>
</tr>
</tbody>
</table>

Our work takes this complementary approach by leveraging the information in each code commit to help prioritize performance regression testing on risky commits. In particular, by conducting a lightweight, static performance risk analysis (abbr. PRA, hereafter) on source code change, we estimate the risk of this revision in introducing performance regression issues. Based on such estimation, we can prioritize performance testing to conduct more comprehensive tests for high-risk revisions while lowering the testing cost for low-risk ones.

For example, by statically analyzing the added code in Figure 1, we know it can be executed indirectly inside a loop. With static cost estimation and profile information, our PRA will recommend this commit to be tested heavily. On the other hand, for commits like adding few arithmetic operations in a cold path, we could suggest skipping testing or testing it lightly.

The major contributions of this paper are:

- We conduct an empirical study on 100 real-world performance regression issues from three widely used software to gain insights on the feasibility and approaches for testing target prioritization with commit performance risk analysis.
- To the best of our knowledge, we are the first to propose a white-box approach on prioritizing performance testing targets with code change performance risk analysis (PRA) to make performance regression testing more efficient.
- We implement a tool and release it in open source. Evaluation on the studied commits as well as 600 randomly selected new commits shows that based on the recommendation of our tool, developers just need to test only 14-22% of the 600 code commits and will be able to catch 87-95% of all performance risky commits.

2. UNDERSTANDING REAL WORLD PERFORMANCE REGRESSION ISSUES

To validate the feasibility of performance regression testing target prioritization and design an effective PRA, we conduct an empirical study on 100 randomly selected real-world performance regression issues from three widely used, open source software. The study focuses on understanding what kind of code changes would cause performance regression and how they impact performance.

Designing in this bottom-up manner may suffer from the overfitting problem. To address this concern, we also evaluate our proposed PRA on 600 randomly selected new code commits that are not used in the study described here.

2.1 Software Studied

Table 3 lists the details of studied software. We choose these software packages because they are performance critical, top-rated, widely used and industry-backed. Chrome is developed by Google; MySQL is now owned by Oracle; and PostgreSQL’s development team consists of employees from Red Hat. Besides, their long evolution history and well-maintained issue tracking systems provide us with many real-world performance regression issues.

2.2 Issue Collection Methodology

For each software, we first query the tracking system or mailing list with a set of broad performance-related keywords like “performance”, “hit”, “drop”, “slow”, “slower” on resolved issues. After getting this initial collection of issues, we manually go over each to prune out those issues unrelated to performance regression. Finally, we randomly sample 100 issues of which not only the fixes can be found, but also the responsible change set can be tracked.

2.3 Threats to Validity

Construct Validity Our study has high construct validity because all the issues we collect are indeed related to performance regression issues based on issue symptom description from the reporter and confirmation of issue fix from the developer.

Internal Validity There is potential selection bias in our study. We try to minimize it by first covering diverse categories and representative software. Except for Chrome which is landed in 2008, the other two software projects have more than 10 years of history. Additionally, the issues we study for each software are randomly sampled without favoring or ignoring particular type of performance regression issues. For each issue, we write a diagnosis report and have at least 2 inspectors agree on the understanding.

Another potential threat is unreported performance regression issues. It is difficult to measure quantitatively them. However, we believe that at least the reported issues are of importance.

External Validity Although we believe our study provides interesting findings on performance regression issues in studied representative software, they may not be generalized to other software projects beyond the specific scope this study was conducted. Thus they should be taken with limitations in mind.

Most importantly, we want to emphasize the primary purpose of this study is for ourselves to gain insights for designing PRA, instead of drawing any general conclusions. Therefore, the potential threats to validity in the study will only impair the efficacy of our design. Nevertheless, as evaluation demonstrates (§5), the PRA guided by this study is indeed effective, even for 600 new code commits that are not examined in our empirical study.

2.4 Code Change Categorization

To analyze what kind of code changes are likely to introduce performance regression, we develop a taxonomy. The key rationale is that program performance depends on how expensive an operation is and how many times the operation gets executed. Therefore, performance regression could be introduced by either more expensive operations or regular operations but executed many times in a critical path. As a starting point, we need to look into what and where are the code changes in their execution context. Moreover, we need to consider code changes that indirectly impact expensive operations or critical path. For example, a code change that modifies a variable controlling the loop iteration count in a critical path may have a big impact on performance. We further zoom into each of the three perspective and divide it into subcategories. The subcategories may not exclusive. For example, in the where perspective, a change could lie in both a loop and a primitive function. We choose the first applicable subcategory in top to bottom order.

In our study, we consider only code changes that are the culprits for performance regression, ignoring innocent changes in the same commit. Also we focus on changes that impact the performance of existing functionalities, ignoring changes for independent, new functionalities that do not interact with existing ones. The result based on this categorization methodology is presented in Table 4.

2.4.1 Where a Change Takes Place

Program scopes such as loop or primitive function are performance sensitive places. It is because these places can be executed or called many times and magnify any overhead added inside them.
Hence we first categorize all the problematic changes based on the scopes that they lie in.

We also need to be context sensitive and consider call paths. For example in Figure 1, although the added expensive function call `find_table_for_trigger` is not inside loop in the static context, if the expensive path call is executed, it will be dynamically executed in a tight loop (loop that iterates many times). Therefore, when we design PRA, we need to be context sensitive and examine possible call paths.

Table 4 shows more than half of the problematic changes are located in performance sensitive scopes. However, using change scope as the sole indicator of performance regression can miss some performance killers that can influence critical paths via data and control flow. That is why there are a significant number of regression issues in the Others subcategory in Table 4. For this reason, we further categorize the change content.

**Implication:** PRA should pay attention to common performance critical places such as loop and primitive function. But using only place as criteria can incur high analysis inaccuracy.

### 2.4.2 What a Change Modifies

Intuitively, for a change set to introduce performance regression, it needs to add high overhead (computation or I/O). Such influence can happen in two ways: one case is that the change directly adds significant cost into hot path; another case is that the change modifies some “control” variables which result in an expensive code region to iterate more times or take a longer execution path.

The former case is straightforward (especially if our PRA can be context sensitive), whereas the latter would require us to consider some special variables and their control and data dependencies. We refer to the variable of which different values can result in dramatically different performance as *performance critical variable* (e.g., loop iteration count, database table index variable as in Figure 3), and the condition that controls whether a critical path will be taken in a branch as *performance sensitive condition* (e.g., the branch condition in function `create_sort_index` in Figure 5).

In the study, we obtain information regarding whether operations are expensive or performance critical by reading the bug reports and consulting with developers. In our implementation of PRA, we obtain such information from static analysis and profiling.

As Table 4 shows, expensive function calls is the most common problematic change content. But other types of changes such as modifying the performance critical variable and performance sensitive condition also cause a significant number of performance regression issues in the dataset.

**Implication:** PRA should also account for the code change content to identify costly instructions and performance critical variables or sensitive conditions.

### 2.4.3 How a Change Impacts Performance

As explained above, not all problematic changes affect performance in a direct way. A change may propagate via data and control flow that eventually causes a performance problem in its forward slice. For example, in Figure 5 although the change itself is not expensive, it modifies the function return value, which later determines whether an expensive `filesort` is called.

Thus we divide all the perilous changes into two categories based on how they impact performance: (1) changes that directly degrade performance in its execution scope; (2) changes that indirectly cause performance problem later in other code regions.

Table 4 shows the majority of the issues belong to the first category. But there are also a significant number of issues impacting performance indirectly. For these issues, the table also lists the program constructs through which the change propagates its effect to critical paths and thereby leads to performance degradation. Most of them are through function return values.

**Implication:** PRA should follow the control and data flow to factor in the indirect effect of a change. The analysis should be inter-procedural and context-sensitive.

### 2.5 Case Studies

We now go through three performance regression issues from MySQL as case studies.

The issue in Figure 4 is introduced when patching a functionality bug in MySQL Cluster. This commit changes the system call `clock_gettime` argument to use `CLOCK_MONOTONIC` instead of real-time clock to prevent potential hang in certain scenarios. But in some platform like Solaris, `clock_gettime` call is expensive when using `CLOCK_MONOTONIC`. This extra overhead is further amplified when it is called in a tight loop.
The objective of PRA is to examine source code commit content and determine whether the commit is likely to introduce performance regression. Therefore, PRA is a white-box approach. But PRA is not meant to replace performance regression testing. On the contrary, its main consumer is performance regression testing.

3. PERFORMANCE RISK ANALYSIS

3.1 Overview

The objective of PRA is to examine source code commit content and determine whether the commit is likely to introduce performance regression. Therefore, PRA is a white-box approach. But PRA is not meant to replace performance regression testing. On the contrary, its main consumer is performance regression testing. It recommends risky commits to performance regression testing to test comprehensively and suggests skipping low-risk commits.

This role relaxes the safety and soundness requirement for PRA. In the extreme case, if PRA recommends every commit to be tested, it is the same situation as testing without PRA; if PRA recommends to skip a commit that is actually risky, it is similar as conducting infrequent such as per-release testing. That said, PRA should aim to flag the commits’ risk accurately to be able to truly improve performance regression testing efficiency.

3.2 PRA Design

PRA is essentially static performance risk estimation of code changes. The challenge is how to reason about performance impact without actually running the software. Guided by the real world issue study, the performance impact of a code change depends on the cost of change operations (whether it is expensive or not) and the frequency of its execution (whether it lies in hot path or not).

Therefore, we design PRA as follows. First, we use a cost model to estimate the expensiveness of a change instruction. If the change touches performance sensitive conditional expression, it is also considered expensive. Then, we estimate the frequency of the change instruction. With the two estimations, we index them into a risk matrix to assess the risk level. The risk levels for the entire patch can then be aggregated into a single risk score.

3.2.1 Cost Modeling

To establish a systematic and efficient estimation regarding the expensiveness information, a static cost model for different operations is necessary. The cost modeling will determine whether a change instruction is expensive or not.

Since the purpose of the modeling emphasizes on relative cost rather than absolute cost, we express cost in abstract unit, denoted as $\delta$, instead of actual CPU cycles. With simple architecture model and heuristics, we build a basic cost table for different instructions based on its type and operands. For example, add instruction has cost $\delta$, multiply has cost $4\delta$ and call instruction has the cost equal to calling convention overhead plus the callee’s cost.

A basic block’s cost is a sum of the cost of those instructions that live inside the basic block. For control flows, we adopt worst case analysis. For example, the cost of a function or a loop body is the maximum cost among the static paths.

Next, we assume if an operation has cost $\delta$, and it’s executed 10 times, the aggregated cost is $10\delta$. While this assumption doesn’t account for factors such as compiler optimization, it is a good start for the purpose of risk assessment. Then for a loop, if its trip count (maximum number of iterations) can be statically inferred, we multiply the loop body’s cost by the trip count as the loop’s cost. Otherwise, the loop is considered to be potentially expensive.
With this model, we can obtain cost in terms of $\delta$, whereas PRA needs to gauge whether the cost is expensive or not. We use thresholds to convert the cost into level. Such thresholds can be configured or computed automatically by running the cost model on the whole program to obtain cost distribution for functions and basic blocks. Then the cost of a change is ranked in the distribution to convert to expensiveness. Besides this static model, we also allow users to add dynamic profile information or domain knowledge to make PRA more accurate. But this is only optional for PRA.

The above modeling mainly deals with add code change type. Code change can also be deleting or replacing statements. For delete type changes, we can offset their direct cost from the total cost. In current implementation, we do not want to make PRA aggressive that miss potential performance regression issues due to inaccurate offsetting. Therefore, the cost of a delete change is by default 0. The theoretical cost for a replace type change would be $new\cdot cost - old\cdot cost$. But for similar reason, the cost of a replace change is the cost of new program elements. The exception is: if a delete or replace change touches performance sensitive variables or conditions, the change cost is directly assigned to be expensive.

### 3.2.2 Performance Sensitive Condition/Variable

Condition expressions need special attention. Conditional branch instruction itself is rarely costly. But as seen from the real world study, a condition can significantly influence whether an expensive path will be taken or not. We define a branch condition to be performance sensitive if its intra-procedural paths (successors) have dramatic cost difference, which is estimated using above cost model. For example, the branch condition for the first if statement in function create_sort_index in Figure[5] is a performance sensitive condition. Change affects such performance sensitive condition is considered expensive.

For performance critical variable, we currently only considers variable that can affect loop termination condition. We leave systematically identifying performance critical variable such as the table index variable in future work.

### 3.2.3 Frequency Estimation

In addition to estimating the cost of change content, we should also analyze whether the change lies in hot path. To do this, we first analyze the intra-procedural scope that a change lies in. If the change is enclosed in any loop and the trip count of this loop including all its parent loops can be statically determined, the execution frequency of this change instruction is estimated by the product of these trip counts. Otherwise, if any of enclosing loop has non-determined trip count, it is considered to be possibly executed frequently. Similarly code change that lies in recursive functions is also assessed to be potentially frequently executed.

Next, we examine the call path backward that could potentially reach the function where the change is located and perform similar frequency estimation for each call site. In implementation, the level of call path length is bounded. Given the estimation, we conclude if a change lies in the context that may be frequently executed using a ranking of the frequency count.

### 3.2.4 Risk Matrix

Combining the above two pieces of information, we use a risk matrix (Table 5) to assess the risk. Such matrix can be extended to express more expensiveness/frequency categories and risk levels. The output of PRA is therefore the risk level distributions. We also calculate a simple risk score based on the distribution:

$$
\text{Risk Score} = N_{\text{extreme}} \times 100 + N_{\text{high}} \times 10 + N_{\text{moderate}} \times \frac{1}{100} + N_{\text{low}} \times \frac{1}{1000}
$$

Practitioners can then define testing target selection criteria against the risk level distribution or summary score. For example, commits with $N_{\text{extreme}} >= 5$ or whose risk score exceeds 300 require comprehensive testing. Moreover, in choosing the criteria in performance testing to judge regression, for different software, these criteria may also require some initial tuning.

### 3.2.5 Indirect Risk

As Section[2] shows, in addition to direct performance impact, there are also a number of risk commits that indirectly affect performance via data flow and control flow to their forward slice, the program subset that may be affected by a given program point. Therefore, we extend the basic PRA with the ability to analyze such cases, named PRA-Slicing. We use static program slicing techniques[5] to compute a forward slice of the change set. Then we check if any element in the slice is performance sensitive or not using the logic as described in 3.2.2. The final risk level is only assigned once to change instruction instead of its slice. The slicing is interprocedural and bounded to a limited depth. While PRA-Slicing can catch more complicated risky commits, it may also recommend excessive commits due to the imprecisions in the slicing. Moreover, computing precise slice can be expensive. Consequently, PRA-Slicing remains an extension to the main PRA analysis and is disabled by default.

### 4. IMPLEMENTATION

We develop a tool called PerfScope that implements the proposed PRA on top of the LLVM compiler infrastructure[19]. The tool is released in open source at [http://cseweb.ucsd.edu/~peh003/perfscope](http://cseweb.ucsd.edu/~peh003/perfscope).

### 4.1 Architecture

We first briefly describe the architecture of PerfScope. It consists of five components (Figure[6]).

Parser parses patch information regarding the changed files, lines and types (add, delete, change). For the change type, unified diff

<table>
<thead>
<tr>
<th>Expensiveness</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expensive</td>
<td>Frequent</td>
</tr>
<tr>
<td>Normal</td>
<td>Normal</td>
</tr>
<tr>
<td>Minor</td>
<td>Rare</td>
</tr>
</tbody>
</table>

Table 5: Risk matrix of a change’s expensiveness and frequency

- **Expensiveness**: New Cost $\delta$
- **Frequency**: Change is the cost of new program elements.
- **Exception**: If a delete or replace change touches performance sensitive variables.

Figure 6: Architecture of our PRA implementation, PerfScope
file contains add and delete but no replace. The parser will pair delete with add and reduce them to replace.

Mapper extracts debug information of a program, build search trees and map given lines to the program constructs, if any, which are fed to the filter.

Filter prunes out insignificant change such as stylistic change or renaming. If all changes in the entire commit are filtered, the commit will be considered trivial and not fed to the analyzer.

Profile database (optional) allows users to incorporate profile information and domain knowledge that may make PRA more accurate. Such profile can be practically incorporated because for large software, the cost of many functions doesn’t change very frequently. Therefore, the profile does not need frequent update.

But note that, PRA already provides a cost model ($\frac{1}{3}$). The profile information is only optional. Interestingly, we obtained the profiles for several software using a popular low-overhead, system-wide profiler—OProfile [8], and found the expensive function list has a large portion of overlap with the list computed by PRA.

Performance risk analyzer runs PRA or PRA-Slicing on the language constructs remained after filtering. It computes the risk level (e.g., extreme, high, moderate) of a change using a risk matrix like in Table 5. The output of the analyzer is a risk level distribution from all changes in the commit and a summary risk score. The two results can be used by performance testing practitioners to determine the performance testing strategy for the commit.

4.2 Challenges

There are two key challenges in implementing PerfScope.

Mapping: Raw information in patch files produced by standard tools like diff are language agnostic and in line granularity. But PRA works on language constructs. For example, it needs to know the change on line 10 corresponds to an if statement.

LLVM provides debugging information such as line number, file path attached in instructions. We use this information to build a search tree. We first find the compile unit for the changed source files, then match the top level constructs such as functions inside that unit and finally the corresponding instruction(s).

Filtering: There are changes on non-source files such as the documentation, test cases. It is unlikely for them to introduce performance regression. We predetermine a set of source code suffixes, e.g., .c, .cpp, to filter non-source changes. We also prune changes on source files that essentially do not alter the program (e.g., add comments, remove variables). We only perform safe, simple checking instead of trying to determine general equivalence of two programs, which is undecidable.

Algorithm 1: Determine if a change is trivial

Input: change C, source code suffix set Suffixes, before-revision program OldP, after-revision program NewP

Output: True if C is trivial, False otherwise

1. if C.file suffix $\notin$ Suffixes then
   2. return True
   3. end if

4. /* map: get instructions at a line in a program */
5. new_instrs ← map(C.new_line, NewP)
6. old_instrs ← map(C.old_line, OldP)
7. if new_instrs.empty and old_instrs.empty then
   8. /* change only comments, spaces, etc */
   9. return True
   10. end if

11. if new_instrs.size $\neq$ old_instrs.size then
   12. /* change added or deleted instructions */
   13. return False
   14. end if

15. for i ← 1 to new_instrs.size do
   16. /* diff: compare two instructions’ opcodes and operands */
   17. if diff(old_instrs[i], new_instrs[i]) = True then
   18. return False
   19. end if
   20. end for
   21. return True

5. EVALUATION

This section evaluates the effectiveness and efficiency of our PRA implementation, PerfScope. It consists of six parts. First, we test whether PerfScope is able to report these problematic commits from our real-world issue study. Second, to address the overfitting concern, we also evaluate PRA using 600 new commits from both studied software and unstudied ones. Third, we evaluate PRA-Slicing extension and also compare our design with random test target selection. Fourth, we estimate the practical testing cost savings with PRA. Fifth, we show the sensitivity of parameters used in the experiment. The last part shows the overhead of PerfScope.

5.1 Subject Software

Six large-scale (up to millions of lines of code), popular open-source software are used as our subjects. They range from database system (MySQL, PostgreSQL), compiler (GCC), web (caching) server (Apache, Squid) and JavaScript Engine (V8, used in Chrome).

5.2 Methodology

Table 6: Subject Software.

<table>
<thead>
<tr>
<th>Software</th>
<th>LOC</th>
<th>Studied?</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>1.2M</td>
<td>Yes</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>651K</td>
<td>Yes</td>
</tr>
<tr>
<td>Apache httpd</td>
<td>227K</td>
<td>No</td>
</tr>
<tr>
<td>Squid</td>
<td>751K</td>
<td>No</td>
</tr>
<tr>
<td>GCC</td>
<td>4.6M</td>
<td>No</td>
</tr>
<tr>
<td>V8</td>
<td>680K</td>
<td>No</td>
</tr>
</tbody>
</table>

Among them, GCC, Apache, Squid and V8 are not used in our real-world study. Table 6 summarizes the details.

5.2.1 Ground Truth for New Commits

For the studied commits, we already know they caused performance regressions. But for the new commits, since they are taken from recent code repository, few feedback on them exists. We need to get the ground truth for each commit with respect to whether it may introduce performance regression. Therefore, we run multiple standard, moderately intensive performance benchmarks on each compiled revision of subject software. These benchmarks are often used internally and in user-reported regression cases. Each benchmarking is run multiple times.

Table 7: Benchmarks and regression thresholds used for subject software. | Benchmarks | Threshold |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>DBT2, SysBench, sql-bench</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>DBT2, SysBench, pgbench</td>
</tr>
<tr>
<td>Apache httpd</td>
<td>SPECweb2005, autobench, ab</td>
</tr>
<tr>
<td>Squid</td>
<td>Web Polygraph, autobench</td>
</tr>
<tr>
<td>GCC</td>
<td>CP2K, SPEC CPU2006</td>
</tr>
<tr>
<td>V8</td>
<td>Octane, SunSpider</td>
</tr>
</tbody>
</table>
Table 8: Coverage of studied problematic commits.

<table>
<thead>
<tr>
<th>Software</th>
<th>Buggy</th>
<th>PRA (Ratio)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>29</td>
<td>30 (93%)</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>25</td>
<td>23 (92%)</td>
</tr>
<tr>
<td>Total</td>
<td>64</td>
<td>59 (92%)</td>
</tr>
</tbody>
</table>

With the benchmarking results, we can compute the performance deviation of a commit from the previous commit. Then if the deviation exceeds certain threshold, the commit introduces performance regression. Therefore what thresholds to use for judging performance regression is crucial. However, there is inevitable bias in choosing the criteria because performance regression is a fundamentally subjective definition. Different practitioners and software use different judgment. For example, in our real-world study, the user-reported performance regression issues contain a variety of criteria even for the same software.

To reduce the inevitable selection bias, we refer to the common practices. In particular, we sample each software’s issue tracker, code commit comment and developer posts to see what degree of performance deviation starts to attract practitioners’ attention. We then choose the minimum as our criteria. Table 7 lists the benchmarks and thresholds we use for each software.

Sometimes the commit causing repeatable regression (i.e., the regression is observed consistently when the testing is exercised several times) may not be a performance bug but rather an expected performance behavior (e.g., a patch adding authentication step to performance regression testing. The recommendation is considered to be useful.

5.2.2 Setup

The benchmarking is carried out on three dedicated machines, each with Intel i7 Quad Core CPU 3.40 GHz, 16GB RAM. The performance overhead measurement is run on the same machines. Additionally, as PRA outputs the distribution of the change set’s risk levels and a score, we need a criterion for recommending performance regression testing. We set it to be if risk score (Equation 1) is larger than 200. \[ \text{Risk Score} = \text{Risk Level} \times \text{Risk Score} \]

5.3 Evaluation on Studied Commits

Table 6 presents the coverage of the studied problematic commits. 11 commits from MySQL issue study are omitted because these changes are for source languages other than C/C++, which our tool currently supports. Chrome is not evaluated because the compiler our analysis bases on, LLVM Clang, cannot reliably compile Chrome into whole-program LLVM bytecode files for analysis rather than because of our analysis.

As the table shows, PRA can report majority (92%) of these regressing commits. Although the basic PRA does not implement slicing, it still can capture cases in the “Indirect” category in Table 3. This is because, the categorization only focuses on the root cause. It can be the case that a change impacts performance indirectly but happens to lie in a loop.

A few cases are missed by PRA either because of long propagation impact or complicated domain knowledge. Listing 1 is such an example. More detailed profile information and deeper analysis are needed to be able to alert on them.

Listing 1 Commit not alarmed by PRA

```c
void init_read_record(READ_RECORD *info, THD *thd)
{
    /* The patch sets mmap flag, which later causes a function pointer to be changed to mmap. */
    + if (table->s->tmp_table == TMP_TABLE &...)
    + VOID(table->file->extra(HA_EXTRA_MMAP));
```

5.4 Evaluation on New Commits

Since our PRA design is guided by a real-world issue study, it might be tailored for these examined buggy commits. Therefore, we also evaluate the tool on 600 new commits from both studied and unstudied software. Table 9 presents the result.

The After Filtering column is the number of commits remaining after pruning. The filtered commits (by our tool) either only change non-source files or only have insignificant changes on source files. Interestingly, filtering already reduces a significant number of commits not worth consideration for performance regression testing. For example, in Apache, more than one third of the commits are just updating documentation files or code styles. We manually checked the filtered commits are indeed trivial.

Our tool PerfScope can successfully reduce at least 81% of the 450 testing candidates (86% if no filtering is conducted in existing testing) and alarm 87% of the new risky commits. In other words, with our tool, developers only now need to test 19% of the original 450 commits and still be able to alert 87% of the risky commits. This means our PRA design can significantly reduce performance testing overhead while preserving relatively high coverage.

From the table, we can also read the number of commits that are reported by PerfScope but not confirmed by our benchmarking from Rec. Commits — (Risky Commits — Miss). However, they should not be interpreted false alarms for two reasons. First, PerfScope is not a bug detection tool but only to reduce testing overhead. These “additionally” recommended testing targets need to be tested anyway in the original performance testing scheme without using PRA. Second, the Risky Commits in the table are lower bounds because of the limitation of our benchmarking. Therefore, some of these additional commits might turn out to be indeed risky if tested more comprehensively.

5.5 Extension and Alternative Solution

PerfScope also implements an extension to PRA: PRA-Slicing (§ 5.2.3). In addition to the basic analysis, PRA-Slicing also performs forward slicing of each change in the commit and checks if any element in the slice is performance sensitive or not.

Table 7 also shows the evaluation of PRA-Slicing. PRA-Slicing performs deeper analysis and as a result has higher coverage (95%) but at the cost of lower reduction percentage (78%).

A simple alternative to PRA is random test target selection. The probability for this approach to achieve the same or better result as our tool in Table 6 is only $1.2 \times 10^{-65}$ (calculated using script 1).

5.6 Practical Cost Saving

The objective of PRA is to reduce the testing target set to the risky commits for performance regression testing. Previous sections mainly evaluate the number of reduced testing target. How much does the reduction translate to actual testing cost saving? It depends on how comprehensive the original testing is carried out.

http://ideone.com/6d70fJ
Table 9: Evaluation of PerfScope on new commits. *: the filtering is done automatically by our tool with Algorithm 1. For reduction rate and testing saving, larger number is based on the 600 commits; smaller number is based on the 450 commits.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>100</td>
<td>10</td>
<td>73</td>
<td>19 (74–81%)</td>
<td>2</td>
<td>324–486</td>
<td>22 (70–78%)</td>
<td>1</td>
<td>306–468</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>100</td>
<td>6</td>
<td>76</td>
<td>12 (84–88%)</td>
<td>0</td>
<td>384–528</td>
<td>16 (79–84%)</td>
<td>0</td>
<td>360–504</td>
</tr>
<tr>
<td>GCC</td>
<td>100</td>
<td>6</td>
<td>76</td>
<td>18 (76–82%)</td>
<td>1</td>
<td>870–1230</td>
<td>19 (75–81%)</td>
<td>0</td>
<td>855–1215</td>
</tr>
<tr>
<td>V8</td>
<td>100</td>
<td>7</td>
<td>85</td>
<td>13 (85–87%)</td>
<td>2</td>
<td>3–4</td>
<td>17 (80–83%)</td>
<td>1</td>
<td>3–4</td>
</tr>
<tr>
<td>Apache</td>
<td>100</td>
<td>5</td>
<td>60</td>
<td>11 (82–89%)</td>
<td>0</td>
<td>74–134</td>
<td>12 (80–88%)</td>
<td>0</td>
<td>72–132</td>
</tr>
<tr>
<td>Squid</td>
<td>100</td>
<td>6</td>
<td>80</td>
<td>12 (85–88%)</td>
<td>0</td>
<td>100–204–264</td>
<td>14 (83–86%)</td>
<td>0</td>
<td>198–258</td>
</tr>
<tr>
<td>Total</td>
<td>600</td>
<td>39</td>
<td>450</td>
<td>85 (81–86%)</td>
<td>5</td>
<td>(87%)</td>
<td>1859–2646</td>
<td>100 (78–83%)</td>
<td>2 (95%)</td>
</tr>
</tbody>
</table>

Figure 7: Sensitivity of PerfScope’s reduction rate and coverage rate of the new commits evaluation to the recommendation criterion ($\geq \text{risk
data\_threshold}$).

In our experiment, we run the benchmarks using recommended settings from popular performance testing guides or the tools.

Therefore, we calculate the expected testing cost saving according to our setup. As we run multiple benchmarks for each software, we use the average running cost as the per iteration testing cost. The per commit cost is calculated by multiplying the per iteration cost with 3 iterations as used in our experiment. In specifics, for MySQL and PostgreSQL, the per commit testing cost is 6 hours; for GCC, it is 15 hours; for V8, it is 3 minutes; for Apache, it is 1.5 hours; for Squid, it is 3 hours.

Table 9 lists the result. The lower saving is calculated assuming the original testing already skips all non-essential commits. The higher saving assumes no filtering is conducted. PerfScope is most useful for software whose original performance regression testing cost is big. For example, for GCC, the saving can be almost two months. However, for V8, the saving is limited because the benchmark we use only takes minutes to test one revision. Admittedly, for software whose performance testing overhead is very small, PerfScope does not save much. Directly running performance testing on every commit is then a better option. One possible usage scenario for this type of software is to run PerfScope as “cross reference” with the performance testing result.

5.7 Sensitivity

Since the output of PRA is risk level distribution and a summary score, a criterion based on the output is necessary to decide whether to recommend the given code commit to be tested. The evaluation in previous sections used the 200 risk score threshold as the criteria.

Figure 7 measures the reduction and coverage sensitivity to the risk score threshold. In general, reduction rate increases as the risk score criteria increases because higher threshold would result in fewer number of commits to be recommended and thus achieving higher reduction; in contrast, coverage decreases as the criteria increases. Such variety means, like performance regression testing result judgment criteria, the selection criteria requires initial tuning. But as seen from the figure, the sensitivity in certain threshold ranges is small because problematic and regular commits usually have quite different risk level distribution.

5.8 Performance

As the primary goal of PRA is to reduce performance testing overhead through testing target selection, the analysis itself shouldn’t become a new bottleneck.

Table 10 shows that in the evaluated software, PerfScope’s average execution time is within 2 minutes for PRA and 6 minutes for PRA-Slicing. The “Loading Module” breakdown is the time to load LLVM object files for analysis using LLVM API `parseRFile`. It is a constant cost and occupies a large portion of the total time for both PRA and PRA-Slicing.

6. RELATED WORK

Performance Regression Testing: There are case studies and research effort on performance regression testing in software systems [21, 25, 50, 17, 59]. To name a few, [21] details the Linux kernel performance testing project to catch kernel performance regression issues. [56] shares the experience in automating regression benchmarking for the Mono project. [17] proposes a model-based performance testing framework to generate appropriate workloads. [50] offers a learning-based performance testing framework that automatically selects test input data based on learned rules. [61] uses symbolic execution to generate load test suites that expose program’s diverse resource consumption behaviors.

These efforts focus building better performance regression testing infrastructure and test cases. Our work assumes the existence of good performance testing infrastructure and test cases, and improves the testing efficiency by prioritizing testing target.

Performance Bug Detection and Analysis: A wealth of literature exists on performance analysis with regard to performance debug-
Reducing testing target set. The analysis is focused on code changes and the relative performance bound for the entire program. Instead, regression testing overhead via testing target prioritization. Test case reduction selects a subset of test cases in the test suite based on test case property and code modification information. Test suite reduction works on removing redundancy in test suite. Test case prioritization orders test case execution in a way to meet time constraints in hope to detect fault faster.

Different from these work, our goal is to reduce performance regression testing overhead via testing target prioritization. In functional regression testing, work that also analyzes code modifications focuses on code coverage. But in the context of performance, more important is information such as whether an operation is expensive or lies in critical path. The analysis we propose is specifically for assessing commits’ performance risk.

There are also practices on improving performance regression testing efficiency mainly through hierarchical test case organization. We differ from them in that we take a white-box approach to prioritize test target by analyzing commit content.

Impact Analysis: There is fruitful work on software change impact analysis techniques to compute the set of program elements that may be affected by the change (impact set). They can be broadly divided into three categories: static analysis based, dynamic execution based, and history-based. Our proposed method is inspired by these work. The key difference is that we focus on the performance risk implication of change, instead of the impact set. PRA assesses the risk of a code change to introduce performance regression in addition to the impact set.

Additionally, many impact analysis work focuses on function level. But PRA needs to examine more fine-grained statement level for detailed analysis. This not only poses challenges in the analysis but also on mapping from textual changes to the corresponding programming constructs.

Worst-Case Execution Time Analysis: Analyzing the worst case execution time (WCET) of a task to be executed on a specific hardware is a necessary process for reliable real-time system because of its stringent timing constraints. Our PRA is similar as the static approach in WCET analysis. For example, both PRA and WCET analysis needs to use control flow information and bound calculation to determine the worst case execution path.

However, WCET analysis mainly applies to real-time systems as they have restricted form of programming (e.g., no recursion allowed). PRA works on regular performance-critical software written in standard C/C++ that supports generic programming constructs. The biggest size of tasks analyzed by WCET analysis tool is around 50K LOC. PRA can scale to millions of LOC for regular software.

More importantly, PRA does not aim to predict the absolute performance bound for the entire program. Instead, PRA calculates the relative performance risk introduced by given code change to reduce testing target set. The analysis is focused in code change scopes. This makes PRA light-weight enough to fit in the performance testing cycle. In contrast, WCET analysis needs to obtain a safe bound as a worst-case guarantee for the entire program. It therefore requires careful modeling of underlying architectural behaviors (e.g., branch prediction) and often requires user annotations (e.g., loop bounds, flow facts), which is very expensive to perform on a per-commit basis to be used by performance testing.

7. LIMITATIONS AND DISCUSSIONS

There are limitations in the current implementation that we are considering for future work.

First, our PRA is designed to recommend straightforward performance regression issues. While this makes the analysis lightweight, the analysis may not accurately assess the risk of sophisticated performance regression issues such as resource contention, caching effect. Second, although our cost model is generic for both computation and I/O, detailed modeling and profiling are needed if I/O behavior is of particular interest. Therefore our current modeling has limited applicability to software like OS kernel. Also our model does not apply to networked software. Third, since for a commit that is considered to be potentially risky, PRA knows the program points that are risky. With the risky program points and test cases’ coverage information, we are extending the analysis to not only select testing target but also recommend which test case may potentially expose the performance issue in the risky version.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a new approach, performance risk analysis (PRA), to improve performance regression testing efficiency through testing target prioritization. It analyzes the risk of a given code commit in introducing performance regression. To gain deep understanding of perilous commits’ code characteristics, we conduct a study on 100 randomly sampled real-world performance regression issues from three large popular software. Based on the insights from the study, we propose a PRA design and implement a tool, PerfScope. Evaluation on the studied problematic commits shows PerfScope can successfully recommend 92% of them for testing. We also evaluate the tool on 600 new commits that are not studied. PerfScope significantly reduces the testing overhead by recommending only 14-22% of the 600 commits and is still able to cover 87-95% of the risky commits. Experiment demonstrates the analysis is lightweight. The source code of PerfScope is released at http://cseweb.ucsd.edu/~peh003/perfscope.

9. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for the insightful comments and suggestions. We are very grateful to Ding Yuan, Soyeon Park, Weiwei Xiong for their constructive feedback. Tianyin Xu generously provided dedicated machines for us to run experiments. Rui Zhang lifted the morale with constant encouragement. This work is supported by NSF CNS-1017784, and NSF CNS-1321006.

10. REFERENCES


