PerfRanker: Prioritization of Performance Regression Tests for Collection-Intensive Software

Shaikh Mostafa
shiakh.mostafa@utsa.edu
University of Texas at San Antonio
TX, USA

Xiaoyin Wang
xiaoyin.wang@utsa.edu
University of Texas at San Antonio
TX, USA

Tao Xie
taoxie@illinois.edu
University of Illinois at Urbana-Champaign
IL, USA

ABSTRACT

Regression performance testing is an important but time/resource-consuming process. Developers need to detect performance regressions as early as possible to reduce their negative impact and fixing cost. However, conducting regression performance testing frequently (e.g., after each commit) is prohibitively expensive. To address this issue, in this paper, we propose PerfRanker, the first approach to prioritizing test cases in performance regression testing for collection-intensive software, a common type of modern software heavily using collections. Our test prioritization is based on performance impact analysis that estimates the performance impact of a given code revision on a given test execution. The evaluation shows that our approach can cover top 3 test cases whose performance is most affected within top 30% to 37% prioritized test cases, in contrast to top 65% to 79% by three baseline approaches.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Performance, Regression Testing, Test Prioritization

ACM Reference format:
Shaikh Mostafa, Xiaoyin Wang, and Tao Xie. 2017. PerfRanker: Prioritization of Performance Regression Tests for Collection-Intensive Software. In *Proceedings of 26th International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 12 pages.
https://doi.org/10.1145/3092703.3092725

1 INTRODUCTION

During software evolution, frequent code changes, often including problematic changes, may degrade software performance. For example, a study [24] found that upgrading from MySQL 4.1 to 5.0 caused the loading time of the same web page to increase from 1 second to 20 seconds in a production e-commerce website. Even small performance degradation may result in severe consequence. For example, Google could lose 20% traffic due to an increase of 500ms latency [40]. Amazon could have 1% decrease in sales due to a 100ms delay in page rendering [56].

Developers can apply systematic, continuous performance regression testing to reveal such performance regressions in early stages [14, 18, 27, 42, 60]. But due to its high overhead, performance regression testing is expensive to conduct frequently. The typical execution cost of popular performance benchmarks varies from tens of minutes to tens of hours [24], so it is impractical to run all performance test cases for each code commit. Recently, PerfScope [24] was proposed to predict whether a code commit may significantly affect software performance. Specifically, PerfScope extracts various features from the original version and the code commit, and trains a classification model for prediction. Although PerfScope helps reduce code commits for performance regression testing, its evaluation shows that a non-trivial proportion of code commits still require performance testing; thus, there is still a strong need of reducing the cost of conducting performance regression testing on a code commit, even after applying PerfScope.

To address such strong need, developers shall prioritize performance test cases on a code commit for three main reasons. First, there can be high cost to execute all performance test cases on a code commit for large systems in practice. Second, as reported in a previous industrial study [61] and our study in Section 2, various random factors may affect the observed execution time, so it typically requires a large number of repetitive executions to confirm a performance regression. Therefore, with prioritized test cases, developers can better distribute testing resources (i.e., do more executions on test cases likely to trigger performance regressions). Third, a code commit may accelerate some test cases while slowing down others. Developers often need to understand the performance of their software under different scenarios, while a coarse-grained commit-level technique is not helpful on this requirement.

To develop an effective test-prioritization solution for performance regression testing, we focus on *collection-intensive software*, an important type of modern software whose execution time is heavily spent on loading, manipulating, and writing collections of data. Collections are widely used in software for scalable data storage and processing, and thus collection-intensive software is very common. Examples include libraries for data structures, text formating and parsing, mathematics, image processing, etc. Also, collection-intensive software is often used as components in complex systems. Moreover, a recent study [25] shows that a large portion of performance bugs are related to loops, which are often used to iterate through collections. Our statistics show that 89% and 77% of loops iterate through collections for our two subjects Xalan and Apache Commons Math, respectively.

ISSTA'17, July 2017, Santa Barbara, CA, USA

ACM ISBN 978-1-4503-5076-1/17/07…$15.00
https://doi.org/10.1145/3092703.3092725

For collection-intensive software, a straightforward approach to prioritizing performance test cases on a code commit would be measuring collection iterations (e.g., loops) impacted by the code commit and executed by each test case. However, such an approach may not be precise enough to differentiate test cases in the presence of newly added iterations, manipulations, and processing of collections, as well as their effect on existing collection iterations. Consider the simplified code example from Xalan in Listing 1. The code commit involves a new loop, and its location may or may not be at the hot spot for all or most test cases. Therefore, its impact on different test cases may largely depend on the different iteration counts of the added loop, the side effect of changing variable `list`, and the operations in the loop. Since $Loop_B$ depends on a collection variable `limits`, which further depends on $Loop_A$, we can infer the test-case-specific iteration count of $Loop_B$ from that of $Loop_A$; such iteration count can be acquired by profiling the base version for all test cases. Furthermore, we can infer the effect of adding `list.add(...)` on `list` with the iteration count of $Loop_B$, and update the iteration count of loops dependent on `list`. Moreover, we can enhance the estimation precision by using test-case-specific execution time of operations (e.g., `new Arc(...)`).

```
1   while(i <= m_size){ //Loop A
2   limits.add(new Limit(...))
3   ...
4   }
5   ...
6   + Collections.sort(limits);
7   + for (int i = 0; i < limits.size()-1; i++) { //Loop B
8   +   list.add(new Arc(limits.get(i), ...));
9   + }
```

Listing 1: Collection Loop Correlation

These observations inspire us for three main insights to effectively model a code commit and its effect on existing collections and their iterations. First, collection sizes (e.g., `limits.size()`) and loop-iteration counts (e.g., $Loop_B$) can often be correlated, so collection sizes can be inferred from loop-iteration numbers and vice versa. Also, collection variables (e.g., `limits`) can be used as bridges to infer iteration counts of new loops (e.g., $Loop_B$) from existing loops (e.g., $Loop_A$). Second, collection manipulations (e.g., `list.add(...)`) are often inside loops, so the size of collections referred by collection variables (e.g. `list`) can be estimated from loop-iteration counts (e.g., $Loop_B$). Third, due to the large number of elements in collections, the average processing time of elements (e.g., `new Arc(...)`) is relatively stable, so a method's average execution time in the new version may be estimated from that in the base version. Based on the three insights, we propose PerfRanker, which consists of four automatic steps. First, on a base version, we execute each test case in a profiling mode to collect information about the test execution, including the runtime call graph and the iteration counts of all executed loops. We also perform static analysis to capture the dependency among collection objects and loops. Second, based on the profiling information, we construct a performance model for each test case. Third, given a code commit, we estimate the execution time of each test case on the new version (formed by the code commit) by extending and revising its old performance model. We use profiling information and loop-collection correlations to infer parameters of the new performance model, and refer to this step as *Performance Impact Analysis*. Fourth, we rank all the test cases based on the performance impact on them. We implement our approach and apply it on two sets of code commits

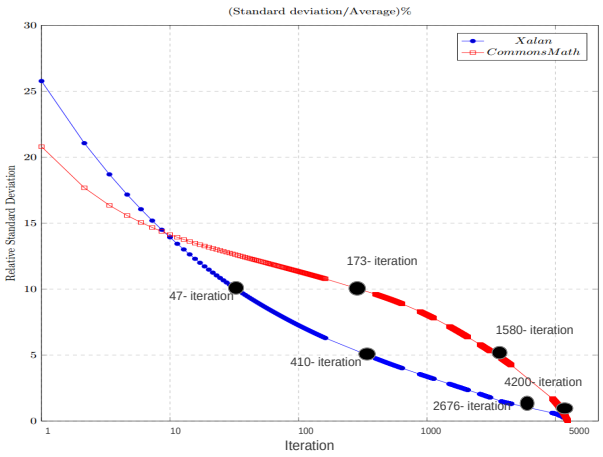

Figure 1: Relative Standard Deviation vs. Sample Size

collected from popular open source collection-intensive projects: Apache Commons Math and Xalan. To measure the effectiveness of test case prioritization for a code commit in performance regression testing, we use three metrics: (1) *APFD-P* (Average Percentage Fault Detected for Performance), an adapted version of the APFD metric [39] for performance testing, (2) *DCG* [5], a general metric for comparing the similarity of two sequences, and (3) *Top-N Percentile*, which calculates the percentage of test cases needed to be executed to cover the top N test cases whose execution time is most affected by the code commit. Our evaluation results show that, compared with the best of the three other baseline approaches, our approach achieves an average improvement of 17.6 percentage points on *APFD-P* and 27.4 percentage points on DCG. Furthermore, for Apache Commons Math and Xalan, our approach is able to rank top 1 affected test case within top 8% and top 16% test cases, and top 3 affected test cases within top 37% and 30% test cases, respectively. This paper makes the following major contributions:

- A novel approach to prioritizing test cases in performance regression testing of collection-intensive software.
- Adaptation of the APFD metric to measure the result of test prioritization for performance regression testing.
- An evaluation of our approach on real-world code commits from two popular open source collection-intensive projects.

2 MOTIVATION

In this section, we provide preliminary study results to motivate prioritization of performance regression tests, due to high cost of executing the same test case for many times for performance regression testing. In particular, modern mechanisms in hardware and software often bring in random factors impacting performance. Some well-known examples are the randomness in scheduling cores and buses in multi-core systems [48], in caching policies [50], and in the garbage-collection process [62], etc. These factors interact with each other and amplify their effect so that the execution time of a test case may vary substantially from time to time.

To neutralize such randomness, researchers or developers execute a test case multiple times and calculate the average performance [36]. To better understand this requirement, we perform a motivating study (with more details on the project website [3]) on the two open source projects used in evaluating our approach. In

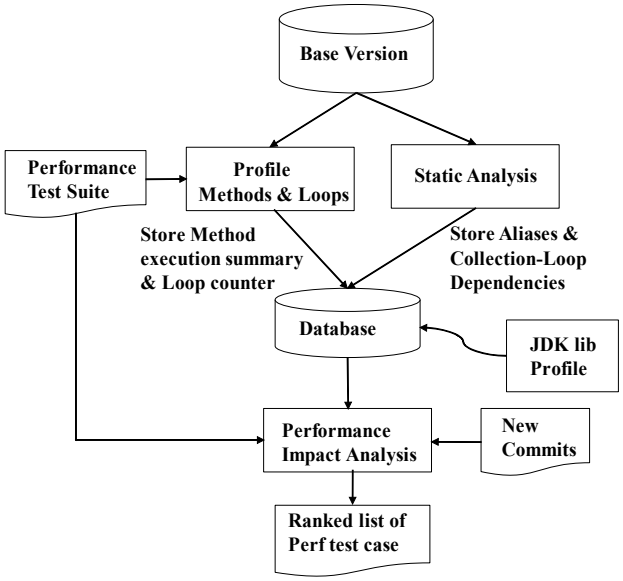

Figure 2: Workflow of Our Approach

particular, we execute the test cases for 5,000 times, and randomly select samples with different sizes to calculate their standard deviation. In Figure 1, we show how the execution time's relative standard deviation (y axis) changes as the execution times (x axis) increase from 1 to 5,000. The figure shows that the average relative standard deviation with sample size 1 is over 20% in both projects. In other words, if only one execution is used, the recorded execution time is expected to have more than 20% difference from the average execution time in the 5,000 executions. Note that 20% is a very large variance because 10% performance enhancement is typically considered significant, and techniques incurring over 10% overhead are often considered too slow for deployment purposes [35]. The figure also shows that 173 and 47 executions are required to achieve relative standard deviation less than 10% for Apache Commons Math and Xalan, respectively. Executing the whole test suite for many times can be prohibitively expensive, calling for prioritization of performance regression tests, as targeted by our approach.

3 APPROACH

In this section, we introduce our test prioritization approach in detail. In particular, we first present the overview of our approach, major technical challenges, and our performance model. After that, we introduce performance-impact estimation of a code commit based on our performance model, including method-execution-time estimation, and method-invocation-frequency estimation based on collection-loop correlation and iteration-count inference.

3.1 Overview

Figure 2 shows the workflow of our approach. The input to our approach includes the project code base, its code commits, and performance test cases. The output of our approach is an ordered list of performance test cases. The first step of our approach is to profile the base version of the project under test. During the profiling, for each test case, we record the dynamic call graph as its original performance model. We also record average execution time and frequency of methods, and iteration counts of all loops. At the same time, we statically analyze the base version to gather dependencies between loops and collection variables, as well as aliases among collection variables. When a new code commit comes, we conduct performance impact analysis to estimate its performance impact on all test cases, and prioritize test cases accordingly.

Technical Challenges. Performance impact analysis is the core of our approach. Although we focus on collection-intensive software, it is still challenging to conduct performance impact analysis, facing three major technical challenges:

- Challenge 1. A code commit may include any type and scope of code changes, from one-line revision, to feature addition and interface revision. Therefore, there is a strong need of a unified and formal presentation for code commits.
- Challenge 2. A code commit may contain newly added code, especially new loops. No execution information of such code is available, but given that loops can have high impact on performance, there is a strong need of estimating the code commit's execution time and frequency.
- Challenge 3. Even if the execution time of changed code in a code commit has little impact on performance, the code commit may include changes on collection variables, eventually affecting the performance of unchanged code.

To address Challenge 1, we present a code commit as three sets of methods: added methods, revised methods, and removed methods. As our performance model is based on the dynamic call graph, any code commit can be mapped to a series of operations for method addition, removal, and replacement in the performance model. To address Challenge 2, we leverage the recorded profiling information of the base version as much as possible. Specifically, if an existing method is invoked in the newly added code, we can use the recorded execution time of the existing method as its execution-time estimation for this new invocation. Furthermore, as discussed in Section 1, we use collection variables as bridges to estimate iteration counts of new loops from those of existing loops. To address Challenge 3, we track all the element-addition and element-removal operations of collection variables in the newly added code, and estimate the size change of collection variables from the iteration count of their enclosing loops. This new size is used to update the iteration counts of loops depending on the changed collection variables.

Since we focus on collection-intensive software, we consider only loops whose iteration number depends on collection variables, e.g., variables of array type, and other collection types defined in Java Utility Collections. Note that there are also some loops whose iteration number depends on simple integers, such as a loop to sum up a numbers from i to j, but such loops are not common in collection-intensive software, and our evaluation results show that our approach is effective on both data-processing software (Xalan) an mathematics software (Apache Commons Maths).

3.2 Performance Model

In this subsection, we introduce our performance model to break down execution time of a test case to all the methods invoked by the test case. The basic intuition behind our model is that the execution time of a method invocation M is the execution-time sum of all method invocations directly invoked in M, together with the execution time of instructions in M. Since most basic operations in

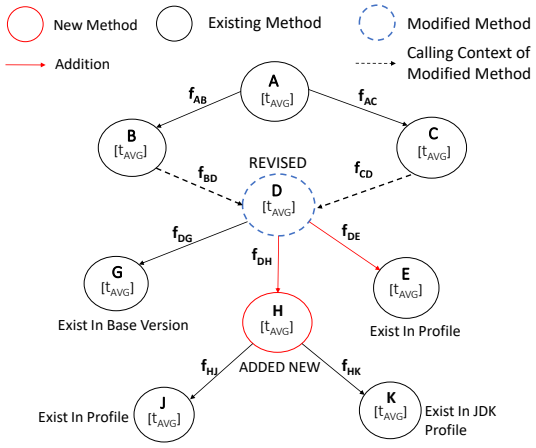

Figure 3: An Example Performance Model

Java programs are performed by JDK library methods (e.g., a string concatenation), the latter part is typically trivial compared with the former part, so our performance model ignores instructions in M itself, but focuses only on methods that M invokes.

We illustrate our model in Figure 3, where each node represents a method and each directed edge represents an invocation relation. Here each node annotated with label t_{avg}, which represents the average execution time of a method. Each edge in the graph is labeled with f_{AB}, which represents the invocation frequency of method B from method A. Given a code commit, the performance model of the post-commit version can be acquired by adding and removing nodes and edges to the original performance model. For example, in Figure 3, method D is a revised method and it now calls (1) method G, which it originally calls, (2) E, which is an existing method in the base version, and (3) H, which is a newly added method. With the average execution time and invocation frequency of all invoked methods in D, we are able to calculate an execution-time estimation for revised method D. The new execution time at D can be propagated upward to its ancestors, until the main method is reached and a new estimation of the whole program's execution time can be made.

3.3 Performance Impact Analysis

The basic idea of performance impact analysis is to calculate the execution-time change of each revised method M in the code commit. Then, through propagating the execution-time change to M's predecessors in the performance model, we can calculate the execution-time change of the whole test case at the root node.

We realize this idea in three steps. First, for each revised method, we extend the performance model to either add it and/or some of its callees (and transitive callees). Second, we estimate the execution-time change of each method in the new performance model. Third, we estimate the invocation frequency on the edges of the new performance model. We next introduce the three steps in details.

3.3.1 Model Extension. For each revised method, we add its direct and transitive callees (e.g., methods E through K in Figure 3 for the revised method m_D) into the performance model, if they do not already exist in the model. In this recursive process, we terminate the extension of a method node if it is an unrevised method existing in the base version, a JDK library method, or a method whose source code is not available. Since we use one base version for a series of code commits, a revised method (and even some of its predecessors) may not exist in the base version because they are added after the base version. In such a case, we transitively determine M's predecessors (callers) until we reach methods in the base version. For example, if C and D in Figure 3 are added between the base version and the code commit under analysis, we determine that D is invoked by B and C, and C is invoked by A, so that we add C and D to the performance model.

When the new code version of the revised methods is available, we statically determine the direct and transitive callers and callees for the revised method, and one remaining challenge is to resolve polymorphism, where one method invocation may have multiple targeted method bodies. Although it is straightforward to apply off-the-shelf points-to analysis, since we have the profiling information of the base version, we make use of the information to acquire a more precise call graph. Specifically, if a method invocation is not involved in the method diff (i.e., the method invocation can be mapped to the same method invocation in the base version, such as G in Figure 3), we assume that its target is not changed and we use the same targeted method body as recorded for the base version. Otherwise, we apply points-to analysis [33] in Soot [58] to find the possible targeted method bodies for the method invocation. When a method invocation is mapped to multiple method bodies, we add all bodies to the new performance model, and we divide the estimated frequency of the method invocation by the number of possible targets to attain the invocation frequency of each target.

3.3.2 Execution-Time Change of Method Bodies. The method bodies invoked from a revised method fall into three categories. The first category is removed method bodies. Their execution-time data are recorded in the performance model of the base version, and their new execution time is estimated as 0.

The second category includes method bodies already existing in the performance model of the base version. Such method bodies include both those defined in the source code and those defined in the JDK library, or those without source code. For existing method bodies, we simply use the recorded average execution time in the base version as their estimated average execution time. For bodies of JDK library methods, we profile Dacapo [10] to acquire the average execution time of those common JDK library methods. For methods without source code or those not invoked by Dacapo, we use the average execution time of all method bodies in the profile as their estimated execution time, as we have no further information. Note that when a method from the second category is added to the performance model, its original execution time is set as 0.

The third category includes newly added method bodies in the source code. *Note that such method bodies include both those added to the source code in the code commit and those added in any other code commits between the base version and the code commit under analysis.* They also include method bodies defined in libraries but are newly reached due to code revisions from the base version to the new version. For a newly added method body (such as H in Figure 3), as discussed in Section 3.3.1, we extract all its callee method bodies, and add them to the performance model (such as J

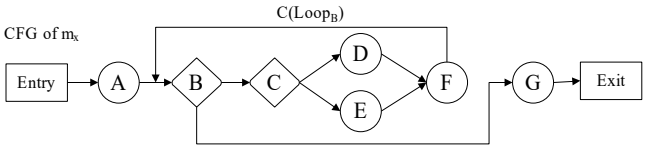

Figure 4: An Example Control Flow Graph

and K in Figure 3), and then we calculate its execution-time change using our performance model.

3.3.3 Invocation Frequencies of Method Bodies. Given a newly added or revised method m_x, for each method m_y that is directly invoked in m_x, we estimate the invocation frequency of m_y in m_x (denoted as $fq(m_x, m_y)$) to apply our performance model on the new version. Our estimation technique is based on the control flow graph of m_x and the average iteration counts of loops in m_x. Specifically, for any code block b in m_x, we use $fq(b, m_y)$ to denote the invocation frequency of m_y from b for each execution of b. If b is a basic block without branches and loops, $fq(b, m_y)$ is exactly the number of invocation statements to m_y in b; such number can be easily counted statically. Then we calculate $fq(m_x, m_y)$ by applying the inference rules for sequential, branch, and loop structures in Formulas 1-3 below recursively on the code blocks of m_x:

$$fq([b_1; b_2], m_y) = fq(b_1, m_y) + fq(b_2, m_y) \quad (1)$$

$$fq([\text{if() } b_1 \text{ else } b_2], m_y) = Max(fq(b_1, m_y), fq(b_2, m_y)) \quad (2)$$

$$fq([\text{while}_i \text{ () } b]) = fq(b, m_y) \times C(loop_i) \quad (3)$$

In the inference rules, the only unknown parameter is $C(loop_i)$, which denotes the average iteration count of the i^{th} loop in A. As an example, given the control flow graph in Figure 4 of m_x, for any m_y that m_x invokes, $fq(m_x, m_y)$ can be estimated as in Formula 4.

$$fq(m_x), m_y) = fq(A, m_y) + (Max(fq(D, m_y), fq(E, m_y)) + fq(F, m_y)) \times C(Loop_B) + fq(G, m_y) \quad (4)$$

3.4 Loop-Count Estimation with Collection-Loop Correlation

With the estimation of invocation frequency, the only remaining unknown parameter in the performance model of the new version is the loop count of all loops. If a loop exists in the base version and is not affected by the code commit, we directly use the recorded profile from the base version to acquire the iteration count. Two more complicated cases are (1) when a new loop is added, and (2) when the code commit affects the iteration count of an existing loop. Here is our insight: for collection-intensive software, we can construct the correlation between collection sizes and loop counts, and use iteration counts of known loops to infer that of unknown loops, as well as a code commit's impact on iteration counts of known loops.

3.4.1 Correlating Loops and Collections. In particular, we consider the following two types of dependencies between loops and collection variables:

- Iteration Dependency. A loop L is iteration-dependent on a collection variable v if L's loop condition depends on the size attribute of v.

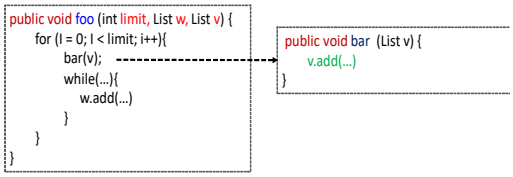

Figure 5: Code Sample of Operation Dependency

- Operation Dependency. A collection variable v is operation-dependent on a loop L if there exists an element addition or removal on v in L.

To identify iteration dependencies, for a For-Each loop (e.g, `for(A a : ListOfA)`), we simply consider that the loop is iteration-dependent on the collection variable being iterated via the loop. For other loops, we use standard inter-procedural data flow analysis [11] [51] to track data dependency backward from the loop condition expression, until we reach a size/length attribute of an array or a known collection class from the Java Collection Library. To make sure that the collection size is comparable with the loop count, we consider only two types of data dependencies: (1) direct assignment (e.g., `a = b;`), and (2) addition or subtraction expression with one operand as constant (e.g., `a = b.size() - 1`).

To identify operation dependencies, for each loop L, we check its body for element-addition and element-removal operations on collection variables. For any other method invocations in the loop, we recursively go into the body of each invoked method to further look for such operations. However, we do not consider nested-loop blocks in L or a method invoked from L, because such blocks are dependent on their direct enclosing loop. For example, in Figure 5, collection variable v is operation-dependent on Loop_A, but w is not (it is operation-dependent on Loop_B).

After identifying these two types of dependency relations, we further apply points-to analysis [33] to identify alias relations among collection variables. Note that in our analyses we consider only variables of known collection classes from the Java Collection Library. User-defined collections are also common. However, since we use inter-procedural analysis for identifying both types of dependencies, as long as the user-defined collections extend or wrap Java-Collection classes from the Java Collection Library, we are able to handle these user-defined collections by building dependencies directly on the Java-Collection variables inside them. Also note that we identify all dependencies and alias relations on the base version and record the results so that we need to re-analyze only the revised/added methods when a code commit comes.

3.4.2 Iteration-Count Inference. With the dependencies identified among collections and loops, when a new code commit comes, we use Algorithm 1 to infer the iteration count of new loops and update the iteration count of existing affected loops.

In the algorithm, we use a work queue to iteratively update sizes of collection variables and loop-iteration counts (stored in *lCount*), and we use the combined map of *MapI* and *MapO* to transit between collections and loops. In particular, as shown in Lines 6-11, we update the iteration count of each loop at most once, to avoid infinite update process caused by cyclic dependency (the more in-depth reason is that we use numbers to represent iteration counts, which are not in a bounded domain). In the end, we remove collection variables from *lCount* to retain only the loops in the map.

Algorithm 1 Iteration Count Inference

Input:
MapO is a map from collection variables to loops
MapI is a map from loops to collection variables
lCount is a map from loops to iteration counts
Output:
updated *lCount*

```
1:  Q ← MapI.keys()
2:  Map ← MapO ⋃ MapI
3:  while Q ≠ ∅ do
4:    top ← Q.pop()
5:    for all val ∈ Map.get(top) do
6:      if val ∉ lCount.keys() then
7:        lCount.add(val, lCount.get(top)
8:        Q.add(val)
9:      else if val is a collection variable then
10:       lCount.set(val, lCount.get(val) + lCount.get(top)
11:       Q.add(val)
12:     end if
13:   end for
14: end while
15: lCount.removeAll(MapO.keys())
```

3.5 Test Case Prioritization

Once our approach estimates the performance impact of the code commit on each test case, we can rank test cases according to their relative performance impact. We use the main method as the root for system tests and each test method as the root for unit tests. We consider both positive and negative effect on execute time as it is often also important for developers to understand whether and where their commit is able to enhance the software performance.

4 EVALUATION

For our evaluation, we implement PerfRanker based on Soot for static analysis and Java Agent for profiling the base version.

4.1 Evaluation Subjects

We apply PerfRanker on two popular open source projects: Xalan [2] and Apache Commons Math [1]. Specifically, Xalan is an XSLT processor, and Apache Commons Math is a library for mathematical operations. We choose these two projects to cover both data formatting and mathematical computations, which are two representative time-consuming components in modern software. Xalan is equipped with a performance test suite of 64 test cases. Since Apache Commons Math is not equipped with a performance test suite, we leverage its unit test cases as performance test cases.

Version Selection. For Apache Commons Math, we use its version on Jan 1, 2013 as its base version. For Xalan, since there are very few code commits after 2013, we use version 2.7.0 as its base version, as 2.7.0 is the first Xalan version compatible with Java 6 and higher. For both software projects, we collect all code commits from the base version until Mar 17, 2016, the time when we started collecting data for our work. From all code commits, we remove those that do not change source files and those that do not involve semantic changes (e.g., renaming variables), as developers can easily determine that those commits will not affect software performance. Furthermore, we choose as our code-commit set the top 15 code commits whose changed code portions are covered by most test cases, where test prioritization is most needed.

Table 1: Evaluation Subjects

Subject	Xalan	Apache Commons Math
Base Ver.	2_7_0	Jan 1st, 2013
Size (LOC) of Base Ver.	413,534	398,171
# Commits Since Base Ver.	354	1,321
# Changed Files	1,206	1,613
Last Commit Date	Aug 11, 2015	Mar 17, 2016

In Table 1, we present some statistics of the studied subjects and versions. The table shows that either project has more than 300K lines of code. Furthermore, there are hundreds of code commits and changed files between the base version and our selected code commits. In our evaluation, we do not update the base version, so the overhead of profiling the base version is low compared with the number of code commits under study. More details about our evaluation subjects can be found on our project website [3].

4.2 Evaluation Setup

To determine performance regressions as the ground truth of performance changes for all test cases and code commits, we execute the test cases for 5,000 times on the base version and each code commit under study. Furthermore, we execute the base version with our Java Agent to record the dynamic call graph and the execution time of each method, as well as the iteration number of each loop. To record average execution time of methods defined in the JDK library, we execute the Dacapo benchmark 9.12 [10] with profiling (we remove Xalan from the benchmark to avoid bias). All the executions are conducted on a Dell x630 PowerEdge Server with 32 cores and 256GB memory, and the server is used exclusively for our evaluation to avoid noises.

4.3 Evaluation Metrics

To the best of our knowledge, our work is the first on prioritizing performance test cases for code commits, and we propose a set of metrics to evaluate the quality of different rankings. In our evaluation, we consider three ranking metrics: Average Percent of Fault-Detection on Performance (*APFD-P*), normalized Discounted Cumulative Gain (*nDCG*), and *Top-N Percentile*.

APFD-P. *APFD* [39] is a commonly used metric for assessing a test sequence produced by test-case prioritization. If the test-suite size is N, the total number of faults detected by the test suite is T, and the number of faults detected by the first x test cases in the test sequence is $detected(x)$, then the *APFD* of the test sequence can be defined in Formula 5:

$$APFD = \frac{\sum_{x=1}^{N} \frac{detected(x)}{T}}{N} * 100\% \quad (5)$$

Unlike functional bugs where a test case either passes or fails, performance regressions are not binary but continuous. Performance downgrades of 20% and 50% are both regressions, with different severity. Therefore, instead of counting detected faults to attain the value of $detected(x)$, we replace the value of $detected(x)$ with the accumulated performance change. We define the *performance change* of the i^{th} test case in the test sequence (denoted as $change(i)$) in Formula 6 as below, in which $exe(i)$ is the execution time of the i^{th} test case in the current version, and $exe(i_{base})$ is the execution time of the i^{th} test case in the base version.

$$change(i) = \frac{|exe(i) - exe(i_{base})|}{exe(i_{base})} \quad (6)$$

Then, we define *APFD-P* the same as Formula 5, except that $detected(x)$ is defined as the accumulated performance change, as shown in Formula 7, and T is the sum of performance changes on all test cases. Actually, with such a definition, *APFD-P* can be viewed as *APFD* where all test cases reveal faults, and these faults are weighted by performance changes.

$$detected(x) = \sum_{i=1}^{\hat{x}} change(i) \quad (7)$$

As an illustrative example, consider 3 tests t_1, t_2, and t_3 with 10%, 20%, and 30% performance downgrades, respectively. The best ranking is t_3, t_2, t_1; the total performance impact is 10% + 20% + 30% = 60%; and the covered performance impact after each test is 30%/60% = 50%, (30% + 20%)/60% = 83%, and (30% + 20% + 10%)/60% = 100%. The P-APFD is thus (50% + 83% + 100%)/3 = 78%.

nDCG. nDCG [5] is a metric of ranking widely used in information retrieval. The basic idea is to calculate the relative score a given ranking with an ideal ranking, and the score of an arbitrary ranking is defined below, where $change(i)$ is defined in Formula 6.

$$DCG(seq) = change(1) + \sum_{i=2}^{N} \frac{change(i)}{log_2(i)} \quad (8)$$

Top-N Percentile. The *APFD-P* and *nDCG* defined earlier are adapted versions of widely used metrics, and can be used to compare different prioritization approaches. However, they are not sufficiently intuitive to help understand how much developers can benefit from an approach. Therefore, in our evaluation, we also measure how high percentage of top-ranked test cases in a test sequence need to be executed to cover the test cases with top N performance impacts (we use 1 and 3 for N). For example, if the test cases with top 1, 2, and 3 performance impacts are ranked in the 2^{nd}, 9^{th}, and 5^{th} positions in a test sequence with length 100, then the top 1, 2, and 3 percentiles are 2%, 9%, and 9%, respectively.

4.4 Baseline Approaches Under Comparison

Although we are not aware of approaches specifically designed for prioritizing performance test cases in regression testing, it is possible to adapt existing approaches for performance test prioritization. In our evaluation, we compare our approach with three baseline approaches: Change-Aware Random, Change-Aware Coverage, and Change-Aware Loop Coverage.

Specifically, in all baseline approaches, we apply change-impact analysis to rule out the test cases that do not cover any revised methods. Since our performance-impact analysis includes basic change-impact analysis, for fair comparison, we apply this change-impact analysis in all baseline approaches. Note that we gathered coverage information from the base version, and we use the same technique as in our approach when selecting the code commits affecting most test cases in the three baseline approaches.

After selecting the relevant test cases, the *Change-Aware Random (CAR)* approach simply ranks the test cases in random order[1]. The *Change-Aware Coverage (CAC)* approach applies coverage-based test prioritization [53] on the covered methods with the additional strategy [65], being a state-of-the-art approach in defect-oriented test prioritization. The basic idea is to first select the test case with the highest coverage, and iteratively select the test case that covers the most not-covered code portions as the next test case. In our evaluation, we use method coverage as the criterion, being consistent with the granularity of our performance model. The *Change-Aware Loop Coverage (CALC)* approach is the same as CAC, except for using coverage of loops instead of methods as the criterion.

[1]To acquire more stable results, we use the average result of 100 random ordered test sequences as the result for CAR.

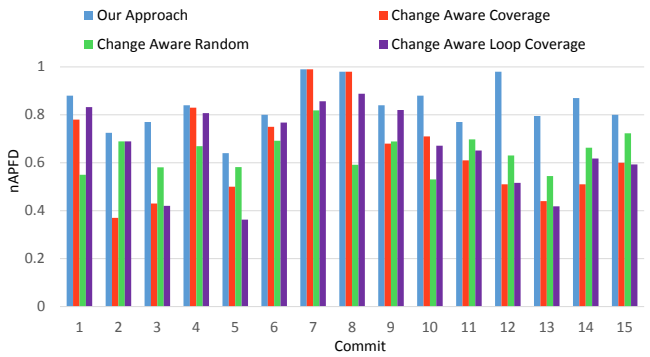

Figure 6: *APFD-P* Comparison on Apache Commons Math

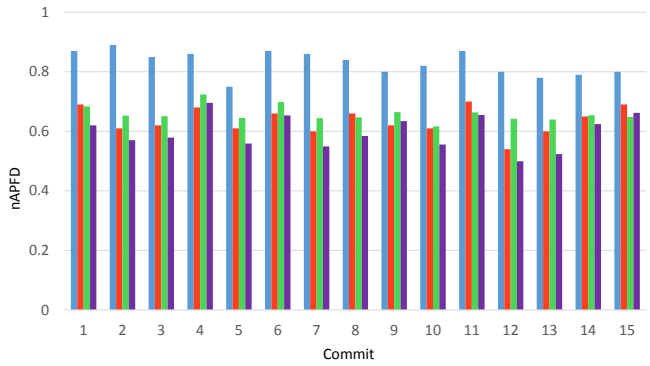

Figure 7: *APFD-P* Comparison on Xalan

4.5 Quantitative Evaluation

In our quantitative evaluation, we compare our approach and the three baseline approaches on all three metrics.

4.5.1 APFD-P Metric. Figures 6 and 7 show the comparison results between our approach and three baseline approaches on the *APFD-P* metric. In the two figures and all the following figures, the X axis lists all the code commits studied chronologically, and the Y axis shows the *APFD-P* value (or *nDCG* value). We use different colors to represent different approaches consistently for all figures according to the legend in Figure 6.

Figures 6 and 7 show that our approach is able to achieve over 80% *APFD-P* value in most code commits affecting performance, and outperforms or rivals all baseline approaches in all code commits affecting performance from both subject projects. Specifically, for Apache Commons Math, our approach achieves an average *APFD-P* value of 83.7%, compared with 64.3% by CAR, 64.6% by CAC, and 66.1% by CALC. For Xalan, our approach achieves an average *APFD-P* value of 83.5%, compared with 65.8% by CAR, 63.6% by CAC, and 59.8% by CALC. Therefore, the improvement on the average *APFD-P* is at least 17 percentage points, compared with baseline approaches

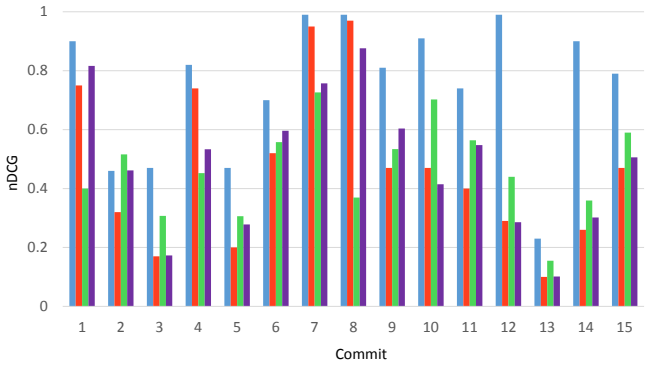

Figure 8: *nDCG* Comparison on Apache Commons Math

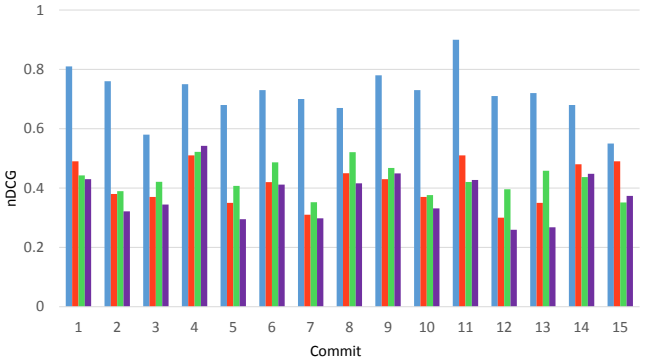

Figure 9: *nDCG* Comparison on Xalan

on both projects. Furthermore, we do not observe significant effectiveness downgrade in the later versions, indicating that one base version can be used for a relatively long time.

4.5.2 nDCG Metrics. Similarly, Figures 8 and 9 show the comparison results between our approach and the three baseline approaches on the *nDCG* metric.

The figures show that our approach outperforms or rivals baseline approaches on *nDCG* in almost all code commits from both subject projects. Specifically, for Apache Commons Math, our approach achieves an average *nDCG* value of 74.5%, compared with 47.2% by CAR, 46.5% by CAC, and 48.4% by CALC. For Xalan, our approach achieves an average *nDCG* value of 71.7%, compared with 43.0% by CAR, 41.4% by CAC, and 37.4% by CALC. Therefore, the improvement on the average *nDCG* is over 26 percentage points on both projects. We also observe that there is one code commit from Xalan, in which our approach performs slightly worse than CAR on *nDCG*. In Section 4.6, we further discuss the details of this code commit in Listing 4.

Finally, we observe that compared with *APFD-P*, the *nDCG* values are generally lower and vary more significantly from code commit to code commit. The reason is that an *nDCG* value is more sensitive to the rank of test cases with the highest performance impacts. For example, consider a test sequence with 100 test cases and only 1 test case has performance impact, and the performance impact value is 100%. When this test case is ranked top, both *APFD-P* and *nDCG* are 1.0. However, if this test case is ranked 25^{th} in the sequence, the *APFD-P* value is still as high as 75%, but the *nDCG* value becomes $1/log_2(25)$, which is less than 25%. This result is reasonable because

Table 2: Top-N Percentile of Apache Commons Math

C #	T #	Top 1 Our	Top 1 CAC	Top 1 CALC	Top 1 CAR	Top 3 Our	Top 3 CAC	Top 3 CALC	Top 3 CAR
1	55	2%	3%	**1%**	49%	**6%**	43	27%	74%
2	60	40%	61%	**38%**	49%	**51%**	61%	93%	74%
3	152	**2%**	84%	79%	49%	**28%**	88%	88%	74%
4	97	**3%**	18%	87%	49%	**5%**	18%	87%	74%
5	130	**4%**	29%	43%	49%	**4%**	96%	97%	74%
6	15	**13%**	40%	33%	46%	66%	**40%**	100%	73%
7	18	**5%**	**5%**	88%	47%	**15%**	40%	88%	74%
8	10	**10%**	**10%**	40%	45%	**60%**	**60%**	70%	70%
9	12	**8%**	66%	75%	45%	**50%**	66%	75%	72%
10	12	**8%**	50%	83%	45%	83%	**50%**	100%	72%
11	36	**5%**	94%	38%	48%	66%	94%	**58%**	74%
12	13	**7%**	76%	38%	45%	76%	84%	100%	**72%**
13	711	**7%**	81%	84%	49%	**7%**	81%	84%	74%
14	39	**2%**	84%	25%	48%	**12%**	84%	79%	74%
15	34	**11%**	52%	17%	48%	**26%**	58%	**26%**	74%
Avg		**8%**	50%	51%	48%	**37%**	65%	78%	73%

Table 3: Top-N Percentile of Xalan

C #	T #	Top 1 Our	Top 1 CAC	Top 1 CALC	Top 1 CAR	Top 3 Our	Top 3 CAC	Top 3 CALC	Top 3 CAR
1	63	20%	46%	**14%**	49%	**36%**	60%	50%	74%
2	63	**17%**	85%	22%	49%	**17%**	85%	80%	74%
3	63	**7%**	85%	50%	49%	**38%**	85%	66%	74%
4	63	20%	80%	**7%**	49%	**20%**	85%	73%	74%
5	63	**6%**	85%	100%	49%	**53%**	85%	100%	74%
6	63	**11%**	63%	31%	49%	**26%**	76%	77%	74%
7	63	**11%**	46%	12%	49%	**15%**	85%	93%	74%
8	63	**9%**	35%	36%	49%	**20%**	82%	95%	74%
9	58	**3%**	96%	5%	49%	**25%**	96%	98%	74%
10	63	30%	47%	**17%**	49%	**31%**	85%	84%	74%
11	63	**1%**	31%	12%	49%	**7%**	62%	74%	74%
12	63	**23%**	85%	88%	49%	**34%**	90%	88%	74%
13	63	**12%**	46%	82%	49%	**53%**	65%	82%	74%
14	58	34%	34%	**8%**	49%	43%	**37%**	72%	74%
15	63	36%	**4%**	12%	49%	**36%**	79%	61%	74%
Avg		**16%**	57%	34%	49%	**30%**	77%	79%	74%

in information retrieval (where *nDCG* was first proposed), ranking the most relevant result at the 25^{th} position is very bad, but in test prioritization (where *APFD* was first proposed), ranking the test case at the 25^{th} position is not so bad, because only 25% test cases need to be executed to execute the test case. Therefore, which of *APFD-P* and *nDCG* is a better metric may depend on whether developers are interested in only a few most severely affected test cases, or a larger number of test cases whose performance is affected.

4.5.3 Top-N Percentile. While *APFD-P* and *nDCG* are normalized quantitative metrics for our problem, they are not sufficiently intuitive for understanding the direct benefit of our approach on developers. Therefore, we further measure how many test cases developers need to consider if they want to cover the top 3 most affected test cases. Tables 2 and 3 show the results. Columns 1 and 2 present the code commit number and the total number of test cases affected by the code commit, respectively. Columns 3-6 and 7-10 present the top proportion of ranked test cases required to cover top 1 and 3 most-performance-affected test cases[2].

Tables 2 and 3 show that on average our approach is able to cover Top 1 and 3 most-performance-affected test cases within top 8%, 21%, and 37% of ranked test cases in Apache Commons Math, and 16%, 22%, and 30% of test cases in Xalan. The improvements over the baselines approaches are at least 33%, 37%, and 28%. Furthermore, on top 1 coverage, our approach outperforms or rivals the best baseline approach on 13 code commits from Apache Commons Math, and 10 code commits from Xalan. On top 3 coverage, our approach achieves the highest percentage on 11 code commits from Apache Commons Math, and 14 code commits from Xalan.

[2]The results for top 2 show similar trends and are available on the project website [3]

4.5.4 Overhead and Performance. In test prioritization, it is important to make sure that the time spent on prioritization is much smaller than the execution time of performance test cases. To confirm indeed that is the case, we record the overhead and execution time of our analysis. Our profiling of the base version has a 1.92 times overhead on Apache Commons Math, and 5.18 times overhead on Xalan. The static analysis per test case takes 29.90 seconds on Apache Commons Maths, and 34.35 seconds on Xalan. Finally, for Apache Commons Math, the average, minimal, and maximal time for analyzing a code commit are 45.35 seconds, 4.23 seconds, and 262.30 seconds, respectively, while for Xalan, the average, minimal, and maximal time for analyzing a code commit are 9 seconds, 3.36 seconds, and 21.80 seconds, respectively. In contrast, it takes averagely 52 (3) minutes to execute test suite of Apache Commons Math (Xalan) for 173 (47) times to achieve an expectation of equal to or less than 10% execution-time variance.

4.6 Successful and Challenging Examples

In this section, with representative examples of code commits, we explain why our approach performs well on some code commits but not so well on some others.

Successful Example 1. Listing 2 shows the simplified code change of code commit hash d074054... of Xalan. On this code commit, our approach is able to improve the *APFD-P* and *nDCG* values by at least 13.53 and 31.22, respectively, compared with the three baseline approaches. In this example, several statements are added inside a loop. Our approach can accurately estimate the performance change because (1) Line 2 in the code is an existing loop and from the profile database for the base version we can find out exactly how many times it is executed, and (2) the added method invocations are combinations of existing method invocations whose execution time is already recorded for the base version.

```
1    int nAttrs = m_avts.size();
2    for (int i = (nAttrs - 1); i >= 0; i--){
3      ...
4  +   AVT avt = (AVT) m_avts.get(i);
5  +   avt.fixupVariables(vnames, cstate.getGlobalsSize());
6      ...
7    }
```

Listing 2: Change Inside a Loop

Successful Example 2. Listing 3 shows the simplified code change of code commit hash 64ec535... of Xalan. On this code commit, our approach is able to improve the *APFD-P* and *nDCG* values by at least 17.14 and 24.33, respectively, compared with the baseline approaches. In this example, the loop at Line 4 depends on the collection variable `m_prefixMappings` at Line 1 whose size can be inferred from the recorded number of iterations of existing loops. In this case, our approach can accurately estimate the iteration count of this new loop and estimate the overall performance impact based on the estimated iteration count.

```
1    + int nDecls = m_prefixMappings.size();
2    +
3    + for (int i = 0; i < nDecls; i += 2){
4    +   prefix = (String) m_prefixMappings.elementAt(i);
5    +   ...
6    + }
```

Listing 3: A Newly Added Loop Correlating to an Existing Collection

Challenging Example 1. Listing 4 shows the simplified code change of code commit hash 90e428d... of Apache Commons Math. On this code commit, with respect to the *nDCG* metric, our approach performs better than the CAC baseline approach but slightly worse than the CAR baseline approach. In the example, at Line 2, an invocation to method `checkParameters()` is added, and the method may throw an exception. In this example, the execution time of `checkParameters()` can be easily estimated with our performance model. However, if the exception is thrown, the rest of the method will not be executed. Although we are able to estimate the execution time of the method's remaining part, it is impossible to estimate the probability of throwing the exception, as `checkParameters()` is a newly added method without any profile information. In such cases, if the probability of throwing the exception is higher in some test cases, the reduction of execution time due to the exception will be the dominating factor and result in inaccuracy in our prioritization.

```
1    protected PointVectorValuePair doOptimize(){
2  +   checkParameters();
3      ...
4    }
5  + private void checkParameters() {
6  +   ...
7  +   throw new MathUnsupportedOperationException;
8  + }
```

Listing 4: Return or Throw Exception at Beginning

Challenging Example 2. There are also cases where developers added a loop that is not relevant to any existing collection variables. As one of such examples, Listing 5 shows the simplified code change of code commit hash a51119c... of Apache Commons Math. On this code commit, the improvement of our approach over the best result of the three baseline approaches is only 0.8 for *APFD-P* and 6.0 for *nDCG*. In the example, Line 2 introduces a new loop that does not correlate to any existing collection or array. In such a case, our approach cannot determine the iteration count of this loop and the depth of recursion at Line 8. To still provide prioritization results, our approach uses the average iteration counts of all known loops to estimate the iteration count of this new loop and always estimates the recursion depth to be 1. However, our prioritization result becomes less precise due to such coarse approximation.

```
1    public long nextLong(final long lower, final long upper){
2  +   while (true) {
3        ...
4  +     if (r >= lower && r <= upper) {
5  +       return r;
6  +     }
7  +   }
8  +   return lower + nextLong(getRan(), max);
9    }
```

Listing 5: New Loop with No Correlation

4.7 Threats to Validity

Major threats to internal validity are potential faults in the implementation of our approach and baseline approaches, potential errors in computing evaluation results of various metrics, and the various factors affecting the recorded execution time for the test cases. To reduce such threats, we carefully implement and inspect all the programs, and execute the test cases for 5,000 times to reduce random noises in execution time. Major threats to external validity are that our evaluation results may be specific to the code commits and subjects studied. To reduce the threats, we evaluate our approach on both data processing/formatting software and mathematical computing software, and both a unit test suite and a performance test suite.

5 DISCUSSION

Handling Recursions. Recursions and loops are two major ways to execute a piece of code iteratively. Our approach constructs dependency relationships between loops and collection variables to estimate the iteration counts of loops. For recursion cycles existing in the base version, we simply update execution-time changes along the cycle only once, and multiply the execution-time change by averaging the invocation depths, attained via dividing the total execution frequency of all methods in the cycle by the product of invocation-frequency sum of these methods from outside the cycle and the accumulated recursive invocations inside the cycle (multiplying invocation frequencies along the cycle once). As an example, consider a cyclic call graph: $X \rightarrow A$, $Y \rightarrow B$, $A \rightarrow B$, and $B \rightarrow A$. When $t(B)$ is changed, the impact is propagated to A as $t(A) = t(B) * f_{AB}$. The propagation then stops to break the cycle. After that, impact on X becomes $f_{XA} * depth * t(A)$ where $depth$ is the cycle's number of execution iterations, estimated as below:

$$\frac{total(A) + total(B)}{(f_{XA} + f_{YB}) * f_{AB} * f_{BA}} \tag{9}$$

where $total(F)$ is the total frequency of method F in profile, and the divisor is the product of all calling frequencies from outside and inside of a cycle. For newly added recursions, our approach currently does not support estimation of the invocation depth, and simply assumes the invocation depth to be 1. In future work, we plan to develop analysis to estimate the termination condition of newly added recursions and relate invocation depths to collection variables.

Approximation in Performance Estimation. Since our approach is not able to make any assumption on the given code commit, we have to make coarse approximations for the parameters of our performance model. For example, we ignore non-method-call instructions in revised methods, assume all newly added recursions to have invocation depth 1, and use the average recorded execution time of existing methods and JDK library methods to estimate their execution time in the new version. More advanced analysis can result in more accurate execution-time estimation, yet with higher overhead in the prioritization process, so future investigation is needed for the best trade-off.

Supporting Multi-threaded Programs. Multi-threaded programs are widely used for high-performance systems. In multi-threaded programs, methods and statement blocks can be executed concurrently, and thus our performance model can be inaccurate because the product of invocation frequency and average execution time of a method is no longer the total execution time. To address concurrent execution, we need to analyze the base version to find out the methods that can be executed concurrently. Then, we can give such methods a penalizing coefficient to reflect the extent of concurrency. We plan to explore this direction in future work.

Selection of Base Versions. The overhead of our approach largely depends on the required number of base versions. In our evaluation, we use one base version to estimate the subsequent code commits ranging over more than 3 years and 300 code commits. The evaluation results do not show significant effectiveness downgrade as time goes by. One potential reason is that, both software projects in our evaluation are in their stable phase and the code commits are less likely to interfere with each other.

6 RELATED WORK

Performance Testing and Faults. Previous work focuses on generating performance test infrastructures and test cases, such as automated performance benchmarking [27], model-based performance testing framework for workloads [8], using genetic algorithms to expose performance regressions [38], learning-based performance testing [19], symbolic-execution-based load-test generation [66], probabilistic symbolic execution [12], and profiling-based test generation to reach performance bottlenecks [37]. Pradel et al. [49] propose an approach to support generation of multi-threaded tests based on single-threaded tests. Kwon et al. [30] propose an approach to predict execution time of a given input for Android apps. Bound analyses [20] try to statically estimate the upper bound of loop iterations regarding input sizes, but they cannot be directly applied as the size of collection variables under a certain test can be difficult to determine. Most recently, Padhye and Sen [47] propose an approach to identify collection traversals in program code; such approach has the potential to be used for execution-time prediction. In contrast to such previous work, our approach focuses on prioritizing existing performance test cases. The most related work in this direction is done by Huang et al. [24], whose differences with our approach are elaborated in Section 1.

Another related area is research on performance faults, including studies on performance faults [25, 64], static performance-fault detection [26, 28, 45, 63], debugging of known performance faults [4, 21, 32, 54], automatic patches of performance faults [44], and analysis of performance-testing results [16, 17].

Test Prioritization and Impact Analysis. Test prioritization is a well explored area in regression testing to reduce test cost [9, 23, 67] or to detect functional faults earlier [15, 29, 34]. Mocking [43] is another approach to reduce test cost, but it does not work for performance testing as mocked methods do not have normal execution time. Another related area is test selection or reduction [13, 22, 52] which sets a threshold or other criteria to select/remove part of the test cases. Most of the proposed efforts are based on some coverage criterion for test cases, and/or impact analysis of code commits. The impact analysis falls into three categories: static change impact analysis [7, 57, 59], dynamic impact analysis [6, 31, 46], and version-history-based impact analysis [41, 55, 68]. Our approach leverages a similar strategy to rank performance tests according to the change impact on them. However, we propose specific techniques to estimate performance impacts, such as method-based performance model and collection-loop correlation.

7 CONCLUSION

In this paper, we present a novel approach to prioritizing performance test cases according to a code commit's performance impact on them. With our approach, developers can execute most-affected test cases earlier and for more times to confirm a performance regression. Our evaluation results show that our approach is able to achieve large improvement over three baseline approaches, and to cover top 3 most-performance-affected test cases within 37% and 30% test cases on Apache Commons Math and Xalan, respectively.

ACKNOWLEDGMENT. UTSA's work is supported in part by NSF grant CCF-1464425 and DHS grant DHS-14-ST-062-001. Tao Xie's work is supported in part by NSF under grants No. CCF-1409423, CNS-1434582, CNS-1513939, CNS-1564274.

REFERENCES

[1] Apache Commons Maths. http://commons.apache.org/proper/commons-math/
[2] Xalan for Java. https://xml.apache.org/xalan-j/
[3] 2017. PerfRanker Project Web. (2017). https://sites.google.com/site/perfranker2017/
[4] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. 74–89.
[5] Azzah Al-Maskari, Mark Sanderson, and Paul Clough. 2007. The Relationship Between IR Effectiveness Measures and User Satisfaction. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 773–774.
[6] Taweesup Apiwattanapong, Alessandro Orso, and Mary J. Harrold. 2005. Efficient and Precise Dynamic Impact Analysis Using Execute-after Sequences. In *Proceedings of the 27th International Conference on Software Engineering*. 432–441.
[7] Robert S. Arnold. 1996. *Software Change Impact Analysis*. IEEE Computer Society Press.
[8] Cornel Barna, Marin Litoiu, and Hamoun Ghanbari. 2011. Model-based Performance Testing (NIER Track). In *Proceedings of the 33rd International Conference on Software Engineering*. 872–875.
[9] Jennifer Black, Emanuel Melachrinoudis, and David Kaeli. 2004. Bi-Criteria Models for All-Uses Test Suite Reduction. In *Proceedings of the 26th International Conference on Software Engineering*. 106–115.
[10] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*. 169–190.
[11] Eric Bodden. 2012. Inter-procedural Data-flow Analysis with IFDS/IDE and Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*. 3–8.
[12] Bihuan Chen, Yang Liu, and Wei Le. 2016. Generating Performance Distributions via Probabilistic Symbolic Execution. In *Proceedings of the 38th International Conference on Software Engineering*. 49–60.
[13] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. 1994. TESTTUBE: A System for Selective Regression Testing. In *Proceedings of 16th International Conference on Software Engineering*. 211–220.
[14] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. 2004. Early Performance Testing of Distributed Software Applications. In *Proceedings of the 4th International Workshop on Software and Performance*. 94–103.
[15] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2000. Prioritizing Test Cases for Regression Testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*. 102–112.
[16] King Chun Foo. 2011. Automated Discovery of Performance Regressions in Enterprise Applications. In *Master's Thesis*.
[17] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, and Parminder Flora. 2010. Mining Performance Regression Testing Repositories for Automated Performance Analysis. In *2010 10th International Conference on Quality Software*. 32–41.
[18] Gregory Fox. 1989. Performance Engineering As a Part of the Development Life Cycle for Large-scale Software Systems. In *Proceedings of the 11th International Conference on Software Engineering (ICSE '89)*. 85–94.
[19] Mark Grechanik, Chen Fu, and Qing Xie. 2012. Automatically Finding Performance Problems with Feedback-directed Learning Software Testing. In *Proceedings of the 34th International Conference on Software Engineering*. 156–166.
[20] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. 2009. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 127–139.
[21] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. 2012. Performance Debugging in the Large via Mining Millions of Stack Traces. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. 145–155.
[22] Dan Hao, Tao Xie, Lu Zhang, Xiaoyin Wang, Jiasu Sun, and Hong Mei. 2009. Test Input Reduction for Result Inspection to Facilitate Fault Localization. *Automated Software Engineering* 17, 1 (2009), 5.
[23] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. 1993. A Methodology for Controlling the Size of a Test Suite. *ACM Trans. Softw. Eng. Methodol.* 2, 3 (July 1993), 270–285.
[24] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. 2014. Performance Regression Testing Target Prioritization via Performance Risk Analysis. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. 60–71.
[25] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 77–88.
[26] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. 2011. 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*. 155–170.
[27] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. 2005. Automated Detection of Performance Regressions: The Mono Experience. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 183–190.
[28] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. 2010. Finding Latent Performance Bugs in Systems Implementations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 17–26.
[29] Jung-Min Kim and Adam Porter. 2002. A History-based Test Prioritization Technique for Regression Testing in Resource Constrained Environments. In *Proceedings of the 24th International Conference on Software Engineering*. 119–129.
[30] Yongin Kwon, Sangmin Lee, Hayoon Yi, Donghyun Kwon, Seungjun Yang, Byung-Gon Chun, Ling Huang, Petros Maniatis, Mayur Naik, and Yunheung Paek. 2013. Mantis: Automatic Performance Prediction for Smartphone Applications. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. 297–308.
[31] James Law and Gregg Rothermel. 2003. Whole Program Path-Based Dynamic Impact Analysis. In *Proceedings of the 25th International Conference on Software Engineering*. 308–318.
[32] Andrew W. Leung, Eric Lalonde, Jacob Telleen, James Davis, and Carlos Maltzahn. 2007. Using Comprehensive Analysis for Performance Debugging in Distributed Storage Systems. In *24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*. 281–286.
[33] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using Spark. In *Compiler Construction: 12th International Conference, CC 2003*. 153–169.
[34] Zheng Li, Mark Harman, and Robert M. Hierons. 2007. Search Algorithms for Regression Test Case Prioritization. *IEEE Trans. Softw. Eng.* 33, 4 (April 2007), 225–237.
[35] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2016. DoubleTake: Fast and Precise Error Detection via Evidence-based Dynamic Analysis. In *Proceedings of the 38th International Conference on Software Engineering*. 911–922.
[36] Tongping Liu and Xu Liu. 2016. Cheetah: Detecting False Sharing Efficiently and Effectively. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. 1–11.
[37] Qi Luo. 2016. Automatic Performance Testing Using Input-sensitive Profiling. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 1139–1141.
[38] Qi Luo, Denys Poshyvanyk, and Mark Grechanik. 2016. Mining Performance Regression Inducing Code Changes in Evolving Software. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 25–36.
[39] Alexey G. Malishevsky, Joseph R. Ruthruff, Gregg Rothermel, and Sebastian Elbaum. 2006. *Cost-Cognizant Test Case Prioritization*. Technical Report.
[40] Marissa Mayer. In Search of a Better, Faster, Stronger Web. http://goo.gl/m4fXx
[41] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. 2012. A History-Based Matching Approach to Identification of Framework Evolution. In *2012 34th International Conference on Software Engineering (ICSE)*. 353–363.
[42] M. MITCHELL. GCC Performance Regression Testing Discussion. http://gcc.gnu.org/ml/gcc/2005-11/msg01306
[43] Shaikh Mostafa and Xiaoyin Wang. 2014. An Empirical Study on the Usage of Mocking Frameworks in Software Testing. In *14th International Conference on Quality Software*. 127–132.
[44] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. CARAMEL: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 902–912.
[45] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-Access Patterns. In *2013 35th International Conference on Software Engineering (ICSE)*. 562–571.
[46] Alessandro Orso, Taweesup Apiwattanapong, and Mary J. Harrold. 2003. Leveraging Field Data for Impact Analysis and Regression Testing. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 128–137.
[47] Rohan Padhye and Koushik Sen. 2017. TRAVIOLI: A Dynamic Analysis for Detecting Data-Structure Traversals. In *Proceedings of the International Conference on Software Engineering*. 473–483.
[48] Heekwon Park, Seungjae Baek, Jongmoo Choi, Donghee Lee, and Sam H. Noh. 2013. Regularities Considered Harmful: Forcing Randomness to Memory Accesses to Reduce Row Buffer Conflicts for Multi-core, Multi-bank Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. 181–192.

[49] Michael Pradel, Markus Huggler, and Thomas R. Gross. 2014. Performance Regression Testing of Concurrent Classes. In *Proceedings of the International Symposium on Software Testing and Analysis*. 13–25.
[50] C. Pyo, S. Pae, and G. Lee. 2009. DRAM as Source of Randomness. *Electronics Letters* 45, 1 (2009), 26–27.
[51] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 49–61.
[52] Gregg Rothermel and Mary J. Harrold. 1997. A Safe, Efficient Regression Test Selection Technique. *ACM Trans. Softw. Eng. Methodol.* 6, 2 (April 1997), 173–210.
[53] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary J. Harrold. 1999. Test Case Prioritization: An Empirical Study. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*. 179–188.
[54] Kai Shen, Ming Zhong, and Chuanpeng Li. 2005. I/O System Performance Debugging Using Model-driven Anomaly Characterization. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*. 23–23.
[55] Mark Sherriff and Laurie Williams. 2008. Empirical Software Change Impact Analysis Using Singular Value Decomposition. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*. 268–277.
[56] Stoyan Stefanov. 2008. Yslow 2.0. In *CSDN Software Development 2.0 Conference*.
[57] Richard J. Turver and Malcolm Munro. 1994. An early impact analysis technique for software maintenance. *Journal of Software Maintenance: Research and Practice* 6, 1 (1994), 35–52.
[58] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. 13.
[59] Xiaoyin Wang, David Lo, Jiefeng Cheng, Lu Zhang, Hong Mei, and Jeffrey Xu Yu. 2010. Matching Dependence-related Queries in the System Dependence Graph. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. 457–466.
[60] Elaine J. Weyuker and Filippos I. Vokolos. 2000. Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study. *IEEE Trans. Softw. Eng.* 26, 12 (Dec. 2000), 1147–1156.
[61] Elain J. Weyuker and Filippos I. Vokolos. 2000. Experience with Performance Testing of Software Systems: Issues, An Approach, and Case Study. *IEEE Transactions on Software Engineering* 26, 12 (2000), 1147–1156.
[62] Paul R. Wilson. 1992. *Uniprocessor Garbage Collection Techniques*. 1–42.
[63] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2012. Uncovering Performance Problems in Java Applications with Reference Propagation Profiling. In *Proceedings of the 34th International Conference on Software Engineering*. 134–144.
[64] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. 2012. A Qualitative Study on Performance Bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. 199–208.
[65] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. 2013. Bridging the Gap Between the Total and Additional Test-case Prioritization Strategies. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. 192–201.
[66] Pingyu Zhang, Sebastian Elbaum, and Matthew B. Dwyer. 2011. Automatic Generation of Load Tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. 43–52.
[67] Hao Zhong, Lu Zhang, and Hong Mei. 2006. An Experimental Comparison of Four Test Suite Reduction Techniques. In *Proceedings of the 28th International Conference on Software Engineering*. 636–640.
[68] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. 2004. Mining Version Histories to Guide Software Changes. In *Proceedings of the 26th International Conference on Software Engineering*. 563–572.