Mining Performance Regression Inducing Code Changes in Evolving Software

Qi Luo, Denys Poshyvanyk, Mark Grechanik*

College of William and Mary
*University of Illinois at Chicago

MSR 2016
Austin, TX, U.S.
public synchronized void regressionFunc ()
{
    static Regressions staticRT = new Regressions();
    public void setTeams(Collection<Team> teams) {
        this.teams = teams;
        Backlog.staticRT.regressionFunc();
    }
}

Set<Integer> selectedBacklogIds = this.getSelectedBacklogs();
if (selectedBacklogIds == null || selectedBacklogIds.size() == 0) {
    Collection<Product> products = new ArrayList<Product>();
    productBusiness.storeAllTimeSheets(products);
    for (Product product: products) {
        selectedBacklogIds.add(product.getId());
    }
}
return Action.SUCCESS;

for (Story child : story.getChildren()) {
    if (child.getId() == story.getId()) {
        continue;
    }
    StoryTreeBranchMetrics childMetrics = this.calculateStoryTreeMetrics(child)
    return metrics;
}
Automated Detection of Performance Regressions Using Regression Models on Clustered Performance Counters

Weiyi Shang, Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University, Kingston, Ontario, Canada
{weiy, ahmed}@cs.queensu.ca
Mohamed Nasser, Parminder Flora
Waterloo, Ontario, Canada

1. INTRODUCTION

Performance testing is conducted before deploying system updates in order to ensure that the performance of large software systems did not degrade (i.e., no performance regressions). During such testing, thousands of performance counters are collected. However, comparing thousands of performance counters across versions of a software system is very time consuming and error-prone. In an effort to auto-

Main Effects Screening: A Distributed Continuous Quality Assurance Process for Monitoring Performance Degradation in Evolving Software Systems

Cemal Yilmaz1, Arvind S. Krishnamurthy, Afif Momon1, Adam Porter1, Douglas C. Schmidt1, Aniruddha Gokhale1, Balachandran Narasaran1

1Dept. of Computer Science, University of Maryland, College Park, MD 20742

Although these software parameters promote flexibility and portability, they also require that the software be tested in an enormous number of configurations. This creates seri-

Abstract

Developers of highly configurable performance regression testing systems use a type of in-house "regression testing" to ensure that their parameters have not adversely affected the software performance across its large configuration space. Although this space can be large, the developers have observed that the performance of the software is often similar across different configurations. They then conclude that the observed performance is likely to be similar across all configurations. However, this conclusion is often not based on any empirical evidence. Thus, the developers are often making decisions about the software without any empirical evidence. This can lead to poor decisions and wasted resources. To address this problem, we propose a new approach for performing empirical performance regression testing. Our approach is based on the idea of using statistical process control techniques to monitor the performance of the software across its large configuration space. We show that our approach can be used to improve the accuracy of performance regression testing, and can be used to make better decisions about the software.

Mining Performance Regression Testing Repositories for Automated Performance Analysis

King Chan Foo1, Zhen Ming Jiang2, Bram Adams2, Ahmed E. Hassan2, Ying Zou2, Parminder Flora1

Department of Electrical and Computer Engineering1
Queen's University
Kingston, ON, Canada
{k.chan, ying.zou}@queensu.ca
Performance Engineering2
School of Computing2
Queen's University
Kingston, ON, Canada
(zmjiang, bram, ahmed)@cs.queensu.ca

Abstract— Performance regression testing detects performance regressions in software systems. Such regression testing is conducted by comparing the performance of software systems with previous versions. Although the new version of the software system is expected to have improved performance, the performance of the new version of the software system may be worse than the previous version. In this paper, we present a method for automating the process of performance regression testing. Our method involves using a statistical technique to automatically detect performance regressions in software systems. We validate our method by applying it to a number of software systems and compare the results to those obtained by manual performance regression testing. Our results show that our method is effective in detecting performance regressions in software systems.

Automated Detection of Performance Regressions Using Statistical Process Control Techniques

Thanh H. D. Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University
Kingston, Ontario, Canada
{thanhnguyen, bram, zmj, ahmed}@cs.queensu.ca
Mohamed Nasser, Parminder Flora
Research In Motion
Waterloo, Ontario, Canada

ABSTRACT

The goal of performance regression testing is to check for performance regressions in a new version of a software system. Performance regression testing is an important phase in the software development process. Performance regression testing is often time consuming and there is usually little time available for it. A typical test run would output features, code changes might degrade the software's performance. Hence, performance engineers must perform regression tests to make sure that the software still performs as good as previous versions. Performance regression testing is very important in large software systems where a large number of changes in the software are required. In this paper, we present a method for automatically detecting performance regressions in software systems. Our method is based on the idea of using statistical process control techniques to monitor the performance of the software across its large configuration space. We show that our method can be used to detect performance regressions in software systems.
Increased Execution Time
Increased Execution Time

Problematic Code Change
public synchronized void regressionFunc();
static Regressions staticRT = new Regressions();

public void setTeams(Collection<Team> teams) {
    this.teams = teams;
    Backlog.staticRT.regressionFunc();
}

Set<Integer> selectedBacklogIds = this.getSelectedBacklogs();
if (selectedBacklogIds == null || selectedBacklogIds.size() == 0) {
    Collection<Product> products = new ArrayList<Product>();
    productBusiness.storeAllTimeSheets(products);
    for (Product product; products) {
        selectedBacklogIds.add(product.getId());
    }
}
return Action.SUCCESS;

for (Story child : story.getChildren()) {
    if (child.getId() == story.getId()) {
        continue;
    }
    StoryTreeBranchMetrics childMetrics = this.calculateStoryTreeMetrics(child)
    return metrics;
PerfImpact – performance regression testing

- Search-based input profiling to find specific inputs for exposing performance regressions
- Change impact analysis (CIA) to recommend the problematic code changes
Stage 1
GA component
Stage 1
GA component

Initial Inputs → Inputs → JMeter

Vi → Profileri
Vi+1 → Profileri+1
Profileri → Execution Trace Analyzer
Profileri+1 → Trace Statistics

GA Analyzer

Stage 1
GA component

1. Initial Inputs
2. Inputs
3. JMeter
4. Vi
5. Profileri
6. Vi+1
7. Profileri+1
8. Execution Trace Analyzer
9. GA Analyzer
10. JMeter
Stage 1
GA component
Initial Inputs → Inputs → JMeter → Vi → Profileri → Execution Trace Analyzer → Trace Statistics → GA Analyzer → Mutation → Crossover → Selection → Vi+1 → Profileri+1 → Method Statistics → Ranked lists of changes → Stage 1

Stage 1
GA component

Stage 2
CIA component

Changes → Impact Analysis → Impact Sets → Mining → Stage 2

Profileri+1

Method Statistics → Stage 2

Profileri

Execution Trace Analyzer

Trace Statistics

Selection → Stage 1

Crossover → Stage 1

Mutation → Stage 1

Inputs

JMeter

Vi

Vi+1

Initial Inputs
Initial Inputs → Inputs (Stage 1: GA component) → JMeter → Vi → Profiler_\text{i} → Execution Trace Analyzer → Method Statistics

Stage 2 (CIA component):
- Changes → Impact Analysis → Impact Sets → Mining → Ranked lists of changes
Stage 1
GA component

Stage 2
CIA component
Genes:

Input 1: http://localhost:8080/Agilefant/editUser.action
Input 3: http://localhost:8080/Agilefant/editProduct.action?productId=8
......

A chromosome/individual

Individual 1: 2, 18, 36, 27, 11, 13, 6, 43, 64, 12, 85, 49, 12, 53, 44, 78, 31, 47
An individual \((I_{j})\)

- Execution time in \(V_i\) \((t_{j}^i)\)
- Execution time in \(V_{i+1}\) \((t_{j}^{i+1})\)
Stage 1
GA component

Stage 2
CIA component
• Selection

Fitness value for $I_j = t_{j+1}^i - t_j^i$
• Selection

Fitness value for $I_j = t_{j+1}^i - t_j^i$

• Crossover

Parent 1: 2, 18, 36, 27, 11, 13, 6, 43, 64, 12, 85, 49, 12, 53, 44, 91, 79, 23, 3, 19
Parent 2: 23, 95, 1, 67, 35, 81, 7, 17, 51, 102, 56, 39, 72, 3, 54, 37, 13, 86, 47, 76

Child 1: 2, 18, 36, 27, 11, 13, 6, 17, 51, 102, 56, 39, 72, 3, 54, 37, 13, 86, 47, 76
Child 2: 23, 95, 1, 67, 35, 81, 7, 43, 64, 12, 85, 49, 12, 53, 44, 91, 79, 23, 3, 19
• Selection

Fitness value for $I_j = t_{j+1}^i - t_j^i$

• Crossover

Parent 1: 2, 18, 36, 27, 11, 13, 6, 43, 64, 12, 85, 49, 12, 53, 44, 91, 79, 23, 3, 19
Parent 2: 23, 95, 1, 67, 35, 81, 7, 17, 51, 102, 56, 39, 72, 3, 54, 37, 13, 86, 47, 76
Child 1: 2, 18, 36, 27, 11, 13, 6, 17, 51, 102, 56, 39, 72, 3, 54, 37, 13, 86, 47, 76
Child 2: 23, 95, 1, 67, 35, 81, 7, 43, 64, 12, 85, 49, 12, 53, 44, 91, 79, 23, 3, 19

• Mutation

Parent: 2, 18, 36, 27, 11, 13, 6, 43, 64, 12, 85, 49, 12, 53, 44, 91, 79, 23, 3, 19
Child: 2, 18, 36, 27, 11, 13, 6, 43, 64, 73, 85, 49, 12, 53, 44, 91, 79, 23, 3, 19
Stage 1
GA component

Stage 2
CIA component
Stage 1
GA component

Initial Inputs

Execution Trace Analyzer

Profiler\textsubscript{i}

Profiler\textsubscript{i+1}

Profiler\textsubscript{i+1}

Method Statistics

Stage 2
CIA component

Changes

Impact Analysis

Impact Sets

Mining

Ranked lists of changes
Change Impact:
propagates along any (and only) dynamic paths that pass through the change  (Law et al. ICSE’03)

Impact Sets:

\[ C_A \Rightarrow M_A, M_D, M_Q, \ldots \]

\[ C_B \Rightarrow M_B, M_D, M_N, \ldots \]

\[ \ldots \]
Impact Sets:

\[ C_A \Rightarrow M_A, M_D, M_Q, \ldots \]
\[ C_B \Rightarrow M_B, M_D, M_N, \ldots \]

Ranked list:

- \( C_B \)
- \( C_A \)
- \( \ldots \)
Research Questions (RQs)

• RQ₁ - How effective is PerfImpact in finding inputs that likely expose performance regressions

• RQ₂ – Can PerfImpact effectively recommend changes likely responsible for performance regressions
Research Question 1

How effective is PerfImpact in finding inputs that likely expose regressions

PerfImpact vs. Random

H₀: There is no statistically significant difference between PerfImpact and Random
Research Question 2

How effective is PerfImpact in identifying regression-inducing code changes

• Inject nine artificial code changes
• Extract real code changes
Experimental Design

Agilefant
V3.2 vs V3.3
V3.2 vs V3.5

JPetStore
V3.0.0 vs V4.0.5

Total: 187 real changes + 9 artificial changes
RQ₁ – Finding Regression-Specific Inputs

Agilefant V3.2 vs V3.3

$p < 1.23e-296 \Rightarrow$ null hypothesis is rejected
RQ₂ - Performance regression in a real code change

Code change: ProjectBusinessImpl.retrieveLeafStories()

Smaller values imply higher ranks
RQ₂ - Performance regression in a real code change

Code change: ProjectBusinessImpl.retrieveLeafStories()
RQ2 - Performance regression in a real code change

Code change: ProjectBusinessImpl.retrieveLeafStories()

```java
public List<StoryTO> retrieveLeafStories(int projectId, StoryFilters filters) {
    .......
    for (Story leafStory : leafStories) {
        StoryTO tmp = new StoryTO(leafStory); .......
        Set<Task> tasks = new HashSet<Task>();
        for (Task task : tmp.getTasks()) {
            TaskTO taskTO = new TaskTO(task);
            tasks.add(taskTO);
        }
        tmp.setTasks(tasks);
    }
    return leafStoriesWithRank;
}
```
Mining Performance Regression Inducing Code Changes in Evolving Software

Qi Luo  
College of William and Mary  
qluo@cs.wm.edu

Denys Poshyvanyk  
College of William and Mary  
denys@cs.wm.edu

Mark Grechanik  
University of Illinois at Chicago  
drmark@uic.edu

ABSTRACT
During software evolution, the source code of a system frequently changes due to bug fixes or new feature requests. Some of these changes may accidentally degrade performance of a newly released software version. A notable problem of regression testing is how to find problematic changes.

1. INTRODUCTION
Performance is an important metric of software quality \cite{60, 43}, whereas performance testing is a vital activity that developers routinely perform during software development and maintenance to ensure quality \cite{19}. During software evolution, a number of code changes are committed, and these changes are inevitably accompanied by performance implications.

Online appendix:  
http://www.cs.wm.edu/semeru/data/MSR16-PerfImpact/
Experimental Design

Agilefant  
V3.2 vs V3.3  
V3.2 vs V3.5

IPetStore  
V3.0.0 vs V4.0.5

Total: 187 real changes + 9 artificial changes
Experimental Design

- Agileant
- JPetStore

PerfImpact

- PerfImpact is effective in finding specific inputs that expose performance regressions
- PerfImpact is effective in identifying regression-inducing code changes

Total: 187 real changes + 9 artificial changes
Additional Slides for Questions
GAs – Independent Variables

• Crossover rate – 0.3
• Mutation rate – 0.1
• Number of individuals per generation – 30

Termination Criterion

Maximum limit for the number of generations – 30
Average fitness value of every individual in one generation
**RQ₁ – Finding Regression-Specific Inputs**

Agilefant V3.2 vs V3.5

$p = 1.37e^{-236} \Rightarrow$ null hypothesis is rejected
RQ_1 – Finding Regression-Specific Inputs

JPetStore V3.0.0 vs V4.0.5

Difference in Elapsed Execution Time

Generations

\( p = 2.64e^{-198} \Rightarrow \) null hypothesis is rejected
RQ₂ - Performance regression in a real code change

Code change: StoryHierarchyBusinessImpl.calculateStoryTreeMetrics()
RQ\textsubscript{2} - Performance regression in a real code change

Code change: StoryHierarchyBusinessImpl.calculateStoryTreeMetrics()
RQ_2- Performance regression in a real code change

Code change: StoryHierarchyBusinessImpl.calculateStoryTreeMetrics()
RQ₂ - Performance regression in an artificial code change
RQ₂ - Performance regression in an artificial code change

![Graph showing total execution time for impact set vs number of users](image)

- $V_{i+1}$
- $V_i$

Number of Users

Total execution time for impact set