Gesellschaft für Informatik e.V. (GI) GI. (Hrsg.): INFORMATIK 2021,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2021 1819

Testselektion für Performanzregressionsbenchmarks in CI-Prozessen

David Georg Reichelt[1] Stefan Kühne[2] Wilhelm Hasselbring[3]

Abstract: Um Performanzregressionen zu finden, werden in Softwareprojekten Performanzregressionsbenchmarks (PRB) eingesetzt. Die Ausführung der PRBs ist zeitaufwändig und wird deshalb oft unregelmäßig, bspw. nach jedem Release, ausgeführt. Dadurch können Regressionen übersehen werden. Darüber hinaus ist die Ursachenanalyse für länger vergangene Regressionen schwer. Durch Quelltextänderungen verursachte Performanzänderungen können schneller gefunden werden, indem nur die PRBs ausgeführt werden, die geänderten Quelltext aufrufen. Wir stellen eine Erweiterung des Testwerkzeugs Peass vor, mit dem die Regressionstestselektion für PRBs möglich wird. Hierbei werden PRBs unterstützt, die mit dem Benchmarkingframework JMH[4] implementiert sind. Wir evaluieren unsere Erweiterung anhand der PRBs des Anwendungsservers jetty[5] und zeigen, dass Peass die Benchmarkausführungsdauer um 97,9 % reduziert und alle von den PRBs abgedeckten Regressionen findet.

1 Einleitung

Eine Herausforderung beim Testen moderner Softwaresysteme ist die Sicherstellung der Einhaltung von Performanzanforderungen. Ineffiziente Implementierungen führen zu erhöhter Rechenzeit, zu einem schlechteren Nutzererlebnis und zu einem erhöhten Energieverbrauch. Durch Lasttests und Benchmarks kann die Performanz einer Anwendung geprüft und mit der vorherigen Version verglichen werden. Diese Ansätze gewinnen in modernen Softwaresystemen an Bedeutung, um bspw. die Performanz von Softwarekomponenten in automatisch deployten containerbasierten Microservices oder akkubetriebenen Endgeräten zu überprüfen. Durch Performanzregressionsbenchmarks (PRB), die bspw. in JMH implementiert werden, wird es möglich, Regressionen in regelmäßig stattfindenden Tests zu identifizieren. Als Regressionen werden dabei Verschlechterung der Performanz desselben Anwendungsfalls gegenüber der Vorgängerversion betrachtet.

Moderne Software wird oft auf virtuellen Maschinen, wie der Java Virtual Maschine, ausgeführt. Die Performanz in virtuellen Maschinen wird von nichtdeterministischen Effekten wie Just-in-Time-Compilation, Garbage Collection und Speicherfragmentierung beeinflusst [GBE07]. Um Performanzänderungen zuverlässig zu bestimmen sind daher

[1] Universität Leipzig, dg.reichelt@uni-leipzig.de
[2] Universität Leipzig, kuehne@uni-leipzig.de
[3] Universität Kiel, hasselbring@email.uni-kiel.de
[4] Java Measurement Harness, https://openjdk.java.net/projects/code-tools/jmh/
[5] https://github.com/eclipse/jetty.project

Wiederholungen der gemessenen Workloads nötig. Aufgrund der nötigen Wiederholungen benötigt der Messprozess für PRBs in 38 % aller Projekte über vier Stunden [St17]. Messungen finden daher oft nur unregelmäßig, bspw. nächtlich [WEH15] oder zu Releases, statt. Meist existieren dabei mehrere Benchmarks und nicht alle Benchmarks rufen den gesamten Quelltext auf. Um Regressionen zu erkennen, ist es ausreichend, die Benchmarks auszuführen, deren aufgerufener Quelltext geändert wurde.

Das bestehende Werkzeug Peass (**Pe**rformanz**a**nalyse von **S**oftware**s**ystemen)[6] automatisiert die Regressionstestselektion, die Messung und die Ursachenanalyse für Performanztests, die bisher aus transformierten JUnit-Tests erzeugt werden. In dieser Arbeit stellen wir eine Erweiterung von Peass vor, mit der es möglich wird, die Messung von in JMH implementierten PRBs zu beschleunigen.

Dieser Beitrag ist folgendermaßen gegliedert: Eingangs wird eine Einführung in das bestehende Werkzeug Peass gegeben. Anschließend wird dargestellt, welche Erweiterungen für die Unterstützung der Ausführung von JMH-Benchmarks durch Peass benötigt werden. Darauf basierend wird durch eine Fallstudie evaluiert, wie effizient Peass die Regressionstestselektion für PRBs durchführen kann. Danach wird der hier verfolgte Ansatz zur Regressionstestselektion mit verwandten Arbeiten verglichen. Abschließend wird eine Zusammenfassung gegeben.

2 Peass

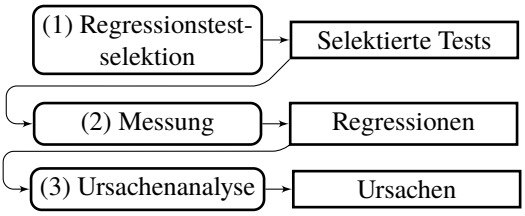

Abb. 1: Ansatz von PeASS

Um Performanzänderungen zu erkennen, ist es notwendig, gleiche Workloads in verschiedenen Versionen zu messen. Da die Implementierung und Wartung von Benchmarks und Lasttests aufwändig ist, sind diese nur selten in Quelltextrepositories vorhanden [St17]. Um dennoch die Performanz von Software in verschiedenen Versionen zu vergleichen, misst Peass die Performanz von Unittests [RKH19]. Unittests sind nicht immer für die Performanzmessung nutzbar, da Testwerkzeuge wie Mocking das Ergebnis verfälschen können oder das Testen von Grenzfällen nicht repräsentativ für die Gesamtanwendung sein kann. Da Unittests in vielen Projekten vorhanden sind und meist eine hohe Abdeckung des Quelltextes aufweisen, sind sie dennoch oft die einzige Quelle für messbare Workloads und werden daher in Peass genutzt.

Peass besteht aus drei Schritten, die in Abbildung 1 dargestellt sind: (1) In neuen Commits werden oft nur kleine Teile der Software geändert. Tests, die den geänderten Teil der Software

[6] Verfügbar unter `https://github.com/DaGeRe/peass`, Jenkins-Plugin unter `https://github.com/DaGeRe/peass-ci`

nicht untersuchen, müssen daher nicht getestet werden. Peass führt eine **Regressionstestselektion** aus, die bestimmt, welche Tests in der aktuellen Version eine geänderte Performanz haben können. Hierfür werden (ggf. vor dem neuen Commit) die Ausführungstraces aller Tests bestimmt. Wird ein neuer Commit erstellt, kann durch statische Quelltextanalyse auf Basis der geänderten Methoden und Klassen ermittelt werden, welche Tests Performanzänderungen aufweisen können. Durch erneute Ermittlung der Traces und Vergleich der aufgerufenen Methoden kann die Anzahl der Tests mit potentiellen Performanzänderungen weiter reduziert werden. (2) Anschließend wird durch die **Messung** bestimmt, welche Tests eine Performanzänderung besitzen. Hierfür werden nacheinander für jede Version eine zu definierende Anzahl VMs gestartet. In jeder VM wird zuerst eine zu definierende Anzahl von Aufwärmiterationen ausgeführt. Anschließend werden Messiterationen ausgeführt. Die Mittelwerte der Messausführungen bildet anschließend die Basis für den Vergleich der Messwerte. (3) Abschließend wird eine **Ursachenanalyse** durchgeführt, bei der für jeden Test durch Messung aller Knoten im Aufrufbaum ermittelt wird, welche Methode(n) die Performanzänderung verursachen.

3 Unterstützung von JMH-Benchmarks

Performanzregressionen können im Produktivbetrieb durch den wiederholten Aufruf ähnlicher Workloads entstehen. Diese Workloads können ggf. nicht durch transformierte Unittests abgedeckt werden, bspw. wenn Unittests einen höheren Anteil von Grenzfällen enthalten als die Workloads im Produktivbetrieb. Daher implementieren Entwickler bzw. Tester Benchmarks und Lasttests, anhand derer Regressionen vor der Überführung in den Produktivbetrieb festgestellt werden können. Um Performanzregressionen in Peass auch durch Benchmarks erkennen zu können, wurde Peass für die Nutzung anderer Workloaddefinitionen erweitert. Um andere Workloaddefinitionen wie Benchmarks und Lasttests messen zu können, wurden generische Schnittstellen extrahiert. Die bestehende Implementierung der Messung von JUnit-Tests in Maven-und Gradleprojekten wurde an die Schnittstellen angepasst. Darüber hinaus wurden die Schnittstellen prototypisch für JMH-Benchmarks in Maven-Projekten implementiert.

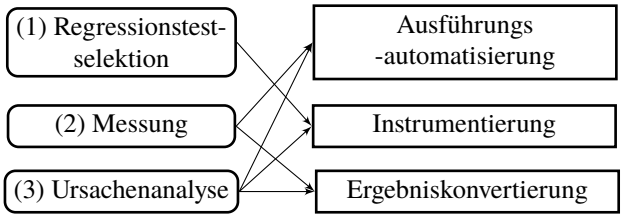

Abb. 2: Erweiterungen von Peass

Für die Unterstützung verschiedener Typen von Workloaddefinition wurden folgende Anforderungen an Peass festgestellt: (1) **Automatisierung** der Ausführung: Für die Messung und Ursachenanalyse ist es nötig, die Workloads zu ermittels, zu prüfen, ob die aktuelle Version lauffähig ist und die Workloads selbst mit der definierten Konfigurationen, bspw. Anzahl von VM-Ausführung, Iterationen und Warmup

auszuführen. (2) **Instrumentierung** der Ausführung: Um die Regressionstestselektion und die Ursachenanalyse auszuführen, ist es nötig, Ausführungstraces zu erzeugen. Dies geschieht für Unittests durch Instrumentierung mit Kieker-Probes [Hv20]. Die Instrumentierung kann durch direkte Änderung des Quelltextes oder eine Weaving-Technologie wie AspectJ geschehen. Diese Instrumentierung muss für den jeweiligen Workloaddefinitionstyp und dessen Quelltextpfade angepasst werden. (3) **Transformation der Messergebnisse**: Peass verwaltet Ergebnisse einzelner VM-Ausführungen in einem XML-Datenformat und speichert die Kieker-Ergebnisdaten in einer standardisierten Ordnerstruktur. Die Messergebnisse von anderen Workloadausführungen müssen in das Peass-Format überführt werden.

Um die Regressionstestselektion, die Testausführung und die Ursachenanalyse von JMH-Benchmarks mit Peass zu automatisieren, wurden die Schnittstellen folgendermaßen implementiert: (1) **Automatisierung** der JMH-Benchmark-Ausführung: Für die Ermittlung der Workloads wurde das Parsing so angepasst, dass `org.openjdk.jmh.annotations.Benchmark` implementierende Klassen gefunden werden. Um JMH-Benchmarks auszuführen, ist es notwendig, diese vorab zu kompilieren und anschließend eine generierte `jar` mit den definierten Konfigurationsparametern aufzurufen. Für die Ermittlung der Lauffähigkeit und die Generierung der jar konnte die bestehende Automatisierung der Maven-Ausführung weitergenutzt werden. Darüber hinaus wurde die Ausführung der Benchmark-jar mit der übergebenen Konfiguration implementiert. (2) **Instrumentierung der Messausführung**: Die Instrumentierung von Maven-Projekten konnte weitestgehend weiterverwendet werden. (3) **Transformation der Messergebnisse**: JMH erzeugt Ausgabedaten in einem eigenen JSON-Datenformat. Die Messergebnisse werden dementsprechend aus dem JMH-JSON-Format in das Peass-XML-Format transformiert.

4 Fallstudie

Um zu überprüfen, ob Peass-JMH in der Lage ist, alle Performanzänderungen zu finden, wurden die Benchmarks der aktuellsten 100 Versionen des Anwendungsservers jetty analysiert. Hierfür wurden die *herkömmliche Ausführung ohne Peass* und die *Ausführung mit Peass* untersucht. Alle Messungen wurden auf OpenJDK 11.0.10, laufend auf CentOS 7.9.2009 mit Intel E5-2620 v3 2.40GHz ausgeführt. Die Ausführung aller Benchmarks erfolgte mit 30 VMs [GBE07] und unter Nutzung der in der Jenkinsfile[7] definierten Parameter. Die Benchmarkausführungen wurden unter Nutzung eines slurm-Clusters verteilt. Die Ausführung der Benchmarks dauert durchschnittlich 15,6 Stunden.

Für die *herkömmliche Ausführung ohne Peass* wurden jede der 100 Versionen ausgecheckt. Anschließend wurden alle 52 Benchmarks ausgeführt. Darauf basierend wurde unter Nutzung des zweiseitigen T-Tests ermittelt, wo Performanzänderungen vorhanden sind.

[7] https://github.com/eclipse/jetty.project/blob/jetty-10.0.x/Jmh_Jenkinsfile

Die Ausführung der 5.200 Benchmarks (100 Versionen á 52 Benchmarks) würde ohne Parallelisierung 65 Tage benötigen.

Für die *Ausführung mit Peass* wurde Peass mit der Regressionstestselektion für jede Version ausgeführt. Durch die Regressionstestselektion wurden 98 Benchmark-Versionspaare selektiert. Diese ließen sich in 29,4 Stunden ausführen. Die Regressionstestselektion für alle 100 Versionen dauerte 2,9 h. Die gesamte Ausführungsdauer konnte also insgesamt um 97,9 % reduziert werden. Alle durch die vollständigen PRBs aufgedeckten Regressionen wurden auch durch die selektierten PRBs aufgedeckt.

5 Verwandte Arbeiten

Die nötigen auszuführenden Regressionstests können durch Regressionstestselektion, -priorisierung und -reduktion eingegrenzt werden [YH12]. Dabei bestimmt die Regressionstestselektion die auszuführenden Tests direkt, die Priorisierung ordnet Tests danach, ob eine Regression wahrscheinlicher ist und die Reduktion wählt Tests aus, deren Ausführung nicht nötig ist. In der Praxis haben sich bislang vor allem Methoden für die Regressionstestselektion funktionaler Tests etabliert, bspw. das Maven-Plugin Ekstazi[8] und das eclipse-Plugin Infinitest[9]. Diese Werkzeuge sind für die Regressionstestselektion für funktionale Tests optimiert und haben daher einen geringen Overhead. Allerdings selektieren sie mehr Tests als die Regressionstestselektion von Peass [RK18] und würden damit zu einer höheren Ausführungsdauer führen.

Verschiedene Arbeiten entwickeln Regressionstestselektionen bzw. -priorisierungen für Performanztests. In der Regel wird ein Kostenmodell entwickelt, das einzelnen Methodenaufrufen oder Quelltextkonstrukten Kosten zuordnet, anhand derer dann (wahrscheinliche) Performanzänderungen selektiert werden. Alcocer et al. [SABV16] entwerfen eine PRB-selektion für Pharo-Benchmarks. Das Kostenmodell basiert auf Analyse von Ausführungstraces. Benchmarks werden anhand der Häufigkeit des Vorkommens von Typen von Quelltextänderungen (bspw. Hinzufügen von Methodenaufrufen oder Schleifen) selektiert. Durch die Ausführung von 17 % aller Versionen können 83 % aller Regressionen identifiziert werden. Mostafa et a. [MWX17] entwickeln ebenfalls ein Kostenmodell, in dem die Ausführungsdauer einzelner Methoden erfasst wird. Anschließend werden Änderungen priorisiert, die Ausführungen von Schleifen über `Collections` hinzufügen oder ändern. Huang et al. [Hu14] entwickeln ebenfalls ein Kostenmodell, in dem Änderungen, die oft aufgerufene und teure Methodenaufrufe betreffen, als risikoreich eingeschätzt werden. Durch die Ausführung von maximal 22 % der Testfälle können in den untersuchten Fallstudien mindestens 87 % aller Performanzänderungen bestimmt. Eine Methode zur Regressionstestselektion für PRBs mit JMH existiert hingegen bislang nicht.

[8] http://ekstazi.org/
[9] https://infinitest.github.io/

6 Zusammenfassung und Ausblick

Peass ist ein Werkzeug, um Performanzmessungen im CI-Prozess durchzuführen. Peass enthält eine Regressionstestselektion zur Beschleunigung der Messung und eine Ursachenanalyse zur Untersuchung von Änderungen. In dieser Arbeit wird eine Erweiterung von Peass vorgestellt, die das beschleunigte Erkennen von Performanzänderungen in JMH-Benchmarks ermöglicht. Es wurde gezeigt, dass die Peass auf den letzten 100 Commits des Anwendungsservers jetty alle Testfälle selektiert, die potentielle Regressionen aufdecken können. Diese Arbeit ist ein Schritt hin zu **verlässlicher und effizienter Performanzmessung** im CI-Prozess äquivalent zum aktuellen Vorgehen für Unittests. Wir planen (1) durch weitere Fallstudien die Effizienz von Peass für JMH-Benchmarks vertieft zu betrachten, (2) weitere Spezifikationen von Workloads für Performanzmessung, wie JMeter-Benchmarks, zu unterstützen, und (3) durch den praktischen Einsatz von Peass zu evaluieren, inwiefern die Aufdeckung realer Performanzprobleme durch die Unittestbasierte Messung möglich ist.

Danksagung Diese Arbeit wurde durch das Bundesministerium für Bildung und Forschung im Rahmen des Projekts „Performance Überwachung Effizient Integriert" (*PermanEnt*, BMBF 01IS20032D) finanziert. Berechnungen für diese Arbeit wurden mit Ressourcen des Universitätsrechenzentrums Leipzig durchgeführt.

Literaturverzeichnis

[GBE07] Georges, Andy; Buytaert, Dries; Eeckhout, Lieven: Statistically rigorous java performance evaluation. ACM SIGPLAN Notices, 42(10), 2007.

[Hu14] Huang, Peng; Ma, Xiao; Shen, Dongcai; Zhou, Yuanyuan: Performance Regression Testing Target Prioritization via Performance Risk Analysis. In: ICSE. ACM, 2014.

[Hv20] Hasselbring, Wilhelm; van Hoorn, André: Kieker: A monitoring framework for software engineering research. Software Impacts, 5, 2020.

[MWX17] Mostafa, Shaikh; Wang, Xiaoyin; Xie, Tao: PerfRanker: prioritization of performance regression tests for collection-intensive software. In: ISSTA. ACM, 2017.

[RK18] Reichelt, David Georg; Kühne, Stefan: Better Early Than Never: Performance Test Acceleration by Regression Test Selection. In: Companion of ICPE. 2018.

[RKH19] Reichelt, D. G.; Kühne, S.; Hasselbring, W.: PeASS: A Tool for Identifying Performance Changes at Code Level. In: Proceedings of the 33rd ACM/IEEE ASE. ACM, 2019.

[SABV16] Sandoval Alcocer, Juan Pablo; Bergel, Alexandre; Valente, Marco Tulio: Learning from Source Code History to Identify Performance Failures. In: ICPE. ACM, 2016.

[St17] Stefan, P.; Horky, V.; Bulej, L.; Tuma, P.: Unit Testing Performance in Java Projects: Are We There Yet? In: Proceedings of ACM/SPEC ICPE 2017. ACM, S. 401–412, 2017.

[WEH15] Waller, J.; Ehmke, N. C.; Hasselbring, W.: Including Performance Benchmarks into Continuous Integration to Enable DevOps. SE Notes, 40(2), March 2015.

[YH12] Yoo, S.; Harman, M.: Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability, 22(2):67–120, 2012.