Schlussbericht

KMU-innovativ

Technologiebereich Datenwissenschaft, Informationstechnologien und Industrie 4.0

PermanEnt

Perfor**man**ce-Überwachung **E**ffizient I**nt**egriert

UNIVERSITÄT
LEIPZIG

Autoren: David Georg Reichelt, Matti Wilhelmi, Hannes Krauß, Behnaz Nabhan, Alex Korn, Claas Gaidies, Stefan Kühne

Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung und Forschung unter dem Förderkennzeichen 01IS20032 gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt bei den Autoren.

Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR)
Projektträger im DLR
Softwaresysteme und Wissenstechnologien
Rosa-Luxemburg-Str. 2
10178 Berlin

1 Kurzbericht

1.1 Projektziel

Die Performance von IT-Systemen, d. h. die Effizienz eines Softwaresystems hinsichtlich Zeit- und Ressourcenverbrauch, spielt in vielen Bereichen der Wirtschaft eine entscheidende Rolle. Sie wird maßgeblich durch die Implementierung bestimmt. Um die Performance einer Software zu verbessern bzw. trotz funktionaler Erweiterungen oder veränderter Hard- und Softwareumgebungen konstant zu halten, war das Ziel des Projekts **Per**for**man**ce-Überwachung **E**ffizient I**nt**egriert (*PermanEnt*), einen **messbasierten¸ Ansatz** zur Erkennung von Performanceunterschieden zu entwickeln und in reale Softwareentwicklungsprozesse zu integrieren.

Die Messungen der Performance von Software erfolgt dabei durch **transformierte Unittests**. Deren Performancemessung wird in den Continous-Integration-Prozess (CI) integriert, um regelmäßig und zeitnah Rückmeldungen über Veränderungen der Performance zu geben. Hierbei ist das Ziel, den bestehenden Peass-Ansatz[1] so auszubauen, dass er für eine regelmäßige Performancemessung in der CI nutzbar ist. Zentrale Teilziele waren dabei die allgemeine **Integration** des Peass-Prozesse in den CI-Prozess, die Implementierung einer effizienten **Ursachenanalyse** sowie die Anpassung der bestehenden **Regressionstestselektion** für die Nutzung in der CI.

1.2 Ablauf des Projekts

Das Projekt bestand aus drei Projektphasen, in denen der Schwerpunkt auf der **Integration** von Peass in den CI-Prozess, der Beschleunigung der **Ursachenanalyse** bzw. der Weiterentwicklung der **Regressionstestselektion** lag. Jede Phase umfasste die Erfassung der Anforderungen sowie Implementierungs- und Testaktivitäten. Zur Erfassung der Anforderungen wurden in einem hybriden, einem rein digitalen und einem rein physisch durchgeführten Workshop jeweils die Eigenschaften der zu untersuchenden Programme diskutiert. Die sich daraus ergebenden Anforderungen wurden in GitHub-Issues im Peass-CI-Projekt überführt. Während der Implementierung wurden die Anforderungen im Detail formuliert, in monatlichen Projekttreffen besprochen und gegebenenfalls in neuen GitHub-Issues dokumentiert. Die Implementierung erfolgte entweder durch einzelne Projektpartner oder in Zusammenarbeit.

Nachdem Peass-CI bei den Projektpartnern voll funktionsfähig war, wurde das Tool in zwei Fallstudien evaluiert: anhand des Entwicklungsprozesses des GeoMap-Immobilientools sowie der geofox-Repositories, die verschiedene Services im ÖPNV-Bereich anbieten.

Für die GeoMap-Fallstudie wurden alle 203 zwischen dem 1.1.2022 und 31.6.2022 aufgetretenen Commits vollständig mit Peass-CI analysiert. Dabei wurden 52 Commits identifiziert, bei denen die geänderten Quelltexte von Testfällen abgedeckt und daher gemessen wurden. In 19 dieser Commits wurden durch Peass-CI auffindbare Performanceänderungen entdeckt. Durch den Vergleich mit Lasttests konnte gezeigt werden, dass vier der fünf stärksten Performanceänderungen, die durch Unittests ermittelbar sind, auch in Lasttests messbar sind. Darüber hinaus wurde ein Performanceproblem in einem GeoMap-Service durch die systematische Untersuchung mit Lasttests entdeckt.

In den größten geofox-Repositories fanden im Zeitraum vom 1.1.2022 bis 31.6.2022 1027 Commits statt. Mit den verfügbaren Ressourcen konnten 27 Commits gemessen werden, bei denen sechs Performanceänderungen festgestellt wurden.

Die Fallstudien zeigen insgesamt, dass Unittests ein geeignetes Mittel sind, um Performanceänderungen mit geringem manuellen Aufwand zu identifizieren. Um auch kleine Performanceänderungen zu finden, sind allerdings umfangreiche Rechenressourcen erforderlich. Darüber hinaus können bestimmte Problemtypen, wie solche, die erst nach langer Laufzeit der Anwendung auftreten, nicht durch Unittests erkannt werden.

Begleitend zu den Projektarbeiten fand zudem die Dissemination der Ergebnisse in Wissenschaft und Praxis statt. Hierbei wurden die Projektergebnisse durch wissenschaftliche Präsentationen auf Fachtagungen sowie durch Tooldemonstrationen und die Vorstellung der GeoMap-Fallstudie auf Anwenderkonferenzen mit interessiertem Fachpublikum diskutiert.

[1] (**Pe**rformance **a**nalysis of **s**oftware **s**ystems, Ansatz zur Erkennung von Performanceunterschieden, siehe `https://github.com/DaGeRe/peass` [5]

1.3 Wesentliche Ergebnisse und deren Nutzen

Im Rahmen des Projekts wurde das Jenkins-Plugin Peass-CI entwickelt, das die Erkennung von Performanceänderungen bestehender Unittests und JMH-Benchmarks durch Messungen basierend auf dem Peass-Ansatz ermöglicht. Im Projekt wurden drei wesentliche Ergebnisse erzielt: Die **Integration von Peass** in CI-Prozesse und damit verbunden die Nutzbarmachung für die Projekte der Anwendungspartner, die **Verbesserung der Regressionstestselektion** und die **Beschleunigung der Ursachenanalyse** von Peass, um sie regelmäßig in den Build-Prozess einzubeziehen.

Um Peass in CI-Prozesse **integrieren** zu können, wurde das Jenkins-Plugin Peass-CI implementiert. Um Peass dabei für den Einsatz in gängigen Entwicklungsprozessen nutzbar zu machen, waren sowohl konzeptionelle als auch technische Erweiterungen an Peass erforderlich. Die ursprüngliche Peass-Methode wurde konzeptionell angepasst, um Nightly Builds, die Weitergabe von Credentials innerhalb des Plugins, Mocking, parametrisierte Tests und die Messung von Unittests, die auf Docker-Containern basieren, zu unterstützen. Darüber hinaus wurden Lösungen für den Umgang mit großen Datenmengen, die bspw. bei großen Aufruftraces entstehen, erprobt und implementiert. Für die technische Kompatibilität wurde der Gradle-Buildprozesse automatisiert und die Kompatibilität zur Nutzung des Spring-Frameworks hergestellt. Für die realistische Messung von Android-Buildprozessen wurde darüber hinaus der Messprozess so angepasst, dass emulierte Android-Tests ausgewertet werden können.

Die **Regressionstestselektion** ermöglicht die Auswahl derjenigen Tests, die von Quelltextänderungen betroffen sind. Durch die ausschließliche Messung der selektierten Tests kann die Messzeit erheblich reduziert werden. Im Projekt wurde die Regressionstestselektion von Peass durch zwei Methoden ergänzt: Die *abdeckungsbasierte*, die *messbarkeitsbasierte* und die *regelbasierte Testselektion*.
Die *abdeckungsbasierte Testselektion* ermöglicht es, Tests so auszuwählen, dass geänderte Methoden mindestens einmal gemessen werden. Dadurch kommt es zu keiner unnötigen Wiederholung der Ausführung derselben Methode. Die *messbarkeitsbasierte Testselektion* ermöglicht es, nur die Tests auszuwählen, die sich für die Unittestmessung technisch eignen. Hierfür werden die Tests mehrfach ausgeführt und es wird geprüft, ob diese Ausführung fehlerfrei durchführbar ist. Die *regelbasierte Testselektion* ermöglicht es, ausschließlich Tests mit definierten `JUnit`-Rules, bspw. der `DockerRule`, auszuführen. Dadurch wird es möglich, eigene Konfigurationen für einzelne Regeln festzulegen.

Die **Ursachenanalyse** ermöglicht die Identifikation derjenigen Knoten eines Aufrufbaums, die Performanceunterschiede verursachen. Um die Ursachenanalyse zu beschleunigen, wurde evaluiert, inwiefern die Auswahl spezifischer Teilbäume für die Messung eine effizientere Ursachenanalyse ermöglicht. Je nach Anwendungsfall sind unterschiedliche Auswahlstrategien effizienter, daher wurden alle Strategien in Peass-CI implementiert. Durch die Feingranularität der Unittests waren bestehende Ansätze der Ursachenanalyse nur bedingt einsetzbar, da der Overhead der Messung einzelner Methoden die Messung kleiner Performanceunterschiede unmöglich macht. Durch die Untersuchung verschiedener Optionen zur *Overheadreduktion* und deren Implementierung in Kieker wurde es darüber hinaus möglich, die Ursache kleinerer Performanceänderungen effizienter zu finden.

Insgesamt wurde ein Werkzeug geschaffen, das es ermöglicht, Performanceunterschiede automatisiert im CI-Prozess zu erkennen und zu untersuchen. Dadurch wird der Aufwand für die Problemanalyse reduziert und Performanceprobleme können genauer untersucht werden. Ohne Peass-CI besteht ein größeres Risiko, dass Performanceprobleme aufgrund des zu hohen Aufwands unbeachtet bleiben.

2 Einleitung

2.1 Motivation

Die Effizienz von Softwaresystemen in Bezug auf Zeit- und Ressourcenverbrauch wird als Softwareperformance bezeichnet [1]. Inakzeptables Antwortzeitverhalten kann auf verschiedene Ursachen zurückzuführen sein, wie Architektur, konkrete Implementierung oder Deployment, d.h. die Verteilung auf der genutzten Hardware oder der Hardware selbst. Auf der Implementierungsebene können ineffiziente Algorithmen, falsche Nutzung von Programmschnittstellen (APIs) oder Ressourcenkonflikte zu Performanceproblemen, d.h. zur Nichterreichung von Performanceanforderungen, führen. Diese Performanceprobleme können während der Entwicklung oder durch die Konfiguration im Betrieb der Software entstehen.

Die Behebung von Performanceproblemen erfordert in der Regel mehr Entwicklerzeit als die Behebung anderer Arten von Fehlern [2, 3]. Aktuelle Trends verschärfen diese Situation weiter: Die zunehmende Verwendung von Big Data erhöht die Datenmengen und damit auch den Ressourcenbedarf, was es schwieriger macht, realistische Testszenarien auf einem Entwicklerrechner nachzuvollziehen. Das Internet der Dinge, das physische Reaktionen von Systemen bedingt, erfordert immer häufiger Echtzeitfähigkeit von Systemen. Durch Containerisierung und die skalierbare Hinzunahme von Hardwareressourcen werden Probleme durch mehr Ressourcen gelöst, statt die Ursachen im Quelltext anzugehen. Dies führt jedoch zu erhöhten Kosten und löst Performanceprobleme nur vorübergehend, bis die Kundennachfrage weiter wächst oder neue Funktionen hinzugefügt werden.

Die dauerhafte und nachvollziehbare Einhaltung von Performanceanforderungen erfordert eine hohe Transparenz bezüglich Auswirkungen von Systemveränderungen. Dabei sollte das Performanceverhalten im Betrieb bzw. im Lasttest für den Entwickler nachvollziehbar und kontinierlich überprüfbar sein. Derzeit ist dies nur durch aufwändig zu implementierende Benchmarks und ggf. hinzuzufügende faktorielle Experimente, die das Systemverhalten reproduzieren, möglich.

Im Bereich der funktionalen Korrektheit hat sich gezeigt, dass Unittests dazu beitragen können, Fehler schneller zu finden, die Fehlerursache einzugrenzen und manuelle Tests zu reduzieren. Das Ziel von *PermanEnt* ist es, ähnlich wie bei Unittests, Performanceprobleme auf Quelltextebene zu behandeln, indem bestehende Unittests in Performance-Unittests (PUTs) umgewandelt werden. Dabei gibt es drei Herausforderungen im Vergleich zum aktuellen Stand der Technik:

Regelmäßige Ausführung Die Performance verhält sich nichtdeterministisch und hängt von verschiedenen Faktoren wie dem Aufwärmzustand der CPU und (Java)-VM, Speicherfragmentierung und Threadscheduling ab. Daher erfordert die Messung der Performance viele Systemressourcen. Aktuelle Performance-Engineering-Aktivitäten, wie die Ausführung von Lasttests oder Benchmarks, werden daher oft mit einem wasserfallartigen Vorgehen durchgeführt [4]. Eine Herausforderung ist besteht darin, eine Ausführungsumgebung für PUTs zu schaffen, die mit vertretbarem Ressourcenaufwand schnelle Rückmeldungen über Performanceänderungen, die durch neue Commits ausgelöst werden, gibt.

Fehleruntersuchung Ähnlich wie bei funktionalen Fehlern ist das Verhalten einer Software hinsichtlich ihres Ressourcenverbrauchs schwer zu verstehen. Aufgrund des Nichtdeterminismus ist auch die Reproduktion von Performanceproblemen auf Entwicklerrechnern schwieriger. Die Eingrenzung einer Performanceänderung auf einen PUT bzw. Benchmark und einen Commit ist oft nicht ausreichend, da Tests oft komplexes Systemverhalten testen und Commits oft komplexe Änderungen enthalten. Eine Herausforderung besteht daher darin, die Methode im Quelltext zu identifizieren, die eine Performanceveränderung verursacht.

Selektion relevanter Tests Softwarerepositories enthalten oft komplexe Produkte, bei denen die Änderungen an einer Komponente keine Auswirkungen auf die Performance eines Teils der anderen Komponenten haben kann. Aufgrund von Abhängigkeiten zwischen den Komponenten sind jedoch auch Auswirkungen von Quelltextänderungen einer Komponente auf die Performance anderer Komponente möglich. Eine Herausforderung ist daher, die relevanten Tests auszuwählen, um für die ressourcenintensive Ausführung nur diese zu untersuchen.

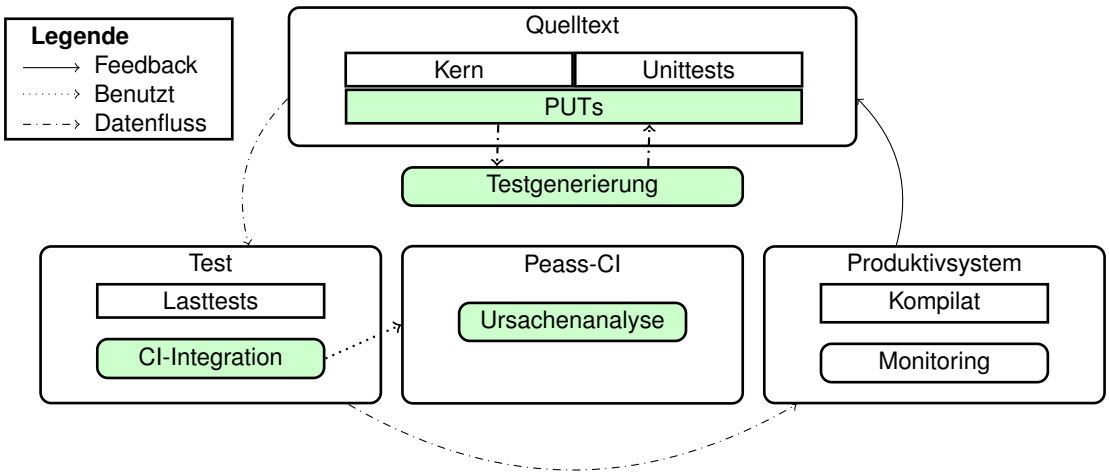

Abbildung 1: *PermanEnt*-Ansatz

2.2 Ansatz

Um eine **regelmäßige Ausführung** zu ermöglichen, wurde in *PermanEnt* eine Ausführungsumgebung für die Transformation und Messung von JUnit-Tests in Jenkins-Integrationsserver entwickelt. Dadurch können Änderungen des Performanceverhaltens dauerhaft durch eine **PUT-CI-Integration** überwacht werden. Dabei wird auf den Vorarbeiten zu Peass aufgebaut, die eine Regressionstestselektion und eine Messumgebung für PUTs definieren.

Zur Unterstützung der **Fehleruntersuchung** bei funktionalen Tests wird der geworfenen Fehler bzw. die geworfene Ausnahme genauer untersucht. Je nach Komplexität des Fehlers ist Debugging notwendig. Um das Verstehen äquivalent zum Verstehen der Ursachen funktionaler Änderungen zu unterstützen, wurde eine **Ursachenanalyse** erforscht, die zeitnah den Teil des Aufrufbaums ermittelt, der eine Verlangsamung verursacht. Durch die Reduktion des Messoverheads konnte die Ermittlung von Änderungsursachen auch für kleine Workloadgrößen ermöglicht werden, die typischerweise in Unittests auftreten. Im funktionalen Test ersetzt die Information über den geworfenen Fehler nicht das Debugging zum genaueren Problemverständnis. Äquivalent ersetzt die Ursachenanalyse nicht das Profiling, sondern ergänzt es.

Die Ausführung von PUTs erfordert erheblich mehr Zeit als die Ausführung von Unittests, da der Workload wiederholt werden muss. Deshalb wurde im Rahmen von *PermanEnt* eine erweiterte **Testselektion** erforscht, die eine PUT-Testsuite mit reduzierter Redundanz aus einer bestehenden Testsuite für einen definierten Commit erzeugt.

Abbildung 1 fasst den *PermanEnt*-Ansatz zusammen.

3 Projektergebnisse nach Arbeitspaketen

In *PermanEnt* wurden drei zentrale Arbeitspakete bearbeitet: (1) die **P**erformance-**U**nit**t**ests-**CI**-Integration, (2) die Erstellung der **Ursachenanalyse** und (3) der **Testselektion**. Vor Beginn der Bearbeitung wurden die Anforderungen gemeinsam mit den Projektpartnern erfasst. Während der Bearbeitung der Arbeitspakete wurden die **Anforderungen** kontinuierlich verfeinert und die entstehenden Werkzeuge mit den Anwendungspartnern getestet und in den Produktivbetrieb überführt. Parallel dazu erfolgte die **Ergebnisdiskussion**. Abschließend bzw. parallel zu den erfolgten Weiterentwicklungen erfolgte eine **Evaluierung**. Die Arbeitspakete sind in Abbildung 2 dargestellt und werden im Folgenden im Detail erläutert.

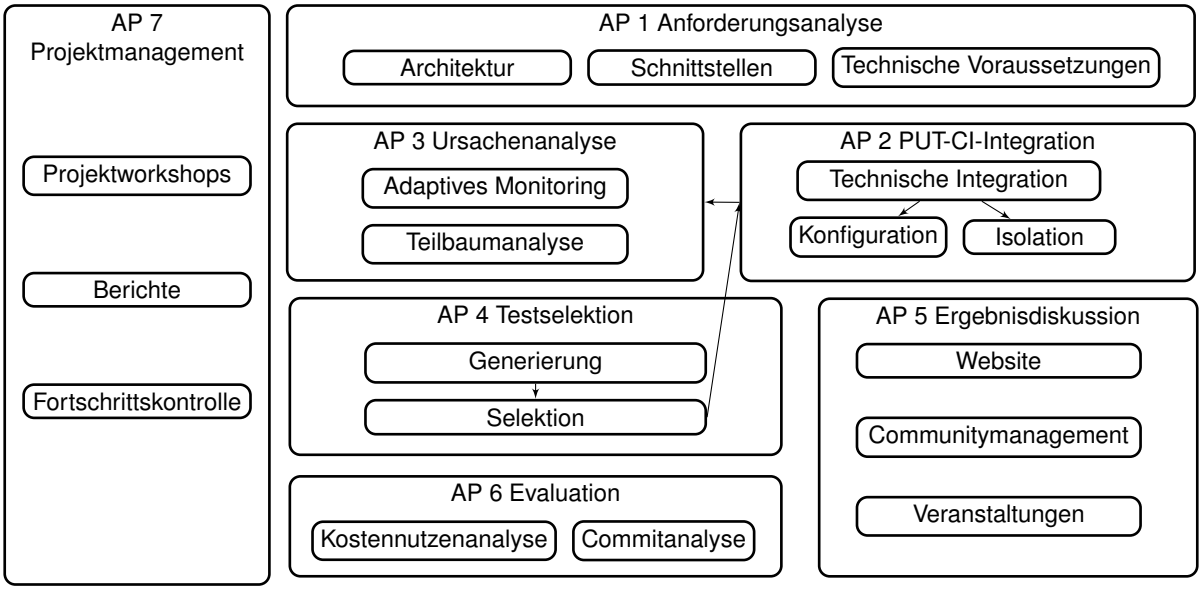

Abbildung 2: Arbeitspakete *PermanEnt*

AP 1 – Anforderungsanalyse und Architektur

AP 1.1 – Generelle Anforderungen und Architektur

Toolunterstützung In einem initialen Anforderungsworkshop wurden die Detailanforderungen der Partner hinsichtlich der Toolunterstützung erhoben. Für die Performancemessung ist es nötig, den Buildprozess zu automatisieren und die Unittests in Performance-Unittests umzuwandeln. Für die zentrale Messung der Performance in jedem Commit ist es nötig, die Messung in einem Integrationsserver umzusetzen.

Bei den Partnern werden maven (evermind) und Gradle (HBT, IOTIQ) als Buildtools sowie JUnit 4 (evermind, IOTIQ) und JUnit 5 (HBT) für Unittests eingesetzt. Die kontinuierliche Integration erfolgt bei allen Anwendungspartnern mit einem Jenkins-Integrationsserver. Daher wurde ein Jenkins-Plugin sowie die Unterstützung von maven, Gradle und JUnit 5 in Peass als zentrale Anforderungen definiert.

Maven und JUnit 4 wurden bereits vor Projektbeginn unterstützt. Die Hauptanforderung war daher, die Buildautomatisierung um Gradle zu erweitern, die Testausführung mit JUnit 5 zu implementieren sowie das Jenkins-Plugin zu entwerfen. Dies ist in Abbildung 3 zusammengefasst.

Management-Overview und Testverlauf Die natürliche Darstellung in CI-Servern ist die buildbasierte Darstellung von Analyseergebnissen, bspw. die Darstellung der Ergebnisse des Kompilierens oder des Testen eines Softwareprojekts in einem definierten Commit. Während der Projektlaufzeit stellte sich heraus, dass in der Regel wenige messbare Performanzänderungen auf eine hohe Anzahl von Commits verteilt sind, bspw. enthielt GeoMap 19 Performanzänderungen auf 203 Commits. Werden nur die Builds angezeigt, müsste der Entwickler 203 Commits durchsuchen, um die 19 Performanzänderungen zu finden. Daraus ergibt sich die Anforderung, Performanzänderungen aus Builds in einem Management-Überblick zusammenzufassen und durch den Benutzer, d. h. den Entwickler oder Teamleiter, klassifizieren zu lassen.

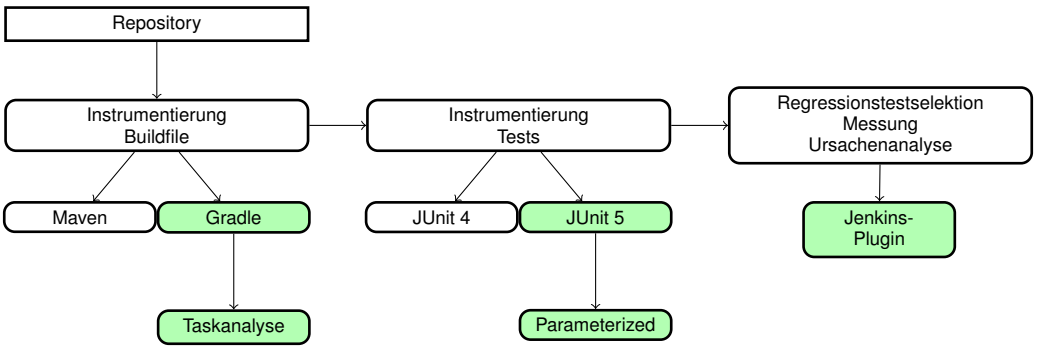

Abbildung 3: Gesamt-*PermanEnt*-Toolanforderungen

***PermanEnt*-Tools Architektur** *PermanEnt* baut auf dem bestehenden Tool Peass [5] auf. Peass benutzt das Tool KoPeMe [6] zur Messung der Performanz eines Unittests innerhalb einer VM und Kieker [7, 8] zur Rekonstruktion der Ausführungstraces und zur Messung der Performanz einzelner Methodenaufrufe. Zur Reduktion des Overheads wurde im Projektkontext das Repository kieker-source-instrumentation[2] erstellt [9].

Peass besteht aus dem `dependency`-Modul zur Regressionstestselektion, dem `measurement`-Modul zur Performancemessung und dem `analysis`-Modul zur Visualisierung der Ergebnisse. Um andere Buildprozesse zu unterstützen, wurden im Projekt die Module `peass-jmh` sowie das Modul `peass-ant` (im Rahmen einer Bachelorarbeit[3]) erstellt. Die Erstellung eines Repositories für die Nutzung von JMeter-Lasttests (`peass-jmeter`) ist geplant.

Die Nutzbarkeit dieser Projekte für Entwickler in Jenkins wurde durch das im Projekt entwickelte Plugin peass-ci[4] ermöglicht.

Während des Projektes wurden Demonstrationsanwendungen für Performanceänderungen in Buildprozessen, die den Buildprozessen der Projektpartnern ähneln, erstellt. Diese umfassen u.a. die `pure`, `android` und `spring-boot`-Demo. Diese sind gleichzeitig in kontinuierliche Buildprozesse eingebunden, so dass funktionale Probleme in den Implementierungen direkt bemerkt werden können.

Abbildung 4 fasst die Architektur zusammen.

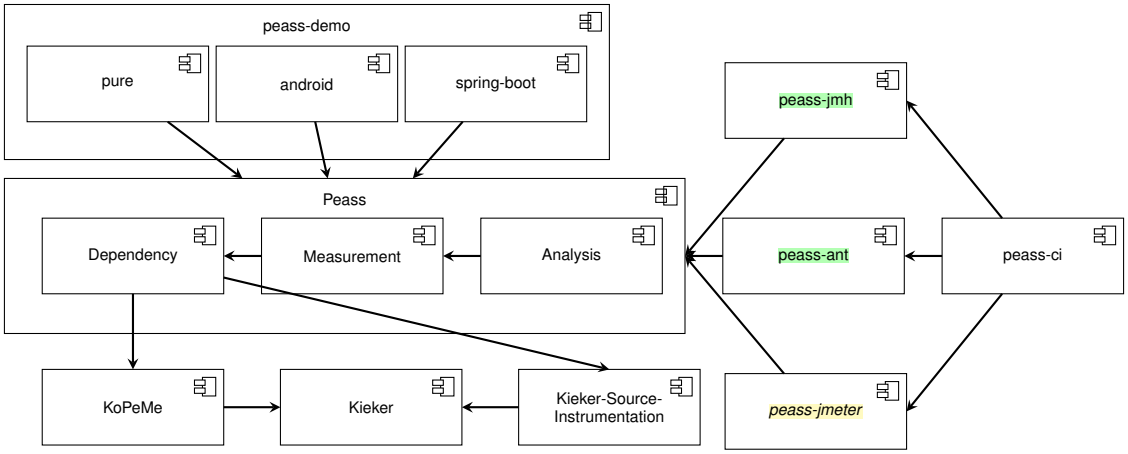

Abbildung 4: Implementierte Architektur

AP 1.2 – Schaffen der technischen Voraussetzungen

Ursprünglich war geplant, zusätzliche, wiederholbare Testfälle in definierten Ausführungsumgebungen für die einzelnen Teilprojekte zu erstellen. Aufgrund der Vielzahl der zu untersuchenden Unittests wurden keine weiteren Testfälle und keine weiteren Ausführungsumgebungen untersucht. Stattdessen wurden die Performance der bestehenden Unittests auf den zur Verfügung stehenden CI-Servern verglichen.

[2] `https://github.com/kieker-monitoring/kieker-source-instrumentation`
[3] Ergebnisse der Bachelorarbeit: `https://doi.org/10.5281/zenodo.6500484`
[4] `https://plugins.jenkins.io/peass-ci/`

AP 1.3 – Spezifische Toolanforderungen und -architektur

Die spezifischen Toolanforderungen wurden für die untersuchten Projekte GeoMap, Geofox und MobiVisor erhoben.

GeoMap

Untersuchte Teilprojekte Die Software von GeoMap besteht aus folgenden Komponenten: (1) den Modulen zum automatischen und moderierten Erheben und Aufbereiten aller in GeoMap verwalteter Daten, (2) der Datenbank und (3) den Module zur Bereitstellung der Daten. Die Prozesse der Module der Erhebung und Aufbereitung der Daten sind entweder nicht zeitkritisch oder die zeitkritischen Faktoren befinden sich außerhalb des mit *PermanEnt* untersuchbaren Bereiches. Die Datenbank ist ebenfalls nicht direkt durch *PermanEnt* untersuchbar. Das Auslesen der Daten aus der Datenbank kann indirekt durch die Messung der Module zum Bereitstellung der Daten gemessen werden.

Die Module zur Bereitstellung der Daten sind hingegen performancekritisch. Dies trifft insbesondere auf das Backend und den interaktiven Client auf Kundenseite zu, da hiervon die Kundenzufriedenheit stark beeinflusst werden kann. Die Clientanwendung ist jedoch nicht in Java geschrieben, und damit außerhalb des Rahmens von *PermanEnt*. Damit ergibt sich das Backend als zentrales im Projekt zu untersuchendes Teilprojekt. Dieses ist modular und nutzt Abhängigkeiten (Bibliotheken), von denen einige im Haus entwickelt werden. Diese wurden ebenfalls untersucht.

Testarten Die Grundlage für die Erstellung der PUTs bilden funktionale Unittests. In GeoMap existieren drei Arten von Tests: (1) Klassische Unittests: Diese nutzen ausschließlich die lokale Codebasis, alle Operationen finden also innerhalb der Java Virtual Machine (JVM) statt. (2) Unittests mit Ad-Hoc Datenbanknutzung: Diese Tests kreieren während ihrer Laufzeit eine Datenbank in einer eigenen Dockerumgebung, befüllen diese mit Testdaten und führen ihren Test auf dieser Grundlage durch. (3) Integrationstests: Diese Tests nutzen den aktuellen Stand der vollständigen Datenbank und dauern typischerweise relativ lange. Die klassischen Unittests (1) sind für *PermanEnt* sehr gut nutzbar, diese entsprechen der Zielstellung bei der vorausgegangenen Entwicklung von Peass. Sie stellen nur einen kleinen Teil der Tests dar und bieten eine geringe Codeabdeckung. Die Unittests mit Ad-hoc Datenbanknutzung (2) bieten eine gute Codeabdeckung und konnten mit geeigneten Erweiterungen an Peass-CI und Peass im Rahmen von *PermanEnt* messbar gemacht werden. Die Integrationstests (3) sind aus zwei Gründen für *PermanEnt* ungeeignet: Ihre Laufzeit ist zu lang, Mehrfachausführen für statistisch nutzbare Messungen sind damit unpraktisch. Darüber hinaus ändert sich die Datenbasis kontinuierlich durch neu hinzugefügte Daten, damit kann sich ihre Ausführungsgeschwindigkeit ebenfalls oft ändern, ohne das eventuelle Codeänderungen dafür verantwortlich wären.

Multibranch-Pipeline Die Entwicklung von GeoMap wird mit git durchgeführt. Dabei findet die konkrete Entwicklung eines neuen Features auf einem eigenen Branch statt. Ist das Feature vollständig umgesetzt, wird es auf den `master`-Branch zusammengeführt, dies dauert häufig einige Tage. Peass-CI soll den Entwicklern möglichst kurzfristig Rückmeldung zu ihren Änderungen geben, deshalb sollten in *PermanEnt* auch die neu erstellten Branches gemessen werden können.

Hierfür wurde ein Ansatz entwickelt, um die Analyse mehrerer Branches durch Peass-CI zu ermöglichen. Dies erfordert die Definition eines Pipelineskripts im Repository sowie die Anpassung von Peass-CI, um die automatisch generierten Jobs für jeden Branch korrekt zu erkennen. Das Pipelineskript definieren einerseits, mit welchen Einstellungen Peass-CI die Messungen durchführt und dienen andererseits der Erkennung neuer Branches. Zusätzlich wurden statisch eingestellte Messungen des `master`-Branches und der abhängigen eigenen Bibliotheken definiert.

Geofox

Geofox basiert auf Gradle und Spring, und enthält Unittests sowie Integrationstests. Daher musste der Buildprozess für diese Technologien angepasst werden. Aufgrund der Vielzahl der Tests wurden vorerst nur die Unittests vollständig integriert. Um den Messaufwand zu reduzieren, wird nur Performance der master-Branches der Teilprojekte gemessen.

MobiVisor

Android In Android gibt es zwei Arten von Tests: Local-Unit-Tests und Instrumented-Tests. Für Local-Unit-Tests reicht Gradle allein aus, während für Instrumented-Tests (emulierte Tests) zusätzlich zu Gradle ein Emulator oder ein Android-Gerät benötigt wird. Im MobiVisor-Projekt werden nur emulierte Tests verwendet, aber es ist möglich, Unittests von den emulierten Tests zu trennen.

Um regelmäßige Tests mit Peass und MobiVisor durchzuführen, wurde ein separater Docker-Container erstellt, auf dem der Jenkins-Server läuft. Jenkins ist so konfiguriert, dass die Tests täglich laufen. Zur Steuerung der Commitselektion haben wir einen *PermanEnt*-Branch im MobiVisor Projekt angelegt. Vor dem Test wird der Master-Branch in den *PermanEnt*-Branch zusammengeführt. Alle Tests werden im Emulator ausgeführt, anstatt zwei Arten von Tests zu haben.

Emulator Die Testabdeckung von Instrumented-Tests ist wesentlich höher, deshalb wurde im Lauf des Projekts die Anforderung hinzugefügt, instumentierte Tests zu unterstützen. Anfangs wurde Anbox als Emulator genutzt und aktuell wird ein Android-Virtual-Device von einem Laptop über SSH mit Jenkins genutzt, da dieser unsere Anforderungen besser erfüllt und eine Hardware-Beschleunigung unterstützt.

AP 2 – Performance-Unittests-CI-Integration

Das Peass-CI-Plugin[5] zur Messung von Performanceunterschieden wurde im Projekt entwickelt und in die Messprozesse der Partner integriert. Dabei konnte das Plugin auch in die Jenkins-Infrastruktur integriert werden, und ist damit für jeden Benutzer von Jenkins direkt installierbar.

Um die ursprüngliche Peass-Methode im CI-Plugin nutzen zu können, musste sie konzeptionell angepasst und technisch erweitert werden. Dabei wurden folgende **Prozessanpassungen** vorgenommen: (1) Anpassung des Regressionstestselektionsprozess zur CI-Unterstützung, (2) Übergabe von Credentials durch Peass-CI an den Buildprozess, (3) Unterstützung der Verarbeitung großer entstehender Datenmengen bei der Messung und (4) Unterstützung von Nightly Builds . Darüber hinaus wurden Anpassungen an der Unterstützung von **Testtypen** umgesetzt: (1) Unterstützung von Docker-Unittests, (2) Unterstützung von Tests mit Mocking und (3) Unterstützung parametrisierter Tests. Zur **Beschleunigung der Messung** wurden außerdem die (1) Konfiguration der Messung für die Anwendungsfälle und die (2) Isolierung von Messungen untersucht. Darüber hinaus war die **Unterstützung neuer Technologien**, insbesondere die Unterstützung von (1) Gradle, (2) Spring und (3) emulierten Android-Tests notwendig. Die einzelnen Erweiterungen werden im Folgenden dargestellt.

Prozessanpassungen

Regressionstestselektionsprozess Das Tool Peass [5] wurde für die Analyse aller Commits eines Projekts entworfen. Um es im Kontext der kontinuierlichen Performanzmessung nutzbar zu machen, musste die Architektur angepasst werden. Abbildung 5 zeigt den geänderten Prozess: Statt, wie vorher, jeden Test einzeln zu analysieren, wird die erste Version statisch und dynamisch beim ersten Aufruf analysiert. Die Ergebnisdaten werden dann in einen persistenten JSON-Speicher im Jenkins-Workspace abgelegt (standardmäßig `jobs/$JOBNAME/peass-data` im `$JENKINS_HOME`-Verzeichnis).

Erfolgt ein `git push`, so wird die statische und dynamische Quelltextanalyse für den neuen Commit durchgeführt. Aus den Daten des neuen Commits und den bestehenden Daten wird dann die Liste der Test-Commit-Paare für die Messung abgeleitet. Äquivalent zu diesem Prozess werden die Zwischenergebnisse der Messungen und der Ursachenanalyse ebenfalls im persistenten JSON-Speicher im Jenkins-Workspace abgelegt.

Credentials und Peass-CI Credentials werden für Buildprozesse benötigt, bspw. um Abhängigkeiten nachzuladen. Damit die Credentials nicht im Klartext gespeichert werden müssen, erlaubt es Jenkins, die Credentials sicher zu speichern und nur während der Laufzeit der Jobs zu entschlüsseln. Da Peass-CI diese Credentials auch benötigt, müssen diese geeignet an das Plugin übergeben werden. Dabei muss verhindert werden, dass diese Credentials in den Logs als Klartext auftauchen, um die Sicherheit nicht zu gefährden.

Hierfür wurden die Credentials mit ihren Ids an das Jenkins-Credentials-Plugin übergeben. Diese werden dann durch Peass an die Kindprozesse weitergegeben, so dass sie in den jeweiligen Prozessen nutzbar sind.

[5] `https://github.com/jenkinsci/peass-ci-plugin`

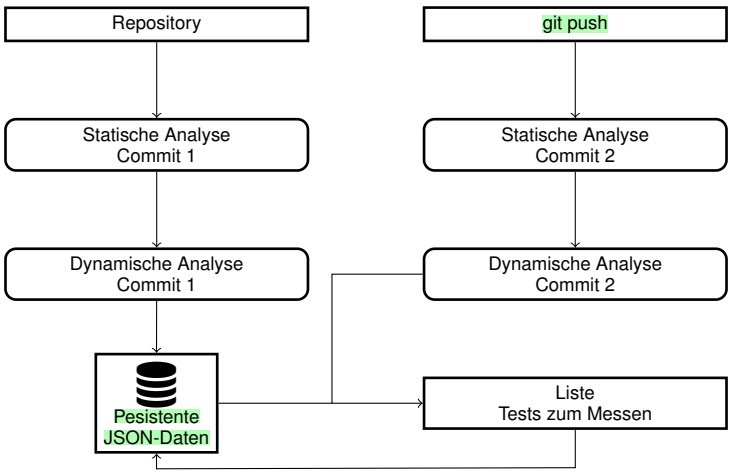

Abbildung 5: Geänderter Regressionstestselektionsprozess

Anschließend erfolgt die Maskierung der Logs mit dem Credentials-Plugin, so dass Credentials nicht für andere Jenkins-Benutzer sichtbar werden.

Große Datenmengen Geofox beinhaltet rechenintensive Routing-Algorithmen sowie Datentransformationen, zu deren Laufzeit bestimmte Codeabschnitte sehr oft (tausende oder hunderttausende Mal) durchlaufen werden. Dies führte zu nicht mehr handhabbaren Dateigrößen bei den Ausführungs-Traces für die Ursachenanalyse, welche von Peass-CI im Dateisystem gespeichert werden. Um damit umzugehen, wurde der Parameter `excludeForTracing` eingeführt, um bestimmte Methoden vom Tracing auszuschließen. Darüber hinaus wurde für die Regressionstestselektion der Parameter `onlyOneCallRecording` eingeführt, um den Aufruf einer vielfach aufgerufenen Methode nur einmal zu erfassen. Da sich die Tracedaten effektiv komprimieren lassen, werden diese nun standardmäßig komprimiert und nicht erst nachträglich, was weitere Einsparungen zu Folge hat. Um zu ermöglichen, dass die größeren Datenmengen in der Mess-JVM verwaltet werden, wurde darüber hinaus der Parameter `xmx` hinzugefügt, der es ermöglicht, die Heap-Größe zu erhöhen.

Jenkins wurde durch den Import großer Anwendungslogfiles sehr langsam. Daher wurde zusätzlich implementiert, dass Jenkins ab einer definierten Größe (`importLogSizeInMb`) Anwendungslogfiles nicht direkt lädt.

Nightly Builds Performancemessungen benötigen viele Wiederholungen, um statistisch aussagekräftig Unterschiede bestimmen zu können. Sind die einzelnen Testausführungen dann noch relativ zeitintensiv, ist es nicht mehr möglich während des gleichen Arbeitstags die Ergebnisse zu ermitteln. Laufen dann mehrere kleine Iterationen auf, kann es dazu kommen, dass die Messungen der Einzeliterationen mehrere Tage dauert. Um stattdessen ein relativ zeitnahes und relevantes Ergebnis zu liefern, wurden nächtliche Messungen implementiert.

Nächtliche Messungen werden täglich nach dem Ende der Arbeitszeit gestartet. Sie messen alle an diesem Tag gelieferten Iterationen mit einem Mal. Dafür wird der Unterschied nicht zum jeweils letzten git-commit bestimmt, sondern zum letzten gemessenen. Mit der Realisierung dieses Features ist es auch möglich, alternative Messintervalle festzulegen. In GeoMap wurde eine wöchentliche Messung mit höherer Empfindlichkeit konfiguriert. Diese nutzt die arbeitsfreie Zeit am Wochenende, und kann dabei kleinere Performanceunterschiede über mehrere Iterationen finden. Damit können auch solche Performanceunterschiede gefunden werden, die verteilt über mehrere Tage schrittweise hinzugekommen sind.

Testtypen

Unterstützung von Docker-Unittest Der größte Teil der in GeoMap gemessenen Tests sind Unittests mit Ad-Hoc Datenbanknutzung, wobei die Ad-hoc Datenbank als Docker-Container gestartet wird. Das Aufbauen des Containers und der Datenbank, das Befüllen mit Testdaten und das nachträgliche Beenden des Containers kosten typischerweise deutlich mehr Zeit und haben eine deutlich höhere Varianz als der auszuführende Test selbst. Wird also die Gesamtzeit des Tests gemessen, werden mögliche Performanceänderungen der eigentlichen Testmethode statistisch deutlich schwerer erkennbar. In *PermanEnt* wurde deshalb Peass (mit

entsprechender Erweiterung der Konfigurationsmöglichkeiten und Darstellung in Peass-CI) weiterentwickelt, so dass es nun möglich ist, ausschließlich die konkrete Testmethode zu messen. Dies wird durch das Flag `onlyMeasureWorkload` aktiviert.

Unterstützung von Tests mit Mocking Bei Geofox wird in den Test das Mockito-Framework zum Mocken von Abhängigkeiten eingesetzt. Zur Unterstützung der Messbarkeit von Testfällen, die nur einmal ausgeführt werden sollen, die `@InjectMocks` oder die statisches Mocking benutzen, waren Änderungen am von Peass benutzen Messframework KoPeMe notwendig.

Mocks und Spies sind darauf ausgelegt, einmal benutzt zu werden. Dabei wird beispielweise geprüft, ob eine Methode n mal aufgerufen wird. Wird der Testfall zur Messung der Performanz 5 mal wiederholt, wird die Methode demzufolge $5 \cdot n$ mal aufgerufen, und die Mockito-Assertion schlägt fehl. Um derartige Testfälle dennoch messen zu können, wurde der Parameter `clearAllCaches` eingeführt. Ist dieser gesetzt, wird bei jeder Testausführung `Mockito.clearAllCaches()` aufgerufen, so dass bspw. die Zählung der Methodenaufrufe erneut bei 0 beginnt.

Durch die Nutzung von `@InjectMocks` ist es möglich, Mock-Objekte zu kombinieren. Hierfür ist es notwendig, eine manuelle Reinitialisierung des `JupiterEngineExecutionContext` durchzuführen. Dadurch erfolgt eine neue Initialisierung der `MockitoExtension` und neue Mocks werden bereitgestellt. Dies wurde implementiert[6] und geschieht für den Nutzer transparent.

Statisches Mocking wird mit `Mockito.mockStatic` erzeugt und ist nur im aktuellen Thread nutzbar. KoPeMe startet den Messprozess in einem Messthread, um diesen Thread ggf. von außen bei zu hoher Ausführungsdauer beenden zu können. Werden in `@BeforeAll` statische Mocks erzeugt, muss das im selben Thread erfolgen. Die Implementierung wurde in KoPeMe dahingehend angepasst, dass alle Initialisierungen im Messthread erfolgen.

Parametrisierte Tests JUnit erlaubt die Definition von Parameterlisten, um einzelne Tests wiederholt mit verschiedenen Parametern auszuführen. Dies wird über Annotationen im Java-Code gesteuert und führt dazu, dass Testmethoden (bspw. `TestA#method1`) mehrere Ausführung mit jeweils unterschiedlichem Index der konkreten Parameterwerte in der Parameterwerteliste haben (bspw. `TestA#method1[0]` und `TestA#method1[1]`). Die Buildtools maven und Gradle ermöglichen aktuell keine direkte Ausführung eines Testfalls mit Parameterindex. Daher wurde implementiert, dass einmalig der gesamte Testfall aufgerufen wird und anschließend der Testfall mit den gewählten Parametern durch KoPeMe ausgeführt wird. Die gewählten Parameter werden in der Peass-CI Oberfläche dargestellt.

Beschleunigung der Messung

Konfigurationen der Messung für Anwendungsfälle Zur Konfiguration der Messung für die Anwendungsfälle wurde der bereits vorab in Peass definierte Prozess zur Messkonfiguration [10] erweitert. Der ursprüngliche Prozess umfasste die (1) die Definition künstlicher Test, (2) die Ausführung verschiedener Mess- und Analyseverfahren und (3) den Vergleich der korrekten Erkennung von Performanzunterschieden. Zur weiteren Beschleunigung wurden erweiterte Messkonfigurationen (bspw. die Parallelisierung der Messung) genutzt und die Konfigurationen anschließend auf die Messprozesse der Partner angewandt.

Zur besseren Verständlichkeit wurden Heatmaps generiert, die für Endanwender verständlich zeigen, welche Konfiguration Performanceänderungen wie gut identifizieren kann. Ein Beispielgraph hierfür ist Abbildung 6: Mit erhöhter VM-Anzahl und mit erhöhter Iterationsanzahl steigt das F_1-Maß typischerweise, wobei die Iterationsanzahl, bei der die JIT-Compilation stattfindet, zu einem niedrigeren F_1-Maß führt. Der Prozess zur Messkonfiguration wurde in [11] publiziert.

Untersuchung der Isolierung von Messungen (bspw. durch Container) Aufgrund der hohen Wiederholungsanzahlen, die für zuverlässige Performancemessungen nötig sind, erfordern Performancemessungen relativ viel Zeit (in der Regel im Bereich von Stunden). Die Ausführung von zwei Messungen auf demselben Server kann dabei dazu führen, dass Messungen verfälscht und damit Performanceunterschiede nicht mehr bestimmbar werden.

[6] https://github.com/DaGeRe/KoPeMe/blob/f466292cf9bad934e1795a7fd49e9c54e7bc3af3/kopeme-junit5/src/main/java/de/dagere/kopeme/junit5/extension/KoPeMeJUnit5Starter.java#L82

Im Rahmen des Projektes wurde daher untersucht, inwiefern die Parallelisierung von Messungen zur Verschlechterung der Messbarkeit führt. Abbildung 6 zeigt hierfür beispielhaft die sequenzielle Messung, die parallelisierte Messung und die parallelisierte Messung mit der Störung der Messung durch zufällige Workloads. Dabei zeigt sich, dass das F_1-Maß durch Parallelisierung verbessert wird und auch die Störung der Messungen nicht zu einer Reduktion der Messgenauigkeit führen. Diese Ergebnisse bestätigen die Messergebnisse in Cloud-Umgebungen [12].

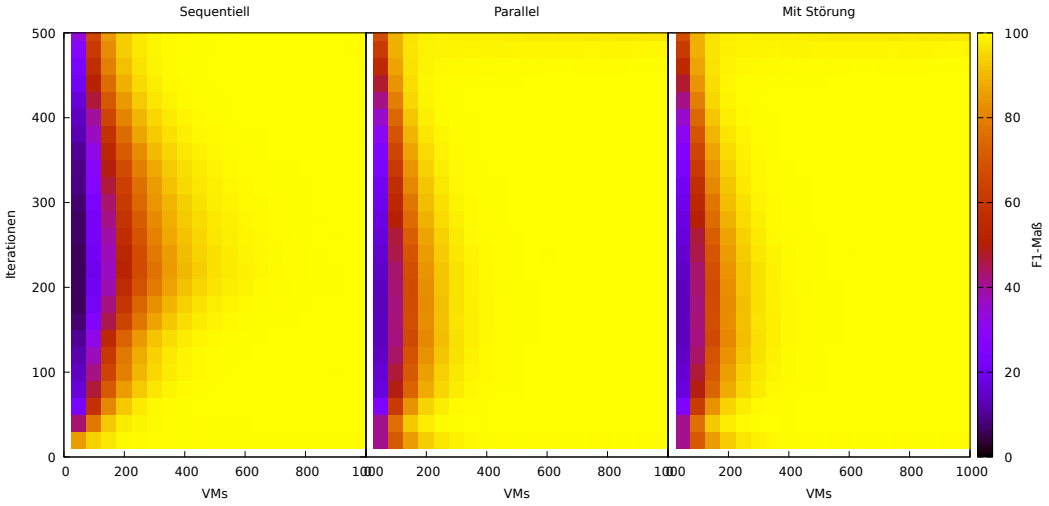

Abbildung 6: F_1-Maß bei der Parallelisierung von Performancemessungen mit und ohne Störung der Messung durch zufällige Workloads

Unterstützung neuer Technologien

Gradle Bei Geofox wird Gradle als Buildsystem eingesetzt. Gradle erlaubt hier vielfältige Anpassungen in Form von Groovy-Skripten. Diese müssen von Peass-CI korrekt ausgeführt werden, damit die Projekte erfolgreich gebaut werden können.

Für die Unterstützung der Messung der Unittestperformance in Gradle-Projekten mussten umfangreiche Anpassungen vorgenommen werden. Die wichtigsten waren dabei:

- Das Gesamtprojekt besteht aus Modulen und untergeordneten Submodulen. Als interne Konvention trägt die Bezeichnung der Submodule im Verzeichnisbaum immer das Präfix “module-”, statt wir bei Gradle üblich nur den Namen. Dies wird in den Gradle-Konfigurationsdateien durch Skripting-Elemente umgesetzt.

  ```
  rootProject.children.each { subproject ->
    subproject.projectDir = file("module-" + subproject.name)
    subproject.buildFileName = "${subproject.name}.gradle"
  }
  ```

  Ähnliche Änderungen waren für die Anpassung der JVM-Argumente oder der System-Properties nötig.

- Peass ist nur auf Module anwendbar, die Java-Quelltexte bauen. In Geofox sind allerdings auch Module vorhanden, die kein Java bauen. Daher wurde durch das Parsing der Ausgabe von `./gradlew tasks` ermittelt, welche Module welche Plugins verwenden. Vorher untersuchte Implementierungsalternativen, die direkt durch Parsing der `build.gradle` ermitteln, ob das Java-Plugin genutzt wird, stellten sich als zu instabil heraus.
- Zur Unterstützung der Messung von Integrationstests mussten entsprechende Konfigurationsparameter, u. a. für die automatisierte Konfiguration des Integrationstestplugins, die Spezifikation des Tasknamen des Integrationstests und den Pfad der Integrationstests in den einzelnen Modulen, hinzugefügt werden.
- Gradle führt Unittests bei Geofox standardmäßig parallel aus. Der Ausführungsmodus in der `build.gradle` muss daher automatisiert durch Parsing der Datei auf `SAME_THREAD` gesetzt werden, damit die Tests sequenziell ausgeführt werden.

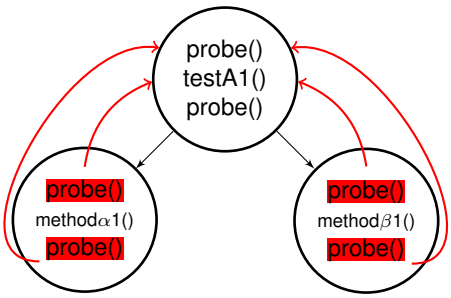

Abbildung 7: Einfluss des Overheads in Kindknoten

- Jeder Start eines Gradle-Builds führt zur Erstellung einer Daemon-Logdatei. Diese mussten automatisiert gelöscht werden, um Ressourcen auf dem Buildserver zu sparen.

Spring-Kompatibilitat Wenn Spring-Tests ausgeführt werden, muss die mit der jeweiligen Spring-Version kompatible JUnit-Version verwendet werden. Daher muss ermittelt werden, ob Spring im Projekt verwendet wird und welche Spring-Version verwendet wird. Ob Spring im Projekt verwendet wird, wird dabei durch Analyse der Ausgabe des `tasks` Gradle-Kommandos ermittelt. Wenn Spring verwendet wird, wird dessen Version dann durch Parsing der Builddateien ermittelt.

Weiterhin führen Spring-Tests vor der Durchführung Startup-Code aus. Werden diese wiederholt ausgeführt, wird die Ausführung des Startup-Codes jedesmal mit gemessen, was unerwünscht ist. Daher wurde eine Option eingeführt, um diese Tests direkt innerhalb der Testmethode zu messen anstatt außerhalb durch KoPeMe.

Anpassung für Emulator Um die Messung mit Android-Emulator zu ermöglichen, müssen der Testklassenpfad, der Klassenpfad sowie die Android-Versionen konfiguriert werden. Für die Ausführung mit Emulator muss general der Parameter `useAnbox` gesetzt sein.

Zur Definition der Klasenpfade wurden die Parameter `clazzFolders` und `testClazzFolders` eingeführt. Zur Definition des Testziels, das für Android-Projekte in der Regel `testRelease` oder `testDebug` ist, wurde der Parameter `testGoal` eingeführt.

Mit dem Parameter `androidGradleTasks` können spezielle Gradle-Tasks vor der eigentlichen Testausführung angestoßen werden. Um sicherzustellen, dass Tests erfolgreich ausgeführt werden, können die Parameter `androidCompileSdkVersion`, `androidMinSdkVersion`, `androidTargetSdkVersion` und `androidGradleVersion` überschrieben werden. Um das Package für die Testausführung zu Spezifizieren kann dieses durch den Parameter `androidTestPackageName` gesetzt werden.

Ein Problem bei der Verwendung von Peass mit einem Emulator besteht darin, dass Peass keine Berechtigung hat, auf Dateien zuzugreifen. Berechtigungen werden in Android-Projekten in der Datei `AndroidManifest.xml` angegeben. Mit dem Parameter `androidManifest` wird der Pfad angegeben, damit Peass automatisch die notwendigen Berechtigungen hinzufügt, um Ergebnisse in den Emulator schreiben zu können.

AP 3 – Ursachenanalyse

Zur Ausführung der Ursachenanalyse mit angemessenem Aufwand wurde die Möglichkeit der Teilbaumanalysen sowie weitere Möglichkeiten zur Overheadreduktion evaluiert. Darüber hinaus war ursprünglich die Implementierung von adaptivem Monitoring und automatisiertem Export von Profilerkonfigurationen geplant. Nach Feststellung der Ursache wurde eine Möglichkeit zur Klassifizierung von Performanceunterschieden definiert.

Die Ursachenanalyse wurde darüber hinaus für die Nutzung in einem Android-Emulator angepasst.

Teilbaumanalysen Für die Bestimmung der Ursachen von Performanceunterschieden ist es notwendig, jeden Knoten des Aufrufbaums einzeln zu messen. Für die Messung werden in Kindknoten sog. Probes hinzufügt, die Overhead verursachen. Problematisch ist hierbei insbesondere, dass die Probes der Kindknoten die Performancemesswerte des Elterknoten beeinflussen (siehe Abbildung 7).

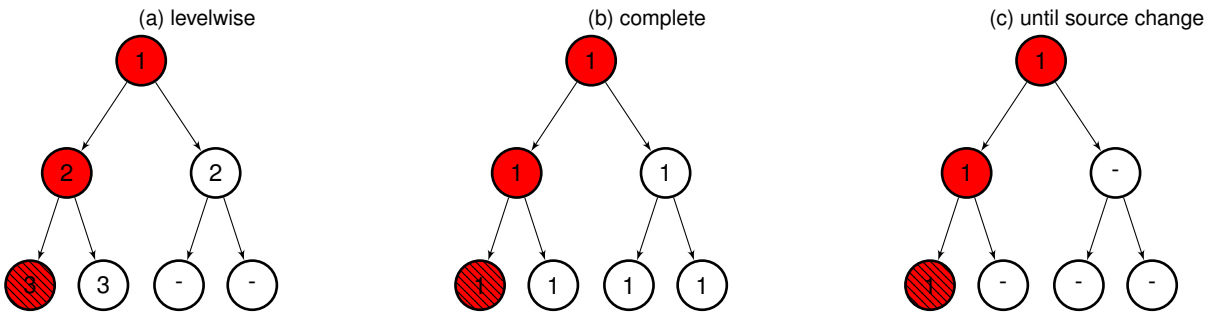

Abbildung 8: Stategien der Ursachenanalyse

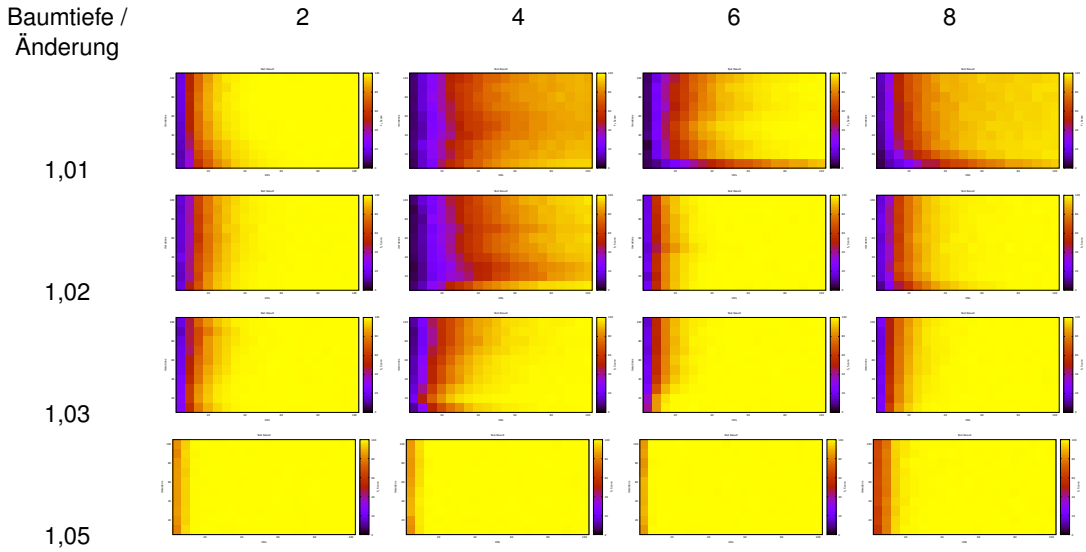

Tabelle 1: F_1-Maße der Messungen der UNTIL_SOURCE_CHANGE-Strategie

Daher wird eingangs die Struktur des gesamten Aufrufbaums ermittelt. Um den Overhead zu reduzieren, können die Ebenen des Aufrufbaums einzeln gemessen werden [13] (im Folgenden **levelwise**-Strategie), was den gemessenen Overhead reduziert. Alternativ kann der gesamte Aufrufbaum auf einmal [14] (im Folgenden **complete**-Strategie) gemessen werden. Basierend auf den Messwerten wird dann bestimmt, wo ein Performanceunterschied vorliegt.

Um den Overhead zu reduzieren, existiert in Peass eine weitere Möglichkeit: Da die Quelltexte und deren Unterschiede aus dem Versionskontrollsystem bekannt sind, kann der Pfad zu Performanceänderung direkt bestimmt werden. Anschließend werden nur die Knoten gemessen, die einen Pfad zur Performanceänderung aufweisen. Diese sog. **until source change**-Strategie wurde in Peass implementiert. Abbildung 8 zeigt beispielhaft, in welchem Schritt welcher Knoten gemessen wird.

Zum Vergleich der Strategien wurden künstliche Projekte erzeugt[7], anhand deren die Effizienz zum Auffinden von Performanceunterschieden mit verschiedener Stärke in verschiedenen Baumtiefen untersucht wurde. Tabelle 1 zeigt die Effizienz der Performanceunterschiedserkennung für die UNTIL_SOURCE_CHANGE-Strategie [15, S. 103-109].

Insgesamt hat sich die Ursachenanalyse als hilfreiches Werkzeug zum Verständnis von Performanceunterschieden erwiesen. Teilweise sind jedoch Ursachen von Performanzunterschieden nicht a priori klar, bspw. wenn mehrere Quelltextänderungen vorliegen, wenn Konfigurationsdateien gleichzeitig geändert worden oder wenn Ursachen in anderen Projekten liegen (wie bspw. Performanceänderungen im GeoMap-Backend, die durch Änderungen in den GeoMap-Bibliotheken verursacht worden). Daher können auch mit Nutzung der Ursachenanalyse faktorielle Experimente zum Verständnis von Performanzunterschieden nötig sein.

Overheadreduktion Für die Messung der Dauer einzelner Methodenaufrufe ist ein geringer Overhead essenziell. Daher wurden verschiedene Optionen untersucht, um den Overhead zu reduzieren [15, S. 94-102].

[7] Siehehttps://github.com/DaGeRe/precision-experiments-rca

Folgende Optionen haben sich dabei als zielführend erwiesen: (1) Nutzung von Quelltextinstrumentierung statt AspectJ zur Instrumentierung, (2) Erfassung reduzierter Methodenausfühungsdaten (`DurationRecord`) statt vollständiger Methodenausfühungsdaten (`OperationExecutionRecord`), (3) Verwendung einer effizienteren Queue (`SynchronizedCircularFifoQueue` statt `LinkedBlockingQueue`) sowie (4) Schreiben von aggregierten Messdaten statt einzelnen Methodenaufrufen [9]. Die Effizienz der Optimierungsoptionen wurde mit dem MooBench-Benchmark [16] überprüft. Abbildung 9 zeigt, dass der Monitoringoverhead mit größerer Aufrufbaumtiefe mit den Optimierungen erheblich langsamer steigt.

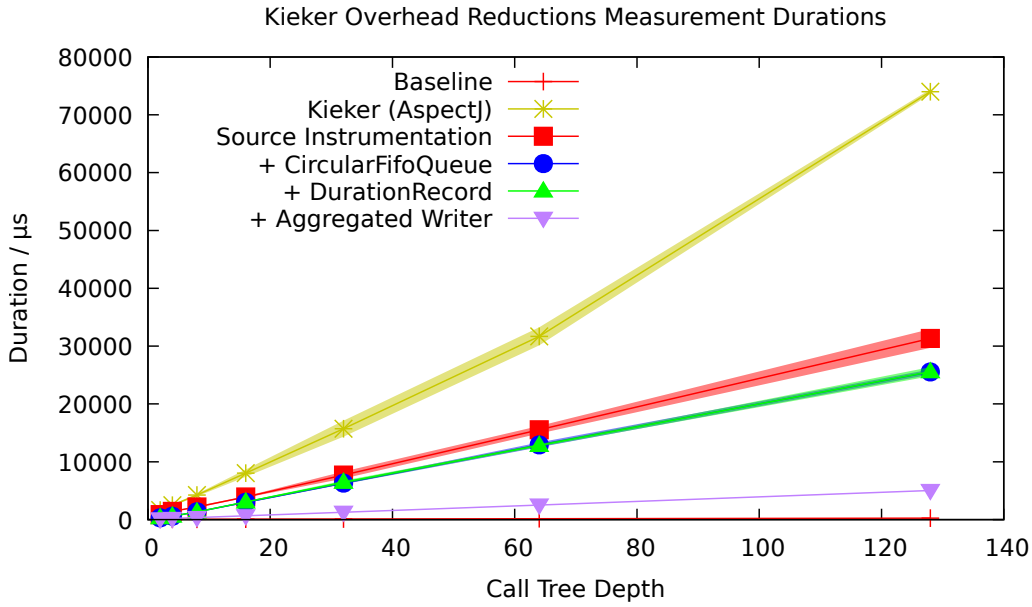

Abbildung 9: Wachstum des Overheads mit steigender Baumtiefe, aus [9]

Adaptives Monitoring Ursprünglich war es geplant, eine weitere Beschleunigung der Ursachenanalyse durch Aktivierung und Deaktivierung des Monitorings in einzelnen Knoten des Aufrufbaums zu erreichen. Durch die Overheadreduktion und die Definition der Strategien ist eine effiziente Ursachenanalyse möglich. Eine weitere Beschleunigung der Ursachenanalyse durch adaptives Monitoring wäre möglich, wurde aber aus Priorisierungsgründen im Projekt nicht untersucht.

Profiler-Konfiguration Die erkannten Performanceunterschiede konnten durch die Ursachenanalyse bzw. faktorielle Experimente eingegrenzt werden. Ein automatisierter Export von Profilerkonfigurationen war daher nicht notwendig.

Klassifizierung Eine manuelle Klassifizierung von Performanceunterschieden kann in der Peass-CI-Oberfläche vorgenommen werden. Hierbei kann bspw. spezifiziert werden, dass ein Performanceunterschied eine funktionale Ursache hat, durch eine Optimierung oder durch ein Refactoring enstanden ist. Die Menge der Klassen kann durch den Endnutzer erweitert werden.

Die aktuelle Nutzung dieser Klassifikationsmöglichkeit basiert auf den Intentionen der Entwickler, nicht auf den technischen Eigenschaften des Quelltextes (wie ursprünglich vorgesehen). Das Vorhaben, Performanceprobleme (ggf. automatisiert) zu klassifizieren setzte die Annahme voraus, dass eine größere Anzahl von Performanceunterschieden vorhanden ist, die sich ähnlichen Klassen zuordnen lassen. Da Entwickler nicht so feingranular committen und dadurch nicht so viele Performanceunterschiede entstehen wie erwartet, war diese Art der Klassifikation im Projekt nicht umsetzbar.

Die nun gewählte Umsetzung der Klassifizierung ermöglicht einen systematischen Umgang mit Performanceunterschieden: Wenn ein Performanceunterschied auftritt, kann dieser durch einen Entwickler begutachtet werden. Dieser kann anschließend manuell die Unterschiedsursache klassifizieren; anschließend kann während der weiteren Entwicklung und durch andere Entwickler diese Klassifizerung eingesehen werden.

Anpassung für Emulator Um die Ursachenanalyse mit dem Emulator lauffähig zu machen, müssen Umgebungsvariablen, Konfigurationsdateien und Ergebnisse zwischen dem Hostsystem, auf dem Peass läuft, und dem emulierten Android-System ausgetauscht werden.

Zur Übertragung der Konfigurationsdateien werden diese vor der Performancemessung mittels der Android-Debug-Bridge (ADB) in den Emulator kopieren. Peass übergibt Verzeichnispfade normalerweise über Umgebungsvariablen und Java-Properties, bspw. `KOPEME_HOME` und `-Dkieker.monitoring.configuration`. Die Pfade müssen auf die entsprechenden Pfade im Emulatur angepasst und die Dateien entsprechend gesendet werden. Nachdem die Perfomance-Tests durchgelaufen sind, benötigt Peass auf dem Host die generierten Ergebnisse aus dem Emulator. Die Dateien werden daher mit ADB vom Emulator auf den Host kopiert und für die weitere Verabeitung bereitgestellt.

In Android gibt es den sogenannten `StrictMode`. `StrictMode` wird verwendet, um unerwünschten Zugriff auf Festplatte und Netzwerk auf dem Hauptthread der Anwendung zu verhindern, um eine reaktionsschnelle Anwendung zu ermöglichen. Bei der Verwendung von Peass innerhalb des Emulators entsteht ein Netzwerkzugriff auf den Hauptthread während der Performancemessung. Um den Modus auszuschalten wurde der Parameter `strictMode` für Peass eingeführt.

AP 4 – Testgenerierung und -selektion

Während des Projektes wurde evaluiert, inwiefern eine Generierung zusätzlicher Tests und eine automatisierte Testgenerierung durch Anpassung bestehender Tests möglich ist. Darüber hinaus wurden drei erweiterte Methoden zur Testselektion, die abdeckungsbasierte Testselektion, die messbarkeitsbasierte Testselektion sowie die regelbasierte Testselektion, implementiert. Diese werden im Folgenden dargestellt.

Generierung Während der Fallstudien (siehe AP6) zeigte sich, dass aus den durch Tests abgedeckten Quelltextunterschieden eine Vielzahl von Test-Commit-Paaren erzeugbar ist, die in den untersuchten Projekten eine umfangreiche Performancemessung und anschließende Analyse nötig machen. Eine zusätzliche Generierung von Testfällen anhand von Monitoringdaten, Lasttests oder deren Kombination mit Unittests war daher nicht notwendig.

Automatisierte Testgenerierung Variablenwert Instanz- oder Klassenvariablen, die auf Klassenebene definiert werden, können dazu führen, dass die Mehrfachausführung eines Tests unmöglich wird. Beispielsweise kann ein in einer Instanzvariable definierter Timeout dazu führen, dass ein Unittest bei einmaliger Ausführung korrekt funktioniert und bei Mehrfachausführung fehlschlägt.

Um dieses Problem zu beheben, ist es notwendig, einen Test zu generieren, in dem die definierte Variable angepasst wird. Daher wurde der Parameter `increaseVariableValue` erstellt. Durch Setzen des Parameters wird ein angepasster Test definiert, der die Instanzvariable mit anderem Initialisierungswert benutzt. So wird es möglich, Tests auch dann auszuführen, wenn die Ausführung aufgrund unpassender Variableninitialisierungen sonst nicht möglich wäre.

Selektion durch abdeckungsbasierte Testselektion Die Messung aller Test-Commit-Paare, die Quelltextänderungen abdecken, verursacht einen enormen Ressourcenverbrauch. Daher wurde im Rahmen des Projektes die abdeckungsbasierte Testselektion (engl. *coverage-based test selection*) entworfen. Die bereits vorher selektierten Tests werden dabei mit zwei Zielen weiter eingegrenzt: (1) Jede Quelltextänderung soll mindestens einmal abgedeckt werden und (2) die Ausführung der Quelltextänderung soll einen möglichst hohen Anteil an der Messung haben. Um beide Ziele zu erreichen, werden alle Quelltextänderungen der Menge Q hinzugefügt. Anschließend wird für jeden Testfall ermittelt, wie oft er Quelltextänderungen aufruft (wobei Mehrfachaufrufe mehrfach gezählt werden). Der Testfall mit der *höchsten Ausführungsanzahl* wird anschließend selektiert und alle von ihm aufgerufenen Quelltextänderungen werden aus der Menge Q entfernt. Dieser Prozess wird wiederholt, bis Q leer ist, d.h. bis für alle abgedeckten Änderungen der Testfall mit der höchsten Ausführungsanzahl selektiert wurde.

Abbildung 10 verdeutlicht diesen Prozess: Liegen Quelltextänderungen in m_2, m_6 und m_9 vor, wird zuerst `Test 3` selektiert, da seine Ausführungsanzahl 6 ist. Dadurch werden m_6 und m_9 aus Q entfernt. Anschließend wird `Test 1` selektiert, da er m_2 abdeckt. `Test 2` wird nicht selektiert, da er keine nicht aufgerufene Methode abdeckt.

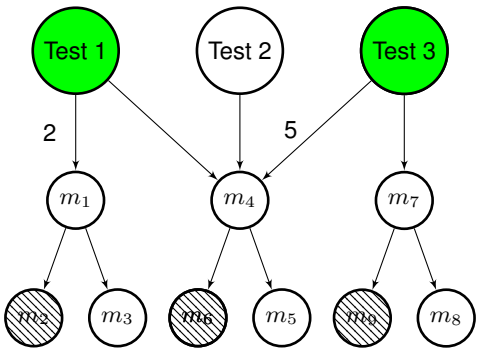

Abbildung 10: Beispiel für abdeckungsbasierte Quelltextselektion

Messbarkeitsbasierte Testselektion Um die Messbarkeit sicherzustellen, wurde implementiert, dass im Rahmen der Regressionstestselektion jeder selektierte Testfall mit zwei Iterationen in einer VM ausgeführt wird. Entstehen keine Messwerte, bspw. weil eine Exception die Mehrfachausführung verhindert, wird der Testfall nicht selektiert.

Selektion durch regelbasierte Testselektion Häufig gibt es Funktionale Tests mit verschieden großen Workloads, oft wird zwischen Unit-, Modul- und Integrationstests unterschieden. Da sie sich in Umfang und Setup teilweise deutlich unterscheiden, kann es nötig sein, sie in Peass mit verschiedenen Konfigurationen zu messen. Ein Werkzeug, das geeignet ist, die verschiedenen Setups für die verschiedenen Klassen von Tests wiederholbar zur Verfügung zu stellen, ist JUnit `TestRule`. Dieses wird beispielsweise in GeoMap genutzt, um die Ad-Hoc Datenbank für entsprechende Tests zur Verfügung zu stellen. Im Projekt wurde Peass bzw. Peass-CI so erweitert, dass bei entsprechender Konfiguration ausgewählte `TestRules` als Whitelist oder Blacklist genutzt werden können. Damit können die reinen Unittests eine andere Messkonfiguration nutzen als die Tests mit Dockerdatenbank.

AP 5 – Ergebnisdiskussion

Um ein kontinuierliches **Communitymanagement** aufzubauen, wurde durch die evermind eine Projektwebseite gestaltet und gepflegt. Die Tools wurden quelloffen auf GitHub zur Verfügung gestellt. Dort erfolgte die Verwaltung der Bugs, Featurerequests und Patches, die im Projektkontext und darüber hinaus entstanden sind. Die GitHub-Repositories ermöglichen auch eine Weiterentwicklung der Tools nach Projektende.

Zum **Austausch mit Experten** wurden die Projektergebnisse auf Anwenderworkshops und wissenschaftlichen Konferenzen vorgestellt. Die Anwenderworkshops umfassten dabei folgende Veranstaltungen:

- it-tage 2021: Performanceprobleme durch CI-Messungen finden (`https://www.ittage.informatik-aktuell.de/programm/2021/performanceprobleme-durch-ci-messungen-finden.html`)
- FOSDEM 2021: Identifying Performance Changes Using Peass (`https://archive.fosdem.org/2021/schedule/event/identifying_performance_changes_using_peass/`)
- Chemnitzer Linuxtage 2022: Überwachung der Performance in der CI mit OpenSource-Tools (`https://chemnitzer.linux-tage.de/2022/de/programm/beitrag/197`)
- QS-Tag 2022: Performanceunterschiede durch kontinuierliche Messungprozesse finden (`https://www.qs-tag.de/abstracts/performanceunterschiede-durch-kontinuierliche-messungprozesse`)

Die wissenschaftlichen Konferenzen umfassten dabei folgende Veranstaltungen:

- Symposium on Software Performance 2020
- GI-Fachgruppentreffen Test, Analyse und Verifikation von Software 2021
- Symposium on Software Performance 2021
- Symposium on Software Performance 2022
- International Conference on Software Quality, Reliability, and Security 2022

Darüber hinaus wurde der Ansatz von *PermanEnt* in einem Artikel in der Informatik Aktuell vorgestellt (`https://www.informatik-aktuell.de/entwicklung/methoden/performance-regressionen-fruehzeitig-erkennen-und-verm.html`).

Daneben fand wissenschaftlicher Austausch unter anderem mit dem FZ Jülich und der Kieker Community an den Universitäten in Kiel, Hamburg und Stuttgart statt.

AP 6 – Evaluation

Ursprünglich war geplant, den Einsatz von Peass-CI auf die untersuchten Projekte anhand von Befragungen der Entwickler und einer Kostennutzenanalyse zu evaluieren. In den untersuchten Projekten sind jedoch weniger Commits, in denen durch Unittests messbare Performanzänderungen stattgefunden haben, als erwartet aufgetreten. Daher werden im Folgenden die gefundenen Performanzänderungen sowie deren Relation zu den Gesamtcommits dargestellt.

GeoMap-Fallstudie

Setup Zur systematischen Untersuchung der in GeoMap auftretenden Performanceänderungen wurden alle vom 1.1.2022-31.6.2022 auftretenden Commits systematisch auf vorkommende Performanceänderungen untersucht. Dabei wurden zuerst alle 203 Commits mit Peass-CI gemessen. Um zu überprüfen, ob die durch Peass-CI gefundenen Performanceänderungen vollständig und korrekt sind, wurde darüber hinaus der ausgewählte Commitbereich durch Lasttests sowie manuell auf potenziell auftretende Performanceänderungen überprüft. Es konnte keine Quelltextänderung festgestellt werden, bei der das Team davon ausgeht, dass die Quelltextänderung eine Performanceänderung verursacht und die von Peass-CI und den Lasttests übersehen wurde.

Im Folgenden werden die Ergebnisse der Lasttestmessung sowie von Peass-CI dargestellt.

Lasttests Im Rahmen des Projektes wurden in Abstimmung mit dem GeoMap-Team Lasttests erstellt, die die zentralen Anwendungsfälle von GeoMap testen: Die Objektverwaltung und die Benutzerverwaltung. Hierfür wurden Skripte erstellt, die jeweils die Datenbank und den jeweiligen Service starten, sowie Lasttests, die die Nutzung der REST-Schnittstellen automatisieren.

Während der Großteil der vorhandenen Schnittstellen erwartungsgemäß funktioniert, trat in einer Schnittstelle ein unerwartetes Wachstum der Antwortzeiten bei der Wiederholung der Anfragen auf. Dies wurde mit Einsatz der im Projekt entwickelten Ursachenanalyse untersucht, wobei Teilschritte manuell durchgeführt werden mussten, da die Lasttestspezifikation nicht wie ein Unittest in Peass-CI eingesetzt werden konnte. Dabei stellte sich heraus, dass eine `Set` mit `Pattern` regelmäßig um neue, gleiche Instanzen ergänzt wurde. Durch die Behebung dieses Problems konnte die Ausführungszeit beschleunigt werden. Diese Beschleunigung war auch durch Peass-CI messbar.

Messergebnisse von Peass-CI Von den 203 analysierten Commits hatten 132 eine Quelltextänderung, bei den verbleibenden Commits wurden entweder nur Kommentare oder nur Konfigurationsdateien geändert. Von den Commits mit Quelltextänderungen enthielten 52 Commits Tests, die diese abdeckten. In 45 Testfällen verteilt über 19 Commits traten Performanceänderungen auf. Diese ließen sich den Klassen funktionale Änderung (24 Tests in 10 Commits), Teständerung (6 Tests in 5 Commits), Optimierung (9 Tests in 3 Commits) sowie Versionsupdates (6 Tests in einem Commit) zuordnen. Von den fünf Änderungen mit der größten Effektstärke in Peass-CI sind vier Änderungen ebenfalls in den Lasttests messbar. Die verbleibende Änderung wird von den Lasttests nicht abgedeckt. Dies zeigt, dass Unittests Performanceänderungen effektiv identifizieren können.

Die Ergebnisse der GeoMap-Fallstudie wurden in [17] publiziert.

geofox-Fallstudie

Setup Um in Geofox auftretende Performanceänderungen zu finden, wurden zwei Schritte ausgeführt: (1) Es wurde ein CI-Server aufgesetzt, der täglich die Commits des master-Banches kontinuierlich prüft. Dies wurde für die Hauptprojekte g3server, tariff, gti und fwr und 16 weitere Projekte erfolgreich eingerichtet. Aufgrund der großen Anzahl von Modulen und Commits wurde davon ausgegangen, dass entsprechende Änderungen

auftreten und gefunden werden können. Allerdings wurde ein großer Teil der Projektlaufzeit zur Bearbeitung von Fehlern und Problemen benötigt, so dass die Messungen im CI-Server erst ab ca. Oktober 2022 saubere Ergebnisse produzierten. (2) Aufgrund der unvollständigen Messergebnisse wurde der Zeitraum 1.1.2022 – 31.6.2022 separat durch Analyse aller Commits für die Hauptprojekte g3server, tariff, gti und fwr analysiert.

Messergebnisse Bei der kontinuierlichen Prüfung wurden zwischen November 2022 und Februar 2023 16 Commits (von insgesamt 562 Commits) ermittelt, die Performanceänderungen aufweisen. Durch fehlende Rechenressourcen und Implementierungsfehler konnten nicht alle Commits analysiert werden; insgesamt wurden 134 erfolgreiche Runs ausgeführt. Alle der Änderungen waren Teständerungen, funktionale Änderungen oder geringfügige Änderungen (bspw. Hinzufügen oder Entfernen eines Loggers), so dass sich keine Handlungsimplikation daraus ergab.

Der Zeitraum 1.1.2022 – 31.6.2022 umfasste insgesamt 1027 Commits. Von diesen Commits enthielten 628 Java-Quelltextänderungen, d. h. die anderen Commits enthielten Änderungen an Konfigurationsdateien oder in der Dokumentation, und konnten daher nicht von Peass analysiert werden. Mit den zur Verfügung stehenden Rechenressourcen konnten 490 Commits (von den 1027 Commits) durch die Regressionstestselektion analysiert werden. Von diesen hatten 187 Quelltextänderungen. 76 davon waren durch Unittests abgedeckt, d. h. ähnlich wie bei GeoMap war etwa jede dritte Quelltextänderung von einem Unittest abgedeckt.

Von den 76 Quelltextänderungen konnten mit den zur Verfügung stehenden Rechenressourcen 27 gemessen werden. Hierbei wurden 6 Commits mit Performanceänderungen identifiziert. Eine dieser Änderungen wurde durch eine Teständerung verursacht und eine Änderung wurde durch eine notwendige Anpassung einer Sicherheitsfunktionalität verursacht. Die anderen vier Änderungen waren notwendige funktionale Änderungen.

MobiVisor-Fallstudie

Da die Implementierung der Anbox-Emulation nicht rechtzeitig abgeschlossen werden konnte, wurde keine Fallstudie in MobiVisor durchgeführt. Es ist geplant, den Einsatz von Peass-CI an MobiVisor nach Projektende zu evaluieren.

4 Notwendigkeit und Angemessenheit der geleisteten Projektarbeiten

Die Messung der Performance durch transformierte bzw. generierte Unittests stellt ein wirtschaftliches und wissenschaftliches Alleinstellungsmerkmal im Kontext des Software Performance Engineerings dar. Gleichzeitig sind die notwendigen Projektarbeiten für kleine und mittelständige Unternehmen eine Forschungsinvestition mit nicht unerheblichem Risiko. Durch die im *PermanEnt*-Projekt erfolgten Forschungsarbeiten sind die wissenschaftlichen und technischen Grundlagen für die eine wirtschaftliche Nutzung von Unittests in Softwareentwicklungsprozessen gelegt wurden. Alle durchgeführten Projektarbeiten und Ergebnisse waren dazu notwendig und angemessen.

5 Verwertungsplan

Durch den Einsatz von Peass-CI können die Partner ihren Softwareentwicklungsprozess effizienter gestalten, wirtschaftliche Mehrwerte in Bezug auf ihre eigene Software generieren und ggf. das eigene Beratungsportfolio erweitern. Im Folgenden werden die wirtschaftlichen Erfolgsaussichten der einzelnen Partner detaillierter beschrieben.

5.1 Wirtschaftliche Erfolgsaussichten

5.1.1 evermind GmbH

Die evermind GmbH betreibt eine DevOps-as-a-Service-Sparte, in der sie Dritte bei ihrer Entwicklung unterstützt. Es finden aktuell Abstimmungen dazu statt, inwiefern parallel dazu auch Beratungen zu Performance-Engineering-Problemen angeboten werden können. Die evermind GmbH kann dabei sowohl bei der kontinuierlichen Performanceüberwachung mit den im Projekt entwickelten Tools beraten als auch aufgrund des im Projekt gewonnenen Know-hows bei der Durchführung von Lasttests und Monitoring zur Performance-Messung beraten.

Die Verwendung der *PermanEnt*-Tools führte in GeoMap direkt zu wirtschaftlichem Mehrwert, indem verschiedene Performanceprobleme behoben und somit **GeoMap performanter** gemacht wurde. Insbesondere die Behebung eines Ressourcenleaks führt dazu, dass eine potenzielle Quelle eines Systemausfalls behoben wurde. Darüber hinaus kann durch den Peass-CI Buildprozess nun **kontinuierlich** geprüft werden, ob Performanceänderungen stattgefunden haben.

5.1.2 HBT

Neben dem weiteren Einsatz der *PermanEnt*-Tools in eigenen Entwicklungsprojekten ermöglicht der Aufbau von spezifischem Know-how und die Veröffentlichung der *PermanEnt*-Tools für HBT, ihr **Beratungsportfolio** zu erweitern. Damit kann HBT dazu beraten, wie Kunden kontinuierliche Performancemessung in ihre eigenen Entwicklungsprozesse integrieren und damit Performanceprobleme finden. Darüber hinaus kann HBT dazu beraten, wie kontinuierliche Performancemessung mit anderen Werkzeugen wie Lasttests und Monitoring verbunden und priorisiert werden. Dies ergänzt das bisherige Portfolio, bei dem u.a. zu Lasttests und Performancemonitoring beraten wird.

5.1.3 IOTIQ

Im Rahmen des Projekts konnten wirtschaftliche Vorteile sowohl in der laufenden Weiterentwicklung von MobiVisor als auch im Bereich der individuellen Auftragsentwicklung für Kunden vorbereitet werden. Im Rahmen der Arbeiten für *PermanEnt* konnten neue Features geplant werden, die dazu führen, dass zum einen die responsive Nutzererfahrung von MobiVisor verbessert und zum anderen Prozesse optimiert werden, damit Akkuverbraucht reduziert werden kann. Beides sind Features, die MobiVisor einen Marktvorteil bringen und somit direkten Einfluss auf die Wirtschaftlichkeit des Unternehmens haben werden. Die bei *PermanEnt* eingesetzte Testmethodik kann auch bei der zukünftigen Auftragsentwicklung eingesetzt werden, da bei automatischer Testgenerierung weniger Entwickler-Ressourcen gebunden sind und somit mehrere Projekte gleichzeitig möglich sind. Neben der daraus resultierenden Umsatzsteigerung erhöht die verbesserte Testmethodik zugleich auch die Erfolgsaussichten sowie die Qualität der umgesetzten Projektarbeiten. Zudem wird durch die gewonnene Expertise bezogen auf Performance-Tests in Java-Umgebungen eine weitere Möglichkeit, Umsätze zu generieren, erschaffen.

5.2 Anschlussfähigkeit

5.2.1 evermind

Eine Erkenntnis im Projekt war, dass Commits, die Performanceänderungen verursachen, insbesondere in kleinen Projekten nicht häufig vorkommen. Die kontinuierliche Überwachung der Performance mit Peass-CI ist somit für kleine Unternehmen in der Regel zu kostspielig. Es finden daher aktuell Abstimmungen dazu statt, inwiefern **Benchmarking-as-a-Service** auf Basis von Peass-CI angeboten werden kann. Hierfür können die bestehenden Tools eingesetzt werden, allerdings wäre es erforderlich, Adapter für andere Technologien (GitLab CI, GitHub Actions) zu entwickeln sowie eine Mandantenfähigkeit in Peass-CI herzustellen.

5.2.2 HBT

HBT hat in der Vergangenheit Kunden erfolgreich bei der Behebung von Performanceproblemen durch Monitoring und Lasttests unterstützt. Zukünftig können Performanceprobleme, die auf spezifische Komponenten zurückzuführen sind, durch die Messung der Performance in der Continuous Integration (CI) untersucht werden. Dadurch kann ein Teil der auftretenden Performanceprobleme lokal und damit erheblich effizienter als in einem komplexen Produktivsystem untersucht werden.

In diesem Kontext treten Performanceprobleme häufig durch das Zusammenspiel von Komponenten und nicht direkt innerhalb einer Komponente auf. Die in *PermanEnt* entwickelten Tools könnten, in Kooperation mit einem Anwendungspartner, so erweitert werden, dass sie die Performanceproblemanalyse in einzelnen Komponenten unterstützen. Hierfür können, statt der Verwendung von transformierten Unittests, Lasttests zur Messung der Performance von Testumgebungen genutzt werden. Dies erfordert eine Definition eines automatisierten Starts der Testumgebungen, konzeptionelle Anpassungen bei der Testselektion, eine Anpassung der Monitoringansätze sowie die Definition einer angepassten Ursachenanalyse. Es finden derzeit Abstimmungen zu diesem Ansatz statt.

5.2.3 IOTIQ

Die Anschlussfähigkeit der Projektergebnisse ergibt sich für die IOTIQ GmbH durch die dauerhafte Integration der PUTs in ihre bisherige Testumgebung. Über die Verwendung der *PermanEnt*-Tools für die interne Weiter- und auch Neuentwicklung eigener Softwareprodukte hinaus, kann die IOTIQ vor allem auch bei der Durchführung externer Kundenaufträge von einer effizienten Erweiterung ihrer Testverfahren dauerhaft profitieren. Da die IOTIQ ein breites Spektrum verschiedener Technologien abdeckt, ist zudem die Adaption der Projektergebnisse auf **andere Programmiersprachen** eine Weiterentwicklungsmöglichkeit über die Projektdauer hinaus.

5.2.4 Universität Leipzig

Die erfolgreiche Durchführung des Projekts ermöglicht es, durch die **Kooperation mit weiteren Forschungspartnern** neue Themenfelder zu erschließend sowie daran anknüpfend weitere Forschungsarbeiten durchzuführen.

Kooperation mit weiteren Forschungspartnern Ein Transfer der Forschungsergebnisse in andere technologische Umfelder wird angestrebt. Durch die Relevanz der Performance kleiner Programmbestandteile im High-Performance-Computing bietet sich dieses als weiteres Anwendungsgebiet des PermanEnt-Ansatzes an. Zu einem Forschungsprojekt in diesem Bereich fanden erste Gespräche mit dem **Forschungszentrum Jülich** statt.

Darüber hinaus findet Forschung im Bereich Software Engineering an der **Lancaster University Leipzig** statt. David Georg Reichelt, der Projektleiter seitens der Universität Leipzig, hat nach Projektende hier eine Assistenzprofessur angetreten und entwickelt die Projektergebnisse hier auch nach Projektende weiter.

Darüber hinaus bietet das Universitätsrechenzentrum universitätsintern Dienste für wissenschaftliches Rechnen an der Universität Leipzig an, insbesondere im Bereich energiewirtschaftliche Modellierung und Digital Humanities. Es ist geplant, die Softwareperformance in diesem Bereich durch kontinuierliche Performanceüberwachung mit PUTs zu verbessern und dabei mit den entsprechenden **Lehrstühlen an der Universität Leipzig** zu kooperieren.

Weitere Forschungsarbeiten bauen auf der Methode zur Erfassung von Performanceunterschieden und den gefundenen Performanceunterschieden auf. Eine zentrale Erkenntnis der Fallstudien ist, dass Quelltextänderungen in den untersuchten Projekten seltener auftreten als erwartet, da oft Konfigurationsdateien geändert werden, und dass der größte Teil der Performanceunterschiede durch funktionale Änderungen verursacht wird. Es ergeben sich daraus drei zentrale Herausforderungen für die weitere Forschung: (1) Inwiefern werden Performanceänderungen in realen Entwicklungsprojekten durch Quelltext- und inwiefern durch Konfigurations- und Ausführungskontextänderungen verursacht? Eine breitere empirische Analyse von Performanceänderungen in Softwareentwicklungsprojekten könnte einen Aufschluss darüber geben, welche Probleme in zukünftiger Forschung stärker adressiert werden können. Hier wird eine Zusammenarbeit mit evermind GmbH und dem

Benchmarking-as-a-Service-Ansatz angestrebt. (2) Inwieweit ist es möglich, Auswirkungen von Konfigurationsänderungen auf die Performance automatisiert zu erkennen und zu untersuchen? Um die *PermanEnt* innewohnende Beschränkung auf Quelltextänderungen aufzuheben, sollten auch die Auswirkungen von Konfigurationsänderungen und geändertem Ausführungskontext auf die Performance untersucht werden. Hier wird eine Zusammenarbeit mit der HBT GmbH und der vorab dargestellten Erweiterung der *PermanEnt*-Tools angestrebt. (3) Inwiefern treten ähnliche Performanceänderungen in anderen Ausführungsumgebungen auf? Eine Übertragung auf andere Ausführungsumgebungen, bspw. emulierte Android-Apps oder Javascript-Anwendungen, kann aufzeigen, welche Muster von Performanceänderungen sprach- und technologieübergreifend auftreten.

6 Fortschritt im Software Performance Engineering

Während der Projektlaufzeit wurden parallel Forschungen im Bereich des Software Performance Engineering durchgeführt. Dabei lagen Schwerpunkte auf der **Regressionstestselektion**, der **Messung und Ursachenanalyse** und der automatisierten Codeverbesserung.

Regressionstestelektion Grambow et al. [18] untersuchen die Selektion von Benchmarks basierend auf überlappenden Funktionsaufrufen. Sie empfehlen bisher unabgedeckte Funktionsaufrufe für neue Microbenchmarks. Das Verfahren zur Selektion von Benchmarks basierend auf überlappenden Funktionsaufrufen ähnelt dabei der abdeckungsbasierten Testselektion. Darauf basierend evaluieren Grambow et al. [19] die Effizienz der reduzierten Microbenchmarksuite. Während diese zu reduzierter Ausführungszeit führt, gibt es Performanceänderungen, die nur durch die Anwendungsbenchmarks erkannt werden können. Samoaa und Leitner [20] untersuchen den Zusammenhang zwischen Microbenchmarkparametrisierung und deren Messwerten. Ihre Studie ergab, dass unterschiedliche Parametrisierungen in 40 % der Fälle keine Auswirkungen auf die Performance hatten. Durch die im Paper definierte Random-Forest-Vorhersage von Performanceänderungen könnte eine Selektion von Microbenchmarkparametern erfolgen. Da die Arbeit die Konfiguration von Microbenchmarks untersucht, unterscheidet sie sich deutlich von der Regressionstestelektion des *PermanEnt*-Ansatzes, bei der die Selektion von Unittests erfolgt. Alshoaibi et al. [21] betachtet die Bestimmung von regressionsverursachenenden Commits als mehrdimensionales Suchproblem. Die Ermittlung der Regeln zur Commiterkennung basiert auf einem evolutionären Algorithmus, der verschiedene Metriken zur Quelltextstruktur auswertet.

Messung und Ursachenanalyse Liao et al. [22] erkennen Performanceänderungen die Analyse von Produktiv-Logdaten und Performancemesszahlen, bspw. CPU-Auslastung. Durch den Vergleich alter Logdaten mit neuen Logdaten lassen sich Performanceänderungen und deren verursachende Requests erkennen. Anschließend werden durch Parsing der Aufrufbäume der Requests mögliche änderungsverusachende Methoden identifiziert. Triani et al. [23] untersuchen die Messung von Performance im stationären Zustand. Dabei wird festgestellt, dass der stationäre Zustand in Benchmarks nicht immer erreicht wird und dass Entwickler oft die nötigen Warmupiterationen unterschätzen. Swanzen et al. [24] untersuchen die Korrelation zwischen Quelltextkonstrukten und dem Erreichen des stationären Zustands. Durch eine Random-Forest-Klassifizierung gelingt es ihnen, mit einer Genauigkeit von 78,6 % vorherzusagen, ob der stationäre Zustand erreicht wird. Triani [4] untersucht den Umgang mit Performance in agilen Entwicklungsprozessen. Dabei wird festgestellt, dass Performanceprobleme auch in agilen Prozessen wasserfallartig bearbeitet werden und dass daher kontinuierliche Performancesicherstellungsaktivitäten notwendig sind. Cortelessa et al. [25] entwickeln einen Ansatz, in dem Monitoringdaten aus Microservices in UML MARTE Modelle importiert werden. Dadurch lassen sich Refactorings ableiten, die Performanceverbesserungen ermöglichen. Farah und Vergilio [26] haben einen Ansatz entwickelt, der ähnlich wie Peass Performanceänderungen von Unittests durch Messung erfasst. Der Fokus liegt allerdings stärker auf der Analyse verschiedener Performancemetriken, bspw. der CPU-Auslastung und den JVM-Events.

Die Arbeiten zur Messung bzw. Ursachenanalyse legen den Fokus auf Analyse von Benchmarkdaten bzw. Monitoingdaten aus Produktivsystemen. Dadurch unterscheiden sie sich vom vorliegenden Ansatz, der transformierte Unittests nutzt. Für Projekte mit ausreichenden Ressourcen zur Entwicklung eigener Benchmarks stellen diese Ansätze daher eine sinnvolle Ergänzung zum *PermanEnt*-Ansatz dar.

Das in *PermanEnt* verwendete Monitoringtool Kieker wurde während der Projektlaufzeit auf für Python [27], c [28] und Fortran [28] angepasst. Daduch ergibt sich die Möglichkeit, zukünftig die Ursachenanalyse für diese Sprachen zu erweitern. Javed et al. [29] erstellen ein Tool, dass, ähnlich wie Peass-CI, Performanceänderungen in CI-Systemen erkennt und eine Ursachenanalyse durchführt. Im Gegensatz zu Peass-CI führt PerfCI

allerdings keine Regressionstestselektion aus, sondern setzt eine Definition der Performancetesttasks durch den Benutzer voraus.

Automatische Verbesserung Oliveira et al. [30] untersuchen, wie Collections effizienter ausgewählt werden können, um den Energieverbrauch zu reduzieren. Sie generieren damit Verbesserungsvorschläge für existierenden Code. Mühlbauer et al. [31] untersuchen, wie unterschiedliche Workloads und unterschiedliche Systemkonfigurationen die Performance beeinflussen. Ihr Ansatz ermöglicht die Ermittlung der effizientesten Konfiguration eines Systems basierend auf dem definierten Workload. Straesser et al. [32] analysieren die Nutzung von Autoscalern, also Tools zur Verwaltung der genutzten Hardware. Durch die automatische Skalierung in Cloud-Umgebungen kann der Einsatz von Autoscalern zu einer Kostenreduktion bei gleichzeitig guter Performance beitragen.

Die Arbeiten zur automatischen Verbesserung untersuchten, wie bestehende Software effizienter implementiert oder betrieben werden kann. Der Ansatz des *PermanEnt*-Projekts hingegen ermittelt Performanceänderungen, die durch Quelltextänderungen verursacht werden. Daher stellen die Arbeiten zur Performanceverbesserung eine sinnvolle Ergänzung sowohl für die Entwicklung als auch den Betrieb von Software dar.

7 Veröffentlichungen

Die Projektergebnisse wurden in folgenden wissenschaftlichen Publikationen veröffentlicht:

- David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring: „Vision of Continuously Assuring Performance“, Symposium on Software Performance (2020). [33]
- David Georg Reichelt, Stefan Kühne, und Wilhelm Hasselbring: „Testselektion für Performanzregressionsbenchmarks in CI-Prozessen“, INFORMATIK 2021. Gesellschaft für Informatik, Bonn. [34]
- David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring: „Overhead Comparison of OpenTelemetry, inspectIT and Kieker“, Symposium on Software Performance (SSP) 2021. [35]
- David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring: „Automated Identification of Performance Changes at Code Level“, 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS), Guangzhou, China, 2022, pp. 916-925, doi: 10.1109/QRS57517.2022.00096. [11]
- David Georg Reichelt, Hannes Krauß, Stefan Kühne, und Wilhelm Hasselbring: „Generic Performance Measurement in CI: The GeoMap Case Study“, Symposium on Software Performance (SSP) 2022. [17]
- David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring. 2023. Towards Solving the Challenge of Minimal Overhead Monitoring. In Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion). Association for Computing Machinery, New York, NY, USA, 381–388. doi: 10.1145/3578245.3584851 [9]
- David Georg Reichelt. Untersuchung von Performanzveränderungen auf Quelltextebene. Dissertation, Universität Leipzig, 2023. [15]

Literatur

[1] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[2] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th ICPE*, pages 1013–1024. ACM, 2014.

[3] Shahed Zaman, Bram Adams, and Ahmed E Hassan. Security versus performance bugs: a case study on firefox. In *MSR 2011*, pages 93–102. ACM, 2011.

[4] Luca Traini. Exploring performance assurance practices and challenges in agile software development: an ethnographic study. *Empirical Software Engineering*, 27(3):74, 2022.

[5] David Georg Reichelt, Stefan Kühne, and Willhelm Hasselbring. Peass: A tool for identifying performance changes at code level. In *Proceedings of the 33rd ACM/IEEE ASE*. ACM, 2019. (in press).

[6] David G. Reichelt and Lars Braubach. Sicherstellung von performanzeigenschaften durch kontinuierliche performanztests mit dem kopeme framework. In *Software Engineering*, pages 119–124, 2014.

[7] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, pages 247–248. ACM, April 2012.

[8] Wilhelm Hasselbring and André van Hoorn. Kieker: A monitoring framework for software engineering research. *Software Impacts*, 5:100019, 2020.

[9] David Georg Reichelt, Stefan Kühne, and Willhelm Hasselbring. Towards solving the challenge of minimal overhead monitoring. In *Companion of the 2023 ACM/SPEC ICPE [Im Druck]*. ACM, 2023.

[10] David Georg Reichelt and Stefan Kühne. How to detect performance changes in software history: Performance analysis of software system versions. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 183–188, New York, NY, USA, 2018. ACM.

[11] David Georg Reichelt, Stefan Kühne, and Willhelm Hasselbring. Automated identification of performance changes at code level. In *QRS 2022 Conference Proceedings [Im Druck]*. IEEE, 2022.

[12] Lubomír Bulej, Vojtěch Horký, Petr Tuma, François Farquet, and Aleksandar Prokopec. Duet benchmarking: Improving measurement accuracy in the cloud. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 100–107, 2020.

[13] Christoph Heger, Jens Happe, and Roozbeh Farahbod. Automated root cause isolation of performance regressions during software development. In *ICPE 13*, pages 27–38, New York, USA, 2013. ACM.

[14] Nina S. Marwede, Matthias Rohr, André van Hoorn, and Wilhelm Hasselbring. Automatic failure diagnosis in distributed large-scale software systems based on timing behavior anomaly correlation. In *ECSMR*, 2009.

[15] David Georg Reichelt. *Untersuchung von Performanzveränderungen auf Quelltextebene*. Dissertation, Universität Leipzig, 2023. `https://nbn-resolving.org/urn:nbn:de:bsz:15-qucosa2-836200`.

[16] Jan Waller, Nils Christian Ehmke, and Wilhelm Hasselbring. Including performance benchmarks into continuous integration to enable devops. *ACM SIGSOFT Software Engineering Notes*, 40(2):1–4, 2015.

[17] David Georg Reichelt, Hannes Krauß, Stefan Kühne, and Wilhelm Hasselbring. Generic performance measurement in ci: The geomap case study. *Softwaretechnik-Trends [Im Druck]*, 2022.

[18] Martin Grambow, Christoph Laaber, Philipp Leitner, and David Bermbach. Using application benchmark call graphs to quantify and improve the practical relevance of microbenchmark suites. *PeerJ Computer Science*, 7:e548, 2021.

[19] Martin Grambow, Denis Kovalev, Christoph Laaber, Philipp Leitner, and David Bermbach. Using microbenchmark suites to detect application performance changes. *IEEE Transactions on Cloud Computing*, 2022.

[20] Hazem Samoaa and Philipp Leitner. An exploratory study of the impact of parameterization on jmh measurement results in open-source projects. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 213–224, 2021.

[21] Deema Alshoaibi, Mohamed Wiem Mkaouer, Ali Ouni, AbdulMutalib Wahaishi, Travis Desell, and Makram Soui. Search-based detection of code changes introducing performance regression. *Swarm and Evolutionary Computation*, 73:101101, 2022.

[22] Lizhi Liao, Jinfu Chen, Heng Li, Yi Zeng, Weiyi Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi. Locating performance regression root causes in the field operations of web-based systems: An experience report. *IEEE Transactions on Software Engineering*, 48(12):4986–5006, 2021.

[23] Luca Traini, Vittorio Cortellessa, Daniele Di Pompeo, and Michele Tucci. Towards effective assessment of steady state performance in java software: are we there yet? *Empirical Software Engineering*, 28(1):13, 2023.

[24] Jared Chad Swanzen, Kyle Thomas Botes, Husnaa Molvi, Omphile Monchwe, Dan Phala, and Dustin van der Haar. Analysing static source code features to determine a correlation to steady state performance in java microbenchmarks. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, pages 89–93, 2023.

[25] Vittorio Cortellessa, Daniele Di Pompeo, Romina Eramo, and Michele Tucci. A model-driven approach for continuous performance engineering in microservice-based systems. *Journal of Systems and Software*, 183:111084, 2022.

[26] Paulo Roberto Farah and Silvia Regina Vergilio. Perfort: A tool for software performance regression. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, pages 119–120, 2023.

[27] Serafim Simonov, Thomas Düllmann, Reiner Jung, and Sven Gundlach. Instrumenting python with kieker. *Symposium on Software Performance*, 2022.

[28] Reiner Jung, Sven Gundlach, and Wilhelm Hasselbring. Instrumenting c and fortran software with kieker. *Symposium on Software Performance*, 2021.

[29] Omar Javed, Joshua Heneage Dawes, Marta Han, Giovanni Franzoni, Andreas Pfeiffer, Giles Reger, and Walter Binder. Perfci: a toolchain for automated performance testing during continuous integration of python projects. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1344–1348, 2020.

[30] Wellington Oliveira, Renato Oliveira, Fernando Castor, Gustavo Pinto, and João Paulo Fernandes. Improving energy-efficiency by recommending java collections. *Empirical Software Engineering*, 26:1–45, 2021.

[31] CKJDSA Stefan Mühlbauer, Florian Sattler, and N Siegmund. Analyzing the impact of workloads on modeling the performance of configurable software systems. In *Proceedings of the International Conference on Software Engineering (ICSE), IEEE*, 2023.

[32] Martin Straesser, Simon Eismann, Jóakim von Kistowski, André Bauer, and Samuel Kounev. Autoscaler evaluation and configuration: A practitioner's guideline. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*, ICPE '23, page 31–41, New York, NY, USA, 2023. Association for Computing Machinery.

[33] David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring. Vision of continuously assuring performance. *Symposium on Software Performance*, 2020.

[34] David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring. Testselektion für performanzregressionsbenchmarks in ci-prozessen. *INFORMATIK 2021*, 2021.

[35] David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring. Overhead comparison of opentelemetry, inspectit and kieker. 2021.