Performanzanalyse von Softwaresystemversionen: Methode und erste Ergebnisse

David Georg Reichelt
Institut für Angewandte Informatik e.V.
reichelt@informatik.uni-leipzig.de

Johannes Schmidt
Universität Leipzig
jschmidt@informatik.uni-leipzig.de

Abstract: Jede Änderung an einem Softwaresystem kann eine Performanzveränderung hervorrufen. Dieser Zusammenhang ist jedoch bisher nicht umfassend empirisch untersucht worden. In diesem Beitrag wird ein Vorgehen zur Performanzanalyse von Softwareversionen beschrieben, mit dessen Hilfe die systematische Quantifizierung von Performanzveränderungen möglichen ist. Auf Basis erkannter Performanzveränderungen können anschließend Performanzprobleme abgeleitet werden. Als Anwendungsbeispiel wurde das Open-Source-Rahmenwerk Apache Commons IO untersucht.

1 Einleitung

Die ISO 9126 [ISO01] beschreibt sechs Qualitätskriterien für Software. Eines davon ist die Effizienz in Hinblick auf Zeit- und Ressourcenverbrauch (auch Performanz genannt). Performanzprobleme sind bereits mit Fokus auf Softwarearchitektur untersucht worden [SW03]. Grundannahme dieser Arbeit ist, dass Performanzprobleme auf Quelltextebene existieren. Ein Performanzproblem auf Quelltextebene liegt genau dann vor, wenn durch ein Code-Refactoring unter Erfüllung der geforderten funktionalen Anforderungen die Performanz maßgeblich verbessert werden kann. Es wird ein Ansatz zur Erkennung solcher Performanzprobleme vorgeschlagen, der auf Messung der Performanz der Testfälle für jeden im Repository abgelegten Versionsstand eines Projektes basiert. Liegt eine signifikante Veränderung der Performanz vor, wird geprüft, ob es sich um ein Performanzproblem handelt. Dies ist nicht immer der Fall, da sich bspw. die Implementierung des Testfalls verändern kann, ohne dass sich die zu testende Methode verändert. Ziel ist es, aus der Analyse der Performanzprobleme eine Problemklassifikation aufzubauen, die im Softwarentwicklungsprozess bei der Bewertung von Performanzänderungen unterstützen kann.

Dieser Beitrag gliedert sich in vier Teile. In Abschnitt 2 wird die Methode der Performanzanalyse von Softwaresystemversionen beschrieben. Anschließend werden erste empirische Ergebnisse in Abschnitt 3 dargestellt. In Abschnitt 4 wird der vorliegende Ansatz mit verwandten Arbeiten in Beziehung gesetzt. Abschließend erfolgt in Abschnitt 5 eine Zusammenfassung.

2 Methode der Performanzanalyse von Softwaresystemversionen

Der Ansatz der Performanzanalyse von Softwaresystemversionen umfasst drei Phasen: (1) Messung der Performanz aller Testfälle für jeden Quelltextversionsstand, (2) Identifikation von Performanzveränderungen und (3) Identifikation von Performanzproblemen. Bei der folgenden Beschreibung dieser Phasen wird auch auf die für Java-Projekte notwendige Werkzeugunterstützung eingegangen. Eine Erweiterung auf andere Sprachen ist möglich.

2.1 Messung der Performanz

Funktionale Anforderungen werden in Testfällen überprüft. Es wird davon ausgegangen, dass eine Beziehung zwischen der Performanz der Testfälle für funktionale Anforderungen und der Performanz des Gesamtsystems besteht. Daher werden alle Tests eines Versionsstandes zur Performanzmessung herangezogen. Der Performanzverlauf eines Testfalls ergibt sich aus der Messung der Performanz über alle Versionsstände. Sind Versionsstände wie bspw. bei CVS nicht über das Gesamtprojekt definiert, ist eine Messung nicht möglich.

Die Erhebung der Performanzverläufe erfolgt durch das in Abbildung 1 dargestellte, vollständig automatisierte Vorgehen. Voraussetzung ist, dass der Quelltext aus einem Versionskontrollsystem wie bspw. SVN oder git abgerufen werden kann. Anfangs wird der zu testende Versionsstand geladen. Anschließend werden die Testfälle um Quelltext zur Performanzmessung erweitert. Die erweiterten Tests werden nachfolgend Performanztests genannt. Der Messquelltext muss an die Besonderheiten des jeweils verwendete Testframeworks, bspw. JUnit 3 oder 4, angepasst werden. Durch Hintergrundprozesse und JVM-Laufzeitoptimierungen können Performanzwerte des gleichen Performanztests starke Abweichungen aufweisen. Bei der Testausführung muss deshalb berücksichtigt werden, dass zur Durchführung von validen Performanzmessungen mehrere Aufwärm-Durchläufe und anschließend weitere Testausführungen notwendig sind, aus denen dann statistische Kennwerte berechnet werden können. Mit Hilfe des KoPeMe-Framework [RB14] können diese Anforderungen erfüllt werden. Daher wurde es genutzt, um Testfälle anzupassen, Performanzwerte zu messen und diese anschließend zu speichern. Zur Einbindung der Performanztests werden Anpassungen am Buildprozess, d.h. an den jeweiligen Ant- oder maven-Skripten, vorgenommen. Anschließend kann die Testausführung erfolgen.

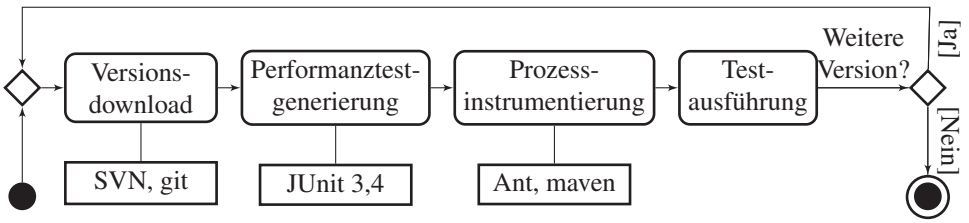

Abbildung 1: Messung von Performanzverläufen

2.2 Identifikation von Performanzveränderungen

Anhand der Messwerte aus der ersten Phase lassen sich Testfälle in aufeinanderfolgenden Versionsständen identifizieren, die potentiell Performanzveränderungen aufweisen. Zur detaillierten Untersuchung werden diese Tests erneut mit einer hohen Wiederholungsrate ausgeführt. Sofern diese Messungen mit Sicherheit auf eine Performanzveränderung hinweisen, erfolgt eine Auswertung der Änderungen im Quelltext. Dieses Vorgehen ist in Abbildung 2 dargestellt.

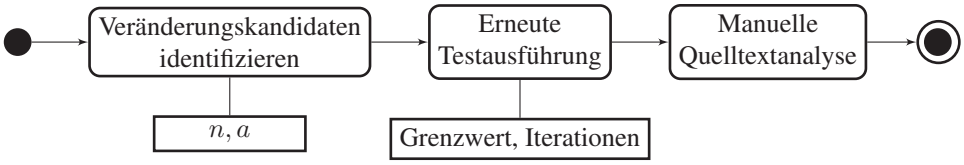

Abbildung 2: Vorgehen zu Identifikation von Performanzveränderungen

Für die Identifizierung potentieller Performanzveränderungen wird der gleitende Durchschnitt der Mittelwerte (über die Performanzmesswerte eines Versionsstandes) über n Versionsstände berechnet. Weicht dieser gleitende Durchschnitt von n Messwerten um eine relative Abweichung a ab, ist der betrachtete Testfall an diesem Versionsübergang ein Performanzveränderungskandidat. Die Parameter n und a sind ausreichend groß zu wählen, um eine sinnvolle Vorselektion zu gewährleisten und zudem ausreichend klein zu wählen, um eine geringe Anzahl falsch negativer Resultate zu erhalten.

Für die identifizierten Kandidaten werden die Performanztests mit einer hohen Wiederholrate erneut ausgeführt. Die resultierenden Messwerte werden als empirische Verteilungen betrachtet. Als Abstandsmaß wird die Kullback-Leibler-Divergenz berechnet. Liegt diese über einem separat zu wählenden Grenzwert, ist davon auszugehen, dass die Performanzveränderung wiederholt messbar ist und somit eine Ursache im Quelltext hat.

Liegt eine Performanzveränderung vor, wird ermittelt, ob diese auf eine Änderung in der Software oder im Testfall zurückzuführen ist. Hierzu ist eine Inspektion des Testfallcodes notwendig. Hat sich der Quelltext der Testmethode nicht geändert, handelt es sich um eine Performanzveränderung der Software.

2.3 Identifikation von Performanzproblemen

Aus den erkannten Performanzveränderungen der Software werden im letzten Schritt Performanzprobleme abgeleitet. Eine einfache Herangehensweise wäre, die Performanzprobleme in der Version mit den schlechteren Performanzmesswerten anzunehmen. Der als schlechter bewertete Quelltextversionsstand muss untersucht werden. Bei der Quellcodeanalyse werden folgende Fälle unterschieden: (1) Änderung der funktionalen Anforderungen und (2) Quelltext-Restrukturierung mit gleichbleibenden funktionalen Anforderungen. In Fall (1) kann keine generelle Aussage über das Vorliegen eines Performanzproblems ge-

troffen werden, da sich neue Funktionalitäten sowohl positiv als auch negativ auf die Testfallperformanz auswirken können. In Fall (2) hingegen ist davon auszugehen, dass ein Performanzproblem vorliegt. Sollen Performanzprobleme automatisiert identifiziert werden, müssten demnach funktionale Änderungen bzw. deren Abwesenheit erkannt werden. Hierzu ist weitere Forschung notwendig. Im Rahmen des vorliegenden Beitrags wurden daher Performanzveränderungen manuell mit Hilfe von Codedifferenzen hinsichtlich der Performanzprobleme untersucht.

3 Empirische Ergebnisse

Zur Unterstützung der Performanzanalyse von Softwaresystemversionen wurde ein prototypisches Werkzeug implementiert. Damit wurde das Projekt Apache Commons IO[1] analysiert. Zur Beschleunigung der Experimente wurden Revisionsbereiche auf mehrere Desktop-Rechnern mit 2,6 GHz, 8 Kernen und 8 GB RAM verteilt. Die insgesamt messbaren 1121 Testfälle wurden 10 mal zum Aufwärmen der JVM und 50 mal zur Messung ausgeführt.

Bei der Erprobung der Methode der Performanzanalyse von Softwaresystemversionen wurde die Analyse des Quelltextes manuell ausgeführt. Das prototypische Werkzeug stellte benötigte Informationen, d.h. Testfallname, Wertveränderung und Diff-Anzeige der Versionsverwaltung bereit und unterstütze so die Analyse.

Nach Durchführung der Experimente sind n und a so zu wählen, dass die Erkennung der Performanzveränderungen möglichst viele richtig positive Zuordnungen und möglichst wenig falsch positive Zuordnungen enthält. In ersten Experimenten konnten n=5 und a=40% als gute Parameter identifiziert werden. Hiermit wurden 158 Performanzveränderungskandidaten identifiziert. Mit einer erneuten Testausführung mit 600 Iterationen und einem Grenzwert von 30% konnten elf Performanzveränderungen identifiziert werden. Hiervon sind vier auf geänderte Methoden-Aufrufreihenfolgen zurückzuführen, die eine veränderte Stackgröße verursachten, zwei auf die Entfernung unnötiger Verarbeitungsschritte, zwei auf geänderte bzw. korrekt implementierte funktionale Anforderungen und zwei auf die Behebung von Synchronisationsproblemen. Eine Performanzveränderung konnte keiner Codeänderung zugeordnet werden, obwohl sie auch bei wiederholter Ausführung messbar war.

Da n und a hoch gewählt wurden, sind Performanzveränderungen übersehen worden. Beispielsweise führte die Umstellung von der alten Java-IO Schnittstelle auf Java NIO[2] zu einer Performanzveränderung in verschiedenen Testfällen, die nur bei niedrigerer Wahl von n und a als solche identifiziert wurde. Eine Durchführung der Experimente mit größerer Iterationsanzahl und eine anschließende Ergebnisanalyse mit niedrigerem n und a sowie geringerem Grenzwert bei der erneuten Testausführung würde voraussichtlich mehr Performanzprobleme identifizieren.

[1] http://commons.apache.org/proper/commons-io/

[2] Java Non-Blocking IO, Bibliothek die große I/O-Operationen effizienter implementiert

4 Verwandte Arbeiten

Arbeiten zur Analyse der Performanz verschiedener Versionen von Software sind nicht bekannt. Es existieren verwandte Arbeiten in den Forschungsbereichen Software Performance Engineering (SPE) und Mining Software Repositories (MSR).

Im SPE wurde in [SW03] beschrieben, welche Software-Performanz-Antipatterns in Praxisprojekten festgestellt werden konnten. Die Verbreitung dieser Antipatterns wurde nicht empirisch untersucht. Unter anderem in [HHF13] und [NAJ+12] wurde betrachtet, wie Performanzverschlechterungen in Regressionstests effizienter gefunden bzw. untersucht werden können. [HHF13] liefert hierbei ein Vorgehen, um den Methodenaufruf, der die Performanzverschlechterung zwischen zwei ggf. nicht aufeinanderfolgenden Einreichungen verursacht, zu finden. Hierbei wird [vHWH12] für die Performanzmessung genutzt. Diese Methode ließe sich für die Untersuchung gefundener Performanzveränderungen verwenden.

Die Arbeiten zum Green Mining und zur Identifikation von Performanzproblemen in Bugtrackern aus dem Forschungsfeld MSR sind für diese Arbeit relevant. Im Green Mining werden Energieverbrauchseigenschaften von Softwareversionen gemessen. Beispielsweise analysiert [HWR+14] die Entwicklung des Energieverbrauchs von Android-Anwendungen. Das methodische Vorgehen ähnelt dem Vorgehen des vorliegenden Beitrags, da ebenfalls Laufzeiteigenschaften von Software in Quelltextversionen ausgewertet werden.

[NJT13] und [JSS+12] analysieren Performanzprobleme in öffentlich zugänglichen Bugtrackern. [JSS+12] nimmt eine Klassifikation von Performanzfehlern (engl. *Performance Bugs*) vor. Hierbei werden drei Fehlertypen unterschieden: Unkoordinierte Funktionen (engl. *Uncoordinated Functions*), Überspringbare Funktionen (engl. *Skippable Functions*) und Synchronisationsprobleme (engl. *Synchronization Issues*). [NJT13] schlussfolgert aus einer statistischen Auswertung der Bugtracker, dass Performanzfehler meist nach Codebetrachtung und nicht durch Benutzerangaben oder Profiling gefunden werden. Diese beiden Arbeiten liefern einen wertvollen Beitrag zur Klassifizierung von Performanzproblemen. Ihre Forschungsergebnisse werden jedoch auf indirekterem Weg gewonnen: Performanzfehler müssen erst in Bugtracker eingetragen werden, während durch Unit-Tests gemessene Performanzprobleme direkt auf Quelltextebene nachweisbar sind.

5 Zusammenfassung und Ausblick

Performanzprobleme sind auf Basis von Performanztests auffindbar. Hierzu wurde eine Methode zur Performanzanalyse von Softwaresystemversionen vorgestellt und anhand eines großen Open-Source-Rahmenwerks erprobt. Grundidee ist es, die Performanzentwicklung von Testfällen zu untersuchen und hierbei typische Performanzprobleme zu erkennen und zu klassifizieren. Die ersten Resultate zeigen, dass Performanzveränderungen zuverlässig identifiziert werden können. Die in dieser Arbeit manuell identifizierten Performanzprobleme sind auf verschiedene Ursachen zurückzuführen, aus der sich derzeitig keine Performanzproblemklassifikation ableiten lässt. Durch die Wiederholung der Experimente mit anderen Parametern können voraussichtlich mehr Performanzproblemen erkannt werden.

Anschließend kann eine verbesserte Unterstützung der Einstufung der Performanzprobleme umgesetzt werden. Durch eine Verbindung mit einer Methode zur Identifikation von Ursachen von Performanzveränderungen wie [HHF13] kann das Auffinden von Problemursachen beschleunigt werden. Die Analyse weiterer Softwareprojekte sollte die Quantifizierung des Auftretens von Performanzproblemen verbessern. Durch die Analyse von Projekten aus dem betrieblichem Kontext könnte ein Zusammenhang zwischen dem Projekttyp und dem Auftreten bestimmter Performanzprobleme identifiziert werden. Weiterhin wäre eine Untersuchung sinnvoll, in welchen Entwicklungsphasen Performanzprobleme auftreten und welche SPE-Techniken Performanzprobleme unter welchen Bedingungen verhindern können. Sobald eine valide Klassifikation von Performanzproblemen bekannt ist, können neue SPE-Methoden so gestaltet werden, dass sie bei der Vermeidung von Performanzproblemen unterstützen.

Danksagung Diese Forschung wurde im Rahmen des Projektes „CVtec“ durch das Bundesministerium für Bildung und Forschung gefördert (Förderkennzeichen: 01IS14016). Wir danken dem Rechenzentrum der Universität Leipzig für die Bereitstellung der Rechenressourcen für die Durchführung der Performanztests.

Literatur

[HHF13] Christoph Heger, Jens Happe und Roozbeh Farahbod. Automated Root Cause Isolation of Performance Regressions During Software Development. In *ICPE 13*, Seiten 27–38, New York, USA, 2013. ACM.

[HWR+14] Abram Hindle, Alex Wilson, Kent Rasmussen, E. Jed Barlow, Joshua Charles Campbell und Stephen Romansky. GreenMiner: A Hardware Based Mining Software Repositories Software Energy Consumption Framework. In *MSR 2014*, Seiten 12–21, New York, USA, 2014. ACM.

[ISO01] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[JSS+12] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz und Shan Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47:77–88, 2012.

[NAJ+12] Thanh H.D. Nguyen, Bram Adams, Zhen M. Jiang, Ahmed E. Hassan, Mohamed Nasser und Parminder Flora. Automated Detection of Performance Regressions Using Statistical Process Control Techniques. In *ICPE*, Seiten 299–310, New York, USA, 2012. ACM.

[NJT13] Adrian Nistor, Tian Jiang und Lin Tan. Discovering, reporting, and fixing performance bugs. In *MSR 2013*, Seiten 237–246. IEEE Press, 2013.

[RB14] David G. Reichelt und Lars Braubach. Sicherstellung von Performanzeigenschaften durch kontinuierliche Performanztests mit dem KoPeMe Framework. In *SE*, Seiten 119–124, 2014.

[SW03] Connie U Smith und Lloyd G Williams. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *CMG Conference*, Seiten 717–725. Citeseer, 2003.

[vHWH12] André van Hoorn, Jan Waller und Wilhelm Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *ICPE 2012*, Seiten 247–248. ACM, April 2012.