

Skalierbare Performanzanalyse durch Prolongation

Moritz Hammer¹, Bernhard Kempter², Florian Mangold¹, Harald Roelle²

¹ Institut für Programmierung und Softwaretechnik, LMU München,
Oettingenstraße 67, 80538 München
{hammer, mangold}@pst.ifi.lmu.de

² CT SE 1, Siemens AG,
Otto-Hahn-Ring 6, 81730 München
{Bernhard.Kempter, Harald.Roelle}@siemens.com

Abstract: Die praxisrelevante Performanzanalyse großer, nebenläufiger Systeme ist unzureichend gelöst. Das Zusammenwirken von Modulen, und damit die teilweise Ursache von Performanzproblemen kann nur sehr schwer erkannt werden. Durch künstliche Verlängerungen der Laufzeit einzelner Module können Hinweise auf Zusammenhänge im System erschlossen werden. Das System wird dazu unter einer statistischen Perspektive betrachtet, die Verlängerungen und die Reaktionen im System können als Varianzen interpretiert werden. Diese Varianzen können mit multivariaten Analysemethoden ausgewertet werden, wie dies anhand der Faktorenanalyse praktisch gezeigt wird.

1 Einleitung

Moderne Softwaresysteme bestehen oft aus einer schier unüberschaubaren Anzahl verschiedener Komponenten, die meist extern entwickelt wurden und über deren Interna nur wenige Informationen vorliegen. So benötigen selbst kleinere Webanwendungen, die auf dem SPRING FRAMEWORK [Joh02] basieren, einige MB externer JAR-Files. Bei der Verwendung komplexerer Bibliotheken kann die Größe leicht auf Werte über 50MB anwachsen. Kommt es in solchen Systemen zu Performanzproblemen stellt ihre schiere Größe bereits ein großes Problem für effektives Profiling dar.

Ist das betrachtete System zudem noch nebenläufig – wie z.B. eine Webanwendung – kommen zusätzliche Effekte hinzu, die sich aus dem Laufzeitverhalten der isolierten Komponenten alleine nicht mehr erklären lassen. Beim klassischen Profiling können die so entstehenden Effekte nur noch unzureichend erfasst werden.

Diese Arbeit stellt eine neue Methode vor, um automatisiert Abhängigkeiten in großen, nebenläufigen Systemen aufzufinden und zu analysieren. Dabei wird das System wiederholt ausgeführt und die Laufzeit einzelner Codefragmente an geeigneten Stellen verändert. Da im Allgemeinen Systeme nicht einfach schneller gemacht werden können, werden diese künstlich verlangsamt (im Folgenden als Prolongieren bezeichnet). Die dabei gemessenen Änderungen des Laufzeitverhaltens lassen Rückschlüsse auf die Architektur des

Systems zu – es wird also eine Art Reverse Engineering des Performanzverhaltens durchgeführt. Dabei kann die Granularität der als atomar betrachteten Einheiten – in dieser Arbeit als „Modul“ bezeichnet – beliebig festgelegt werden: das System kann beispielsweise auf Methoden-, Klassen- oder Komponentenebene analysiert werden. Da die Technik des Prolongierens zudem sehr einfach zu realisieren ist, ist unser Ansatz weitestgehend unabhängig von der verwendeten Programmiersprache. Geeignete Mittel vorausgesetzt, kann dabei sogar auf eine Analyse oder eine Modifikation des Quellcodes verzichtet werden – beispielsweise kann das Prolongieren für in JAVA implementierte Systeme mittels ASPECTJ [KHH⁺01] durchgeführt werden.

Die durch die Prolongation erhobenen Daten über Abhängigkeiten zwischen Modulen (d. h. Methoden/Klassen/Komponenten ...) können mit statistischen Analyseinstrumentarien weiterverarbeitet werden, um die relevanten Informationen zu extrahieren. Wir stellen eine Methode vor, die die bedeutendsten Abhängigkeiten automatisiert erkennt und visualisiert. Hierdurch skaliert unser Ansatz noch besser, da auf diese Weise die unüberschaubare Menge der Informationen auf wenige Faktoren aggregiert werden.

Dieser Artikel ist folgendermaßen gegliedert: Kapitel 2 gibt einen Überblick über existierende Ansätze der Performanzanalyse. In Kapitel 3 werden die Grundlagen unseres Ansatzes dargelegt und die betrachteten Systeme eingegrenzt. In Kapitel 4 wird dann die praktische Anwendung beschrieben und die Technik anhand eines realen Webcrawler demonstriert, bevor in Kapitel 5 ein Ausblick auf mögliche Erweiterungen gegeben wird.

2 Stand der Technik

Performanzanalyse bestehender Systeme ist ein weithin beachtetes Feld, da Performanzprobleme bei sonst voll funktionaler Software als suboptimales Verhalten wahrgenommen werden, Kosten verursachen und im Extremfall zur Gebrauchsunfähigkeit der Software führen können.

Bei *Lasttests* werden Laufzeiten des realen Systems, mit ähnlichen Eingaben wie in der Praxis zu erwarten sind, gemessen. Dabei wird das System unter „Stress“ oder „Last“ gesetzt, um Grenzen aufzudecken und das Verhalten des Systems beim Erreichen dieser Grenzen zu analysieren. Der Vorteil dieses Vorgehens ist, dass keine Analyse des Codes gemacht werden muss und die Granularität der betrachteten Einheiten sehr grob ist, was die Analyse von beliebig komplexen Systemen zulässt. [Reu07] gibt einen guten Überblick über aktuell eingesetzte Software zur Laufzeitanalyse bzw. Lasttests.

Im Gegensatz zu unserem Ansatz werden bei Lasttests jedoch keine Performanz-Zusammenhänge zwischen den bestehenden Modulen betrachtet.

Profiler liefern genaue Messungen vom Laufzeitverhalten einzelner Module. Somit ist es möglich, Module zu identifizieren, die für sich einen hohen Zeitverbrauch aufweisen. Diese Module müssen jedoch nicht unbedingt die Verursacher der schlechten Performanz eines Systems sein. Es ist möglich, dass diese Module aufgrund anderer Einflüsse (z.B. zu lange gehaltene Sperren seitens anderer Module) schlechte Laufzeiten aufweisen.

Sind Module mit schlechter Performanz durch Laufzeitmessungen identifiziert, können diese modifiziert werden, in diesem Kontext spricht man auch von „tunen“. Die Intention dieses Modifizierens ist es, eine bessere Performanz im Gesamtsystem zu erreichen. Einige moderne Profiler bieten dafür ein „Snapshot-Differencing“ an [Sof07], womit Co-variationen direkt im System gemessen werden können.

Beim Profiling wird zumeist ein Call-Tree analysiert. Dies hat allerdings den Nachteil, dass nur gegenseitige Aufrufe ausgemacht werden können. Call-Trees geben daher nur bedingt Aufschluss über die performanzmindernden Eigenschaften eines (asynchronen) Systems. Unser Ansatz versucht diese Einschränkung durch das Auffinden von performanzmindernde Abhängigkeiten, wie gegenseitige Ressourcennutzung, zu beseitigen.

Neben der Analyse bestehender Systeme besteht viel Interesse an einer Berücksichtigung von Performanzaspekten in den *Modellen* der Analyse- und Designphase eines Softwareprojekts. [Woo00] schlägt vor, (Pfad-)Modelle aufgrund von Szenarios zu bilden. Für diese Szenario-Modelle werden unter anderem UML Aktivitäts- oder Sequenzdiagramme verwendet [PW05]. Diese Diagramme bilden den Fluss zwischen Ausführungen, Operationen, Aktivitäten und Verantwortlichkeiten ab. Den Komponenten wird der zu erwartende Ressourcenbedarf zugewiesen. Diese Szenariomodelle können in Performanzmodelle transformiert werden und geben so Aufschluss über die zu erwartende Performanz des Systems.

Andere Methoden umfassen z.B. die Untersuchung von Performanz verteilter Systeme in frühen Projektphasen an Hand von automatisch aus annotierten UML-Modellen generierten Prototypen [HWPL04]. Die generierten Prototypen werden dann in der Infrastruktur verteilt und deren Performanz durch anschließende Lasttests ermittelt.

[BGdMT98] gibt einen guten Überblick über weitere verwendete Modellierungsmethoden bei der Performanzanalyse. Es werden Queueing Systems, Queueing Networks, Markov Ketten mit diskreter und stetiger Zeit und Simulationen eingeführt, die zur Performanzanalyse verwendet werden.

Unser Ansatz konzentriert sich jedoch auf eine Analyse bereits implementierter Systeme. [SW02] spricht hier treffend von einer Feuerwehr-Performanzanalyse zum Tuning des Systems.

Um von einem existierenden System ein Modell zu bilden, wird Reverse Engineering angewandt, was auch im Kontext der Performanzanalyse versucht wird. [LCH04] verfolgen dabei einen Ansatz, der mit unserem am ehesten vergleichbar ist. Dabei werden statische und dynamische (also durch Ausführung des Systems durchgeführte) Analysen kombiniert, allerdings bleibt der Ansatz auf die Granularität von Methoden beschränkt.

Unser Ansatz ist im Vergleich zu den erwähnten Ansätzen als praktikabler anzusehen. Die Performanzanalyse durch Prolongation ist skalierbar, sie kann auf beliebig große Systeme angewandt werden, was einen enormen Vorteil zur statischen Codeanalyse darstellt. Insbesondere wird die in dieser Arbeit vorgestellte Performanzanalyse an einem laufenden System durchgeführt, Seiteneffekte durch Interaktionen mit dem Betriebssystem oder ähnliche Wechselbeziehungen können erkannt werden.

3 Grundlagen

3.1 Abhängigkeit von Modulen

Wir betrachten Softwaresysteme, die sich aus *Modulen* zusammensetzen. Ein Modul ist die kleinste betrachtete Einheit und kann von beliebiger Granularität sein. Beispielsweise können die Module durch die Methoden, die Klassen oder die größeren Komponenten (z.B. Paketen oder ganzen Bibliotheken wie JAR-Files oder DLLs) eines Softwaresystems gegeben sein. Wir betrachten eine *Abhängigkeitsrelation* zwischen Modulen:

Definition 3.1. Ein Modul m heißt *abhängig* von einem Modul m' , wenn eine Veränderung von m' eine Veränderung der Performanz von m bewirkt.

Ein Modul m heißt *abhängig*, wenn es ein anderes Modul m' gibt, von dem es abhängig ist, ansonsten heißt es *unabhängig*.

So ist beispielsweise eine Methode m von allen Methoden m_1, \dots, m_n , die von m aufgerufen werden, im Normalfall abhängig, da die Ausführungsdauer von m sich aus der Ausführungsdauer der aufgerufenen Methoden (und der Dauer der eigenen Anweisungen) zusammensetzt. Die Performanzänderung einer Methode $m_i, 1 \leq i \leq n$ wird also normalerweise auch eine Performanzänderung von m bewirken. Methoden können jedoch auch abhängig voneinander sein, ohne sich aufzurufen, wenn z.B. in einem nebenläufigen System gemeinsame Ressourcen verwendet werden, und somit eine Veränderung der Ressourcennutzung einer Methode eine Performanzänderung bei einer anderen Methode bewirkt, die ebenfalls auf diese Ressource zugreift.

Ein Modul ist unabhängig, wenn es in seiner Performanz von allen anderen Modulen unbeeinflusst ist (oder zumindest die Performanz des Moduls vernachlässigbar gering abhängig von anderen Modulen ist). Bei Methoden trifft dies beispielsweise auf einfache Funktionen zu, die selbst keine Methodenaufrufe durchführen und ohne bedeutende Ressourcennutzung auskommen.

Die Abhängigkeit zwischen Modulen ist von Bedeutung, da hierdurch Optimierungspotential aufgezeigt wird: Da die Laufzeit eines abhängigen Moduls m von wenigstens einem anderen Modul m' abhängt, kann eine Änderung von m' – die nicht notwendigerweise eine Performanzverbesserung für m' bedeuten muss – die Performanz von m verbessern. Dies ist insbesondere in nebenläufigen Systemen interessant, wo z.B. durch angepasste Ressourcennutzung Race-Conditions vermieden werden können.

3.2 Prolongation

Die grundlegende Idee dieses Artikels besteht darin, durch künstliche Veränderung einzelner Module Abhängigkeiten der Laufzeit zu erkennen. Dafür sind zwei Schritte notwendig:

1. Die Laufzeiten der einzelnen Module müssen erhoben werden können. Beispiels-

weise müssen für Methoden die Zeiten zwischen Aufruf und Rückkehr gemessen werden.

2. Das Laufzeitverhalten einzelner Module muss geändert (prolongiert) werden können, ohne dabei funktionale Änderungen herbeizuführen. Dies kann beispielsweise durch das Hinzufügen von Wartezeit geschehen.

Eine mögliche Realisierung dieser Schritte wird in 4.1 vorgestellt.

Bei der Prolongation wird nun wiederholt das System ausgeführt, wobei die Laufzeit der einzelnen Module gemessen wird und pro Durchlauf ein anderes Modul prolongiert wird. Wird das Modul m verlängert, ändern sich auch die Laufzeiten der von m abhängigen Module.

Die gemessenen Daten des variierten Systems können nun vielfältig analysiert werden. Eine naheliegende Möglichkeit besteht darin, ausgehend von einem Referenzlauf des nicht variierten Systems, Abhängigkeiten zu bestimmen. Unabhängige Module sind dabei diejenigen, deren Laufzeitverhalten nur dann vom Referenzlauf abweicht, wenn sie selbst prolongiert wurden.

In dieser Arbeit wird aber ein anderen Ansatz zur Analyse vorgestellt. Dabei betrachten wir das System aus einer statistischen Perspektive. Die Laufzeiten der Module sind die Variablen, deren Zusammenhänge im ersten Moment unklar sind, aber analysiert werden sollen.

Durch die Prolongation wird künstlich Varianz in den einzelnen Modulen erzeugt. Mit dieser Varianz kann das System analysiert werden, da abhängige Module durch Prolongation die Varianz „weitertragen“. Ist das Modul m abhängig von m' , korrelieren die Varianzen von m und m' nach einer Prolongation (der künstlich eingefügten Varianz) von m' .

3.3 Faktorenanalyse

Es ist praktisch nahezu unmöglich aus den Daten eines gemessenen Systems ohne eine Analysemethode die Auswirkung der einzelnen Module auf die Gesamtperformanz festzustellen. Durch die große Anzahl der gemessenen Module sind die Daten unübersichtlich und bieten keinen Anhaltspunkt für eine leicht zugängliche Interpretation. Zudem sind viele der Abhängigkeiten redundant: Ist Modul a von Modul b abhängig, das wiederum von c abhängig ist, wird normalerweise auch die Abhängigkeit von a und c in den Daten zu finden sein.

Deshalb wurde von uns die Faktorenanalyse [Spe04, Bac96], insbesondere die Hauptkomponentenanalyse, zur Untersuchung der Daten verwandt. Die Faktorenanalyse nimmt eine Dimensionsreduzierung der multidimensionalen (multivariaten) Daten vor, wodurch die dahinter liegenden Ursachen oder Faktoren entdeckt werden.

Die Faktorenanalyse wird als Analyseinstrumentarium eingesetzt. Es kann damit verstanden werden, welche Module im System konkret zusammenwirken. Diese Kenntnis hat ein Entwickler nicht a priori. Code wird wiederverwendet, Systeme werden verteilt entwi-

ckelt. Es ist nicht bekannt, welche *Faktoren* bei der Performanz des Systems eine Rolle spielen, wie die Module zusammenarbeiten und wodurch konkret die schlechten Laufzeiteigenschaften resultieren.

Informell kann ein Faktor in unserer Arbeit als eine Kombination von Modulen verstanden werden, die sich unter einer Prolongation ähnlich verhalten, was anhand der Varianz gemessen werden kann. Durch die Faktorenanalyse werden Module gruppiert, die bei einer Prolongation ähnliche Varianz aufweisen.

Zur Durchführung der Faktorenanalyse werden die Messwerte in einer Matrix M aufgetragen, wobei der Wert m_{ij} durch die Laufzeit des j -ten Moduls in der i -ten Messung gegeben ist. Diese Matrix wird zu einer Matrix Z normiert, indem $z_{ij} = \frac{m_{ij} - \bar{m}_i}{\sigma_i}$ gesetzt wird (wobei \bar{m}_i das arithmetische Mittel und σ_i die Standardabweichung der i -ten Spalte von M ist).

Aus der normierten Messungsmatrix Z wird nun die Korrelationsmatrix R berechnet:

$$R = \frac{1}{m-1} \cdot Z^t \cdot Z$$

Gesucht ist nun eine Darstellung von Z durch Faktoren P und Faktorladungen A – welche die Zugehörigkeit der Variablen zu den Faktoren beschreiben – mit

$$Z = P \cdot A^t.$$

Unter der Annahme der Existenz einer solchen Zerlegung nutzen wir die Herleitung

$$\begin{aligned} R &= \frac{1}{m-1} \cdot Z^t \cdot Z = \frac{1}{m-1} \cdot (P \cdot A^t)^t \cdot (P \cdot A^t) = \frac{1}{m-1} \cdot A \cdot P^t \cdot P \cdot A^t \\ &= A \cdot \underbrace{\left(\frac{1}{m-1} \cdot P^t \cdot P \right)}_{P^*} \cdot A^t. \end{aligned}$$

Sind die Faktoren wie gewünscht unkorreliert, ist P^* eine Einheitsmatrix. Damit ergibt sich das Fundamentaltheorem der Faktorenanalyse:

$$R = A \cdot P^* \cdot A^t$$

Ein simplifiziertes Beispiel soll das Vorgehen verdeutlichen: Wir nehmen die in Tabelle 1(a) aufgeführten Messungen von drei Modulen a , b und c an. Diese werden normiert und einer Faktorenanalyse unterzogen. Daraus ergeben sich die Faktorladungen in 1(b).

Um den Umfang nicht zu sprengen, wurde die Faktorenanalyse nur für unseren Bedarf ausreichend eingeführt. Für fundiertere Betrachtungen sei auf [Bac96] verwiesen.

4 Vorgehensweise

Durch die Messung eines prolongierten Systems erhält man eine Tabelle mit einer Vielzahl von Spalten. Eine Spalte repräsentiert die Laufzeit eines Moduls über alle Messungen

(a) Messdaten M				(b) Faktorenladung A			
	a	b	c		$PC1$	$PC2$	$PC3$
1	6	2	1	1	0.771	-0.637	0
2	3	4	2	2	-0.570	-0.689	0.447
3	3	4	2	3	-0.285	-0.344	-0.894
\bar{m}_i	4	3,33	1,67				
σ_i	1,73	1,15	0,58				

Tabelle 1: Beispiel zur Faktorenanalyse

hinweg. Nun interessiert es, wie die Spalten (Variablen), also die Laufzeiten der Module, zusammenhängen. Die gemessenen Daten werden bei der Faktorenanalyse in ihrer Dimension reduziert, so dass die zugrundeliegende Ursache leichter verstanden werden kann.

Es können zugrundeliegende Faktoren entdeckt und Zusammenhänge aufgezeigt werden, die nicht auf den ersten Blick ersichtlich sind. A priori können zumeist durch die Größe des Systems und durch die Wiederbenutzung von Code keine Hypothesen aufgestellt werden, wieso ein System langsam ist, Faktoren können (zunächst einmal) nicht benannt werden. Jedoch ist es (zumeist) möglich, bei einer graphischen Interpretation der Faktoren die Module zu identifizieren und somit das System besser zu verstehen.

Abbildung 1 zeigt den Ablaufprozess bei der Performanzanalyse mit Hilfe der Faktorenanalyse. Im folgenden werden die einzelnen Prozessschritte anhand einer Beispielanwendung näher erklärt.

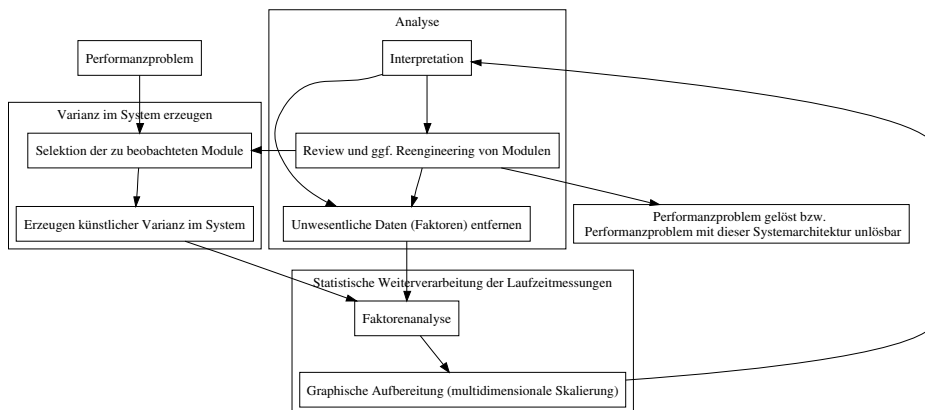


Abbildung 1: Ablauf einer Performanzanalyse mit Hilfe der Faktorenanalyse

4.1 Prolongieren mittels ASPECTJ

Für unser Anwendungsgebiet, der Performanzanalyse mit Hilfe von multivariaten Analysemethoden muss also Varianz im System erzeugt werden, damit die Kovarianz gemessen und bestimmt werden kann.

Diese Varianz kann vielfältig erzeugt und anschließend im System gemessen werden. Beispielsweise wurde in unseren Versuchen JAVA-Code mittels ASPECTJ instrumentiert, Varianz erzeugende Funktionalität könnte jedoch auch direkt im Code implementiert werden. Instrumentieren stellt aber die praktikablere Lösung dar, da das zu untersuchende System als Black-Box betrachtet werden kann und nicht direkt in den Code eingegriffen werden muss.

Das instrumentierte System wird nun mehrmals mit unterschiedlichen Prolongationen ausgeführt. Der instrumentierte Code variiert die Ausführungsdauer einzelner Module, das Gesamtsystem zeigt also Varianz in seiner Laufzeit. Das Tracing bzw. das Logging wurde in unseren Versuchen auch mittels eines Aspects implementiert und in das System eingewoben. Die Laufzeiten aller zu beobachtenden Module wird mitgemessen und zur Analyse gespeichert.

Es empfiehlt sich relevante Module mindestens dreimal mit verschiedenen zusätzlichen Intervallen zu prolongieren. Abhängig davon, ob das Betriebssystem Messungen verfälscht und wie deterministisch sich das zu messende System verhält, sollten mehr Messungen durchgeführt werden, bis sich repräsentative Daten ergeben.

Als Beispiel betrachten wir einen Webcrawler, der als Testanwendung für ein JAVA-basiertes Framework für nebenläufige Komponenten implementiert wurde. Der Aufbau des Webcrawler ist aus Abbildung 2 ersichtlich. Ausgehend von einer Startseite werden in verschiedenen Threads Seiten geladen, deren Links extrahiert und Worte gespeichert, wobei sowohl Links als auch Worte vorher normalisiert werden. Dieses Programm soll mittels der Performanzanalyse mit Hilfe einer Prolongation analysiert werden. Dabei wählen wir als Granularität die Komponentenebene, die über der Klassenebene angesiedelt ist. So wird

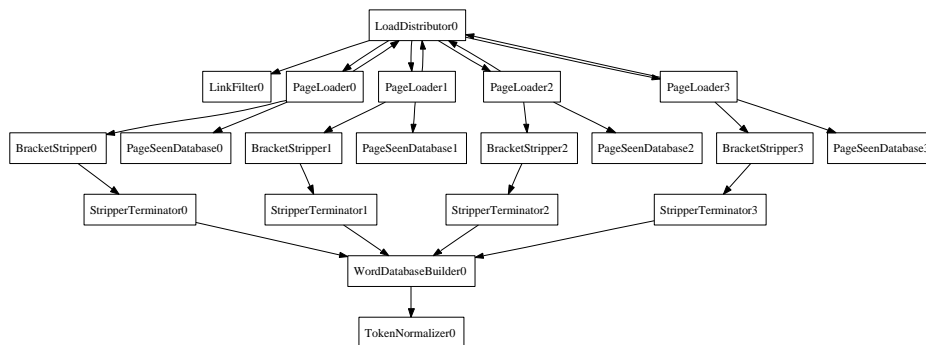


Abbildung 2: Komponenten des Webcrawlers. Die Pfeile einzelner Komponenten deuten auf abhängige Komponenten

z.B. ein Crawler-Thread, in dem Seiten geladen werden, als einzelnes Modul aufgefasst, obwohl es aus einer Vielzahl von Klassen besteht. Die Interaktionen auf Komponentenebene werden drei mal mit verschiedenen Werten prolongiert. Dann werden die Laufzeiten der einzelnen Komponenten gemessen.

4.2 Multivariate Analyse des gemessenen, variierten Systems

Die gemessenen Daten ergeben eine Matrix bzw. eine Tabelle. Diese Matrix sollte optimalerweise mit einem Statistikprogramm weiterverarbeitet werden. Bestehende Applikationen wie R[R D07], S-PLUS[Sta92] und SPSS[SPS07] stellen dafür eine Vielzahl von Optionen zur Verfügung und sind sehr komfortabel.

Die Messdaten werden in das Statistikprogramm eingelesen, eine Faktorenanalyse durchgeführt und die Faktoren multidimensional skaliert ausgegeben. Multidimensional heißt in diesem Kontext, dass jeder der gefundenen Faktoren eine Dimension ist und die gemessenen Module bezüglich ihrer Faktoren skaliert dargestellt werden. Ist beispielsweise die Laufzeit der main-Methode von zwei Faktoren abhängig, wird sie anteilig (Vektoraddition) auf diesen Faktoren im Diagramm dargestellt.

Die Daten, die zur Ausführungszeit des prolongierten Webcrawlers erzeugt werden, werden durch die eingewebten Aspekte in eine Datei gespeichert. Diese Datei öffnen wir mit einem Statistikprogramm, beispielsweise mit der Open Source Programmiersprache R und führen eine Hauptkomponentenanalyse (eine Faktorenanalyse [Bac96]) durch.

Aus Tabelle 4.2 wird folgender Zusammenhang klar: die Komponenten LOADDISTRIBUTOR und LINKFILER0 laden zusammen stark auf einen Faktor hoch, während die anderen gemessenen Komponenten für diesen Faktor (PC1) nicht ins Gewicht fallen. Diese beiden Komponenten bilden einen Faktor, den wir, unter Kenntnis der Funktionalität beider Komponenten, „Links extrahieren“ nennen.

Auf den zweiten Faktor (PC2) laden die Komponenten WORDDATABASEBUILDER0 und TOKENNORMALIZER0 positiv, die STRIPPERTERMINATOR-Komponenten negativ hoch. Wir nennen, unter Kenntnis der Funktionalität der beteiligten Komponenten, diesen Faktor „Wort-Datenbank“.

Analog lassen sich alle weiteren Faktoren interpretieren. Der Erklärungsgehalt der weiteren Faktoren nimmt jedoch immer weiter ab, d. h. diese Faktoren erklären das System nur noch minimal, können also vernachlässigt werden.

In Abbildung 3 sind die Faktoren multidimensional skaliert dargestellt. Will man die Gesamtlaufzeit verbessern, sollten Komponenten aus den ersten Faktoren optimiert werden, da diese die größten Varianzen erklären und somit den meisten Einfluß auf die Gesamtlaufzeit des Systems haben. Stellt eine Komponente ein Performanzproblem dar und ist selber nicht optimierbar bzw. ist die Ursache für die schlechte Performanz unbekannt, stellen die Komponenten im gleichen Faktor die Kandidaten für ein Reengineering zur Performanzverbesserung dar.

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8
BracketStripper0	0,00	0,00	0,00	0,00	0,00	-0,02	-0,02	-0,01
BracketStripper1	0,00	0,00	0,00	0,00	0,00	-0,02	-0,01	-0,01
BracketStripper2	0,00	-0,01	0,00	0,00	0,00	-0,03	-0,02	-0,01
BracketStripper3	0,00	-0,01	0,00	0,00	0,00	-0,02	-0,02	-0,02
LinkFilter0	-0,65	0,02	-0,01	0,01	0,02	0,30	-0,69	-0,06
LoadDistributor0	-0,75	-0,02	0,00	0,00	-0,01	-0,19	0,60	0,06
PageLoader0	0,08	0,01	0,02	0,00	0,02	-0,05	-0,15	-0,02
PageLoader1	0,03	0,00	0,01	0,02	0,00	-0,06	-0,14	-0,01
PageLoader2	-0,05	0,02	0,02	0,02	-0,01	-0,06	-0,05	0,00
PageLoader3	-0,04	0,01	0,02	0,01	-0,01	-0,05	-0,04	-0,01
PageSeenDatabase0	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
PageSeenDatabase1	0,00	0,00	0,00	0,00	0,00	-0,02	-0,02	-0,01
PageSeenDatabase2	0,00	0,00	0,00	0,00	0,00	-0,02	-0,02	-0,01
PageSeenDatabase3	0,00	0,00	0,00	0,00	0,00	-0,03	-0,02	-0,01
StripperTerminator0	0,02	-0,13	0,39	0,77	0,17	0,44	0,15	0,06
StripperTerminator1	0,02	-0,12	0,15	-0,25	-0,79	0,50	0,14	0,06
StripperTerminator2	0,03	-0,15	0,21	-0,57	0,59	0,46	0,16	0,07
StripperTerminator3	0,03	-0,13	-0,88	0,15	0,08	0,37	0,13	0,05
TokenNormalizer0	0,01	0,58	-0,01	0,00	0,03	0,22	0,16	-0,77
WordDatabaseBuilder0	0,01	0,77	-0,02	0,00	0,02	0,13	0,01	0,62

Tabelle 2: Faktorenladung des Webcrawler-Beispiels

4.3 Analyse der Performanz-Abhängigkeiten

Durch die multidimensionale Skalierung und mit Hilfe der gefundenen hypothetischen Faktoren fällt die Analyse eines Systems leichter. Es sind nun performanztechnische Abhängigkeiten sichtbar, die vorher eventuell nicht bekannt waren. Blockieren sich beispielsweise zwei Module durch eine gemeinsam benutzte Ressource, haben diese dieselbe Kovarianz und sind damit in einem Diagramm als korreliert erkennbar. Beide Module können nun einem Review unterzogen werden. Bei Optimierungspotential wird der Code der Module verändert, was zu einer besseren Laufzeit führt bzw. führen kann.

Eventuell kann durch die Vielzahl der Daten das Diagramm unübersichtlich sein. Besteht ein Faktor aus Modulen, deren Zusammenhang bekannt ist, können keine neuen Informationen aus dem Diagramm abgelesen werden. In diesen beiden Fällen sollten die Messdaten der unwesentlichen Module entfernt (Spalten) und die Faktorenanalyse erneut durchgeführt werden, um eine bessere Interpretation zu ermöglichen.

Sollen die Auswirkungen der Optimierung eines Moduls sichtbar gemacht werden oder ist die Performanz des Gesamtsystems immer noch nicht gut genug, muss eine neue Datenbasis mit dem neuen, variierten System geschaffen werden. Das in seiner Laufzeit variierte System muss erneut ausgeführt und die Laufzeiten der Module gemessen werden.

Die vorgestellte Methode der Performanzanalyse mit Hilfe der Faktorenanalyse erweitert bestehende Profilingmethoden, wobei es keinen Unterschied macht, ob das zu analysieren-

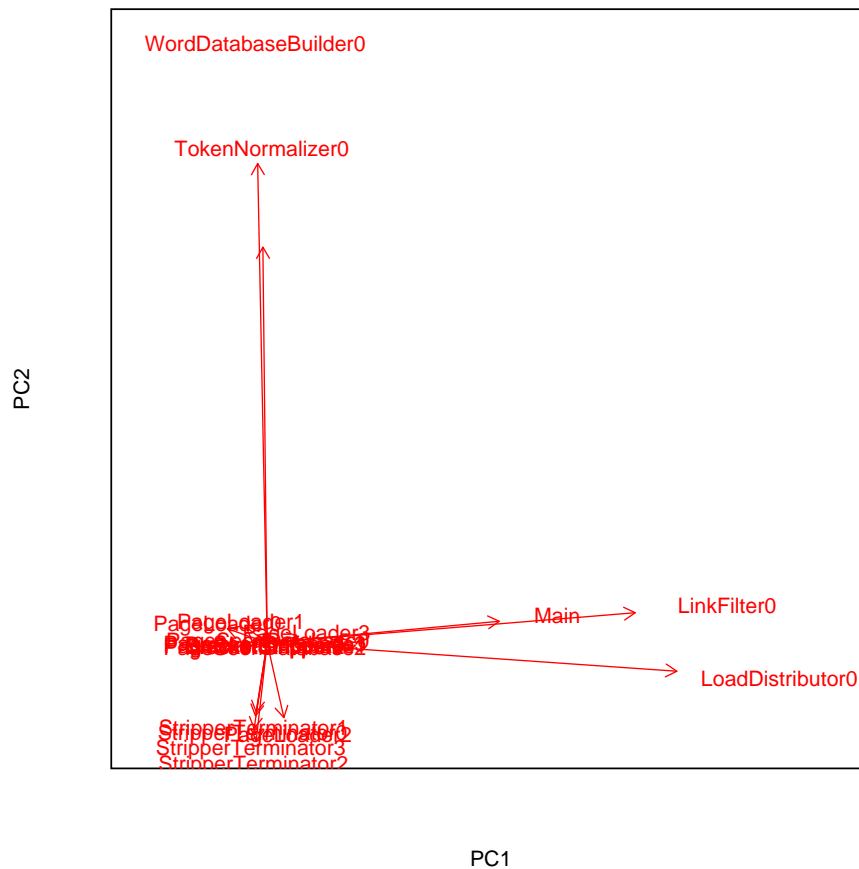


Abbildung 3: Biplot des Webcrawler-Beispiels

de System asynchron oder synchron ist. Herkömmliche Analysemethoden zeigen die nicht gut genug performenden Module isoliert auf. Mit Hilfe der künstlichen Varianz, die durch die Faktorenanalyse ausgewertet wird, werden dahinterstehende „Ursachen“ oder „Faktoren“ sichtbar. So ist es mit dieser Methode beispielsweise möglich, Performanzprobleme, die auf Grund von gemeinsam genutzten Ressourcen entstehen, zu entdecken.

Im Gegensatz dazu sind bei einem call tree Profiling die dahinter liegenden Ursachen bei der Analyse nicht leicht ersichtlich. Prinzipiell problematisch erscheint bei dem hier vorgestellten Ansatz jedoch die hohe Anzahl der Messläufe für die Varianzmessungen, gerade bei Systemen mit bestehenden Performanzproblemen. Wir schlagen hierfür eine Skalierung über die Granularität der zu variierenden Module vor, um erst bei hoher Granularität relevante Teile des Systems zu eruieren und unrelevante Teile von der Analyse auszuschließen. Konkret würde dies bedeuten, dass zuerst Komponenten, dann die Klassen aus einer Teilmenge dieser Komponenten und zum Schluß die Laufzeit der einzelnen Module prolongiert wird.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein neuer Ansatz zur Performanzanalyse großer, nebenläufiger Systeme vorgestellt. Die zentrale Neuerung dabei ist die Berücksichtigung von Phänomenen, die durch die Nebenläufigkeit entstehen und aus der isolierten Betrachtung des Verhaltens einzelner Module nicht ablesbar sind. Zudem ist der Ansatz nahezu beliebig skalierbar. Um die resultierende Datenmenge beherrschen zu können, wurde die Faktorenanalyse eingesetzt. Unsere Erfahrungen legen nahe, dass sich der Ansatz sowohl durch seine technische Einfachheit als auch durch sein neuartiges Erklärungspotential für bestehende, große Softwaresysteme anbietet.

Mit der Faktorenanalyse wurde eine Methode der multivariaten statistischen Analyse verwendet, um eine konzentriertere Erklärung des Systemverhaltens zu erhalten. Andere statistische Methoden, wie die Regressionsanalyse, könnten das gemessene Verhalten unter einem anderen Blickwinkel analysieren. Beispielsweise können die unterschiedlichen Einflüsse einzelner Faktoren oder Module gewichtet und bewertet werden.

Unsere Analyse ist aktuell nur durch technische Gegebenheiten in ihrer Granularität begrenzt. So ist es in der aktuellen Version von ASPECTJ nicht möglich, eine feinere Granularität als die Methoden zu wählen. Diese Einschränkung ist jedoch arbiträr, und mittels einer spezialisierten Instrumentierung könnte bis auf die Granularität einzelner Bytecodes prolongiert werden. Weiterhin ist es notwendig, Techniken zu erforschen, die eine Prolongation externer Module im Allgemeinen möglich macht. Die Implementierung entsprechender Tools, sowie die Anpassung der Granularität auf Artefakte des Codes (z.B. durch eine statische Analyse, die mögliche Probleme mit Nebenläufigkeit aufdeckt) versprechen eine lohnende Erweiterung unseres Ansatzes zu sein.

Literatur

- [Bac96] Backhaus, Klaus et al. Multivariate Analysemethoden. Springer, 1996.
- [BGdMT98] Gunter Bolch, Stefan Greiner, Hermann de Meer und Kishor S. Trivedi. Queuing networks and Markov chains: modeling and performance evaluation with computer science applications. Wiley-Interscience, New York, NY, USA, 1998.
- [HWPL04] Andreas Hennig, Rainer Wasgint, Boris Petrovic und Olkhovich Lev. Instant Performance Prototyping of EJB/J2EE Applications - A car rental example. In Proceedings des 5. Workshop des Arbeitskreis PEAK, Munich, 2004.
- [Joh02] Rod Johnson. Expert One-on-One J2EE Design and Development. Wrox Press Ltd., Birmingham, UK, 2002.
- [KHH⁺01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm und William Griswold. Getting started with ASPECTJ. Commun. ACM, 44(10):59–65, 2001.
- [LCH04] Seon-Ah Lee, Seung-Mo Cho und Sung-Kwan Heo. SAAT: Reverse Engineering for Performance Analysis. In Second International Workshop on Dynamic Analysis (WODA 2004) W10S Workshop - 26th International Conference on Software Engineering, Seiten 40–47, 2004.

- [PW05] Dorin Bogdan Petriu und Murray Woodside. Software performance models from system scenarios. *Perform. Eval.*, 61(1):65–89, 2005.
- [R D07] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2007. ISBN 3-900051-07-0.
- [Reu07] Lennart Reuther. Tools für Performance- und Lasttests. <http://www.l-ray.de/curriculum/amacont/tools.html>, 2007.
- [Sof07] Quest Software. Datasheet - JProbe Suite. www.quest.com/Quest_Site_Assets/PDF/Datasheet--JProbe_Suite_3.pdf, 2007.
- [Spe04] Charles Spearman. General intelligence objectively determined and measured. In *American Journal of Psychology* 15, Seiten 201–293. University of Illinois Press, 1904.
- [SPS07] SPSS. SPSS, Data Mining, Statistical Analysis Software, Predictive Analysis, Predictive Analytics, Decision Support Systems. {<http://www.spss.com/de/>}, 2007.
- [Sta92] StatSci, a Division of MathSoft, Inc., Seattle, WA, USA. *S-PLUS Programmer's Manual*, version 3.1. Auflage, Oktober 1992.
- [SW02] Connie U. Smith und Lloyd G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. 2002.
- [Woo00] C. Murray Woodside. Software Performance Evaluation by Models. In *Performance Evaluation: Origins and Directions*, Seiten 283–304, London, UK, 2000. Springer-Verlag.