Analyse von IT-Anwendungen mittels Zeitvariation

Florian Mangold

München 2010

Analyse von IT-Anwendungen mittels Zeitvariation

Florian Mangold

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig–Maximilians–Universität
München
zur Erlangung des Grades
Doctor rerum naturalium (Dr. rer. nat.)

vorgelegt von
Florian Mangold
aus Landsberg am Lech

München, den 21. Oktober 2010

Erstgutachter: Prof. Dr. Martin Wirsing
Zweitgutachter: Prof. Dr. Ruth Breu
Tag der mündlichen Prüfung: 18. November 2010

Für Lena

(✻ 31.05.1983 – †20.05.2003)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Zusammenfassung

Performanzprobleme treten in der Praxis von IT-Anwendungen häufig auf, trotz steigender Hardwareleistung und verschiedenster Ansätze zur Entwicklung performanter Software im Softwarelebenszyklus. Modellbasierte Performanzanalysen ermöglichen auf Basis von Entwurfsartefakten eine Prävention von Performanzproblemen. Bei bestehenden oder teilweise implementierten IT-Anwendungen wird versucht, durch Hardwareskalierung oder Optimierung des Codes Performanzprobleme zu beheben. Beide Ansätze haben Nachteile: modellbasierte Ansätze werden durch die benötigte hohe Expertise nicht generell genutzt, die nachträgliche Optimierung ist ein unsystematischer und unkoordinierter Prozess. Diese Dissertation schlägt einen neuen Ansatz zur Performanzanalyse für eine nachfolgende Optimierung vor. Mittels eines Experiments werden Performanzwechselwirkungen in der IT-Anwendung identifiziert. Basis des Experiments, das Analyseinstrumentarium, ist eine zielgerichtete, zeitliche Variation von Start-, Endzeitpunkt oder Laufzeitdauer von Abläufen der IT-Anwendung. Diese Herangehensweise ist automatisierbar und kann strukturiert und ohne hohen Lernaufwand im Softwareentwicklungsprozess angewandt werden. Mittels der Turingmaschine wird bewiesen, dass durch die zeitliche Variation des Analyseinstrumentariums die Korrektheit von sequentiellen Berechnung beibehalten wird. Dies wird auf nebenläufige Systeme mittels der parallelen Registermaschine erweitert und diskutiert. Mit diesem praxisnahen Maschinenmodell wird dargelegt, dass die entdeckten Wirkzusammenhänge des Analyseinstrumentariums Optimierungskandidaten identifizieren. Eine spezielle Experimentierumgebung, in der die Abläufe eines Systems, bestehend aus Software und Hardware, programmierbar variiert werden können, wird mittels einer Virtualisierungslösung realisiert. Techniken zur Nutzung des Analyseinstrumentariums durch eine Instrumentierung werden angegeben. Eine Methode zur Ermittlung von Mindestanforderungen von IT-Anwendungen an die Hardware wird präsentiert und mittels der Experimentierumgebung anhand von zwei Szenarios und dem Android Betriebssystem exemplifiziert. Verschiedene Verfahren, um aus den Beobachtungen des Experiments die Optimierungskandidaten des Systems zu eruieren, werden vorgestellt, klassifiziert und evaluiert. Die Identifikation von Optimierungskandidaten und -potenzial wird an Illustrationsszenarios und mehreren großen IT-Anwendungen mittels dieser Methoden praktisch demonstriert. Als konsequente Erweiterung wird auf Basis des Analyseinstrumentariums eine Testmethode zum Validieren eines Systems gegenüber nicht deterministisch reproduzierbaren Fehlern, die auf Grund mangelnder Synchronisationsmechanismen (z.B. Races) oder zeitlicher Abläufe entstehen (z.B. Heisenbugs, alterungsbedingte Fehler), angegeben.

Abstract

Performance problems are very common in IT-Application, even though hardware performance is consistently increasing and there are several different software performance engineering methodologies during the software life cycle. The early model based performance predictions are offering a prevention of performance problems based on software engineering artifacts. Existing or partially implemented IT-Applications are optimized with hardware scaling or code tuning. There are disadvantages with both approaches: the model based performance predictions are not generally used due to the needed high expertise, the ex post optimization is an unsystematic and unstructured process. This thesis proposes a novel approach to a performance analysis for a subsequent optimization of the IT-Application. Via an experiment in the IT-Application performance interdependencies are identified. The core of the analysis is a specific variation of start-, end time or runtime of events or processes in the IT-Application. This approach is automatic and can easily be used in a structured way in the software development process. With a Turingmachine the correctness of this experimental approach was proved. With these temporal variations the correctness of a sequential calculation is held. This is extended and discussed on concurrent systems with a parallel Registermachine. With this very practical machine model the effect of the experiment and the subsequent identification of optimization potential and candidates are demonstrated. A special experimental environment to vary temporal processes and events of the hardware and the software of a system was developed with a virtual machine. Techniques for this experimental approach via instrumenting are stated. A method to determine minimum hardware requirements with this experimental approach is presented and exemplified with two scenarios based on the Android Framework. Different techniques to determine candidates and potential for an optimization are presented, classified and evaluated. The process to analyze and identify optimization candidates and potential is demonstrated on scenarios for illustration purposes and real IT-Applications. As a consistent extension a test methodology enabling a test of non-deterministic reproducible errors is given. Such non-deterministic reproducible errors are faults in the system caused by insufficient synchronization mechanisms (for example Races or Heisenbugs) or aging-related faults.

Danksagung

Großer Dank gilt meinem Doktorvater Herrn Professor Dr. Martin Wirsing. Nur durch seine geduldige Führung und Motivation und seine stets konstruktiven Kritik hat diese Arbeit heute diese Form. Durch ihn kamen die Inspiration und der Mut, die Mechanismen und Effekte tiefgreifend zu verstehen und zu erklären. Unter seiner Anleitung zu den theoretischen, konzeptuellen und methodischen Aspekten wissenschaftlicher Forschung konnten hoffentlich beständige Ergebnisse geschafft und grundlegende Erkenntnisse erzielt werden.
Großer Dank gilt Frau Professor Dr. Ruth Breu, die ohne Zögern sofort das Zweitgutachten übernommen hat. Vielen Dank für Ihre wertvolle Zeit und die Diskussion aller umfangreichen Facetten dieser Arbeit.
Großer Dank gilt Professor Mirco Tribastone und Professor Dr. Alexander Knapp für die fachlichen Diskussionen, die Unterstützung und Ermutigungen.
Ein spezieller Dank gilt Herrn Rainer Wasgint für die warme, kollegiale, fleißige und motivierende Arbeitsatmosphäre, die er mit seiner Persönlichkeit schafft, und seiner Unterstützung in jeglichen Belangen. Großer Dank gebührt auch meinem Siemens internen Betreuer Herrn Dr. Harald Rölle. Ich habe viel von ihm gelernt, unter andrem wichtige konzeptuelle und methodische Vorgehensweisen. Wertvoll waren die Unterstützung und die fachlichen Diskussionen.
Ein weiterer spezieller Dank geht an den PST Lehrstuhl und das PST Team. Vielen Dank für die herzliche und wissenschaftliche Atmosphäre. Hier gilt mein besonderer Dank meinem Freund Herrn Dr. Moritz Hammer, insbesondere für seine Gedankenanstöße und die tolle Zusammenarbeit.
Für wertvolle Diskussionen danke ich: Herrn Dr. Bernhard Kempter, Herrn Dr. Michael Pönitsch, Herrn Dr. Stephan Janisch und Herrn Gefei Zhang.
Schließlich gilt mein letzter, aber nicht mein geringster, Dank meiner Verlobten Frau Sandra Laumer für ihr Lachen, ihre Liebe und ihren Glauben an uns.

München, den 21. November 2010

Florian Mangold

Kapitel 1

Einführung

„Wenn du es eilig hast, geh langsam."

Konfuzius

Übersicht des Kapitels

Die folgende Einführung motiviert diese Arbeit und zeigt den Bedarf für eine weitere Methodologie zur Performanzanalyse. In Abschnitt 1.1 wird anhand des Status Quo gezeigt, dass die bestehenden Ansätze teilweise unzureichend sind. Abschnitt 1.2 skizziert den verwendeten Lösungsansatz, definiert den Fokus dieser Arbeit und benennt die sich daraus ergebenden Teilfragestellungen. Abschnitt 1.2.2 zeigt das Vorgehensmodell und listet die behandelten Aspekte der einzelnen Kapitel dieser Arbeit auf.

1.1 Motivation

essentielles Qualitätsattribut Performanz

Ausreichende Performanz[1] stellt ein erfolgskritisches Qualitätsmerkmal von IT-Anwendungen dar [CPL09]. Werden diese Anforderungen nicht erfüllt kann dies zu schlechter Qualität [Fuc07, S. 431] sogar hin bis zur Unbenutzbarkeit, also zur Infunktionalität der Anwendung führen [Sha00, S. 80]. Trotz exponentieller Leistungssteigerung der Hardware [Jos03, S. 37][Sch97] verursacht unzureichende Performanz jedes

[1] Wieso in dieser Arbeit der deutsche Terminus „Performanz" und nicht sein englisches Pendant „Performance" verwendet wird, wird in Abschnitt 1 auf Seite 16 erörtert.

Kosten und Folgeschäden

Jahr extrem hohe Kosten und weitere Folgeschäden [WS03, S. 1], kritisch sind die meist schnell verfügbaren, performanteren und damit qualitativ bessere Alternativen der Konkurrenz [WS03, S. 1][CP09]. Ein im Jahre 2006 durchgeführtes Gutachten bestätigt, dass Performanzprobleme bei ausgelieferten IT-Anwendungen eher im Regelfall und nicht als Ausnahme erwartet werden [Com06, S. 3, Frage 5].

Ansätze

Wegen der enormen Signifikanz gibt es daher unterschiedliche Ansätze zur Realisierung ausreichend performanter IT-Anwendungen. Vor einer Behebung von Performanzproblemen oder einer Optimierung der Performanz erfolgt zu meist eine vorangehende

Performanzanalyse

Analyse – die Performanzanalyse – um die Ursachen zu identifizieren. Der Betrachtungsfokus der Performanzanalyse liegt zu meist einseitig auf der Software der IT-

System

Anwendungen nicht jedoch auf dem gesamten System, also der Summe ausgeführter Software und der zugrundeliegenden, angesteuerten Hardware.

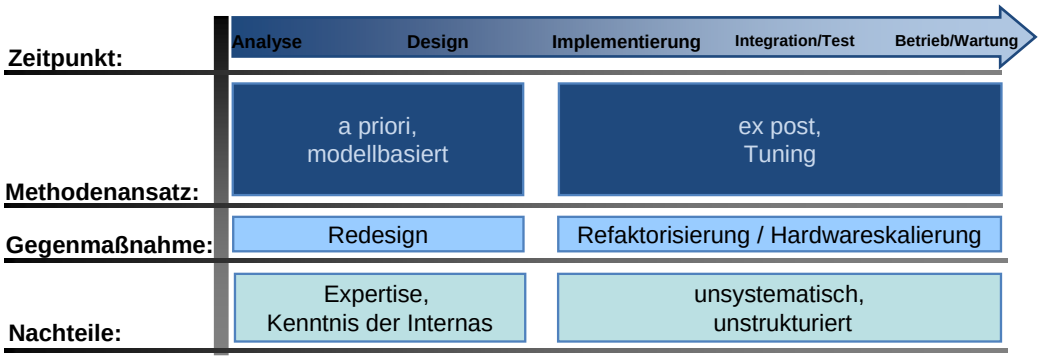


Abbildung 1.1: Gegenmaßnahmen und Nachteile der Methodenansätze zur Realisierung ausreichender Performanz einer IT-Anwendung im Softwarelebenszklus.

1.1.1 *A priori Ansätze*

Prävention von Performanzproblemen

In den frühen Phasen des Softwareentwicklungsprozesses gibt es angewandte Methodensammlungen, die Performanzprobleme präventiv verhindern sollen, da diese oft auf Grund von Designfehlern entstehen [SW02]. Hierbei wird oft nicht ein neues Modellierungskonzept verwandt, sondern bestehende Modellierungskonzepte (z. B. UML 2.2 [Gro09]) werden um performanz-relevante Gesichtspunkte erweitert, z. B. annotiert (z. B. mittels des *UML Profile for Schedulability, Performance and Time* (*SPT*) [Theo5]). Subsequent kann mit mathematischen Mitteln eine (Performanz-)Analyse durchgeführt werden und durch diese Vorhersage das System noch vor der Implementierung durch ein Redesign iterativ optimiert werden.

1.1.2 *Ex post Ansätze*

reaktive, ex post Optimierungen

Viele Softwareentwickler haben eine „*fix-it-later*“ Einstellung [SW02], der Fokus wird auf die Implementierung der funktionalen Anforderungen gelegt. Nach der Implementierung wird das System getestet und notfalls optimiert, man spricht hier von einem „*Tuning*“ [Shi02; SW02].
Hierbei gibt es zwei Strategien für das Tuning:

- **Leistungsfähigere Hardware** soll die unzureichende Performanz kompensieren.
- **Profiling** soll Optimierungskandidaten im Code identifizieren, die nachfolgend refaktorisiert werden können.

1.1.3 *Probleme der Ansätze*

unzureichend

Der Status Quo zur Behebung unzureichender Performanz in einer IT-Anwendung, insbesondere der essentiellen vorgegliederten Performanzanalyse, ist unzureichend. Die bestehenden Ansätze, präventive sowie reaktive Methoden, haben einige inhärente Nachteile.

1.1.3.1 Probleme der a priori Ansätze

unpopulär

Trotz aller elaborierten modellbasierten Techniken zur Sicherung und Vorhersage der Performanz (vgl. z. B. [Bal+04]) werden diese nur in einem geringen Bruchteil aller Projekte angewandt (das erwähnte Gutachten bestätigt, dass nur 14% aller Organisationen konsistent eine formale Methode benutzen [Como6, S. 2, Frage 1]), was insbesondere durch die benötige hohe Expertise bzw. den hohen Lernaufwand, fehlender oder schlechter Methoden- oder Werkzeugunterstützung im Softwareentwicklungsprozess und einer notwendigen a priori Kenntnis des Systems und seiner Nutzung zu erklären ist.

benötigte Interna ausgeblendet

Kapselung, Wiederverwendung und die Integration von bestehenden Komponenten und Code ist extrem effizient und wird propagiert [Boe07, S. 179ff][Zwe+95]. Parallel hat sich Abstraktion (auf funktionale Anforderungen) sehr nützlich im Entwicklungsprozess erwiesen. Ein freier Austausch von Diensten und Komponenten, auch während der Systemausführung, wird angestrebt [Ham09]. Dadurch bedingt sind die a priori Ansätze schwer durchführbar, da die Interaktionen der Komponenten zu meist unbekannt sind und nicht eruiert werden können. Des Weiteren werden Systeme insgesamt komplexer, sollen integriert werden und verändern sich durch Wartung der Software. Benötigte Interna für einen modellbasierten Ansatz werden ausgeblendet, verändern sich oder sind teilweise nicht zugänglich (Softwarebausteine von Drittanbietern, Altlastsysteme etc.) was eine a priori Performanzanalyse ausschließt oder drastisch erschwert [Kro07].

1.1.3.2 Probleme der ex post Ansätze

Kompensation mit leistungsfähiger Hardware

Bei unzureichender Performanz kann versucht werden, mittels einer Skalierung der Hardware diese zu kompensieren („*kill it with iron*-Ansatz“) [Wei+02, S. 22]. Dadurch wird gehofft den Effekt zu beseitigen ohne langwierig die Ursache identifizieren zu müssen. Jedoch können Bottlenecks verhindern, dass die Performanz linear mit der Hardware skaliert [SW02], oder es tritt der sogenannte Rebound-Effekt auf, hierbei kann sich die Performanz der IT-Anwendung sogar verschlechtern [RS99, S. 274]. In der Regel ist dieses Tuning durch Skalierung der Hardware ein unkoordinierter Prozess [SB10]. Verbesserungen um den Faktor zwei bis drei können erreicht werden, gefordert sind zu meist Verbesserungen um den Faktor zehn und mehr [RS99, S. 265].

Profiling

Das Profiling zur Identifikation von Optimierungskandidaten scheint zur Performanzanalyse nicht ausreichend. Performante Softwarebausteine sind notwendig aber nicht hinreichend für eine gute Gesamtperformanz, Wechselbeziehungen mit anderen Komponenten des Systems können beim Profiling nur schwer betrachtet werden. Wird dem Profiling (zum Zwecke einer Optimierung im Softwareentwicklungsprozess) jedoch eine gewisse kritische Relevanz zugemessen, profitieren die Anwendungen in über 80% der Fällen in ihrer Performanz [Com06, S. 3, Frage 7a, Frage 7b].

mühsam, langwierig

Beim Profiling kann oft nicht auf die unterschiedlichen Zielumgebungen eingegangen werden, der Analysefokus liegt auf dem Code und nicht auf der unterliegenden Hardware. Eine gewisse Kenntnis des Systems um die zu profilenden Softwarebausteine zu identifizieren muss vorhanden sein oder mühsam erworben werden. Für die Selektion der Optimierungskandidaten ist eine Expertise nötig und der Code muss offen vorliegen. Die sich, notwendigerweise, immer mehr etablierenden parallelen Systeme sind sehr komplex und schwer zu überblicken, was ein Profiling und mit nachfolgender Optimierung drastisch erschwert [KL94; KK00; XKJ10; TPB10].

Tuning unstrukturierter Prozess

Die Wirkzusammenhänge im Kontext der ex post Optimierungen (durch Profiling und Refaktorisierung potenzieller Optimierungskandidaten und der Skalierung der unterliegenden Hardware) sind bislang nicht ausreichend untersucht worden. Insgesamt ist das Tuning ein unkoordinierter und unsystematischer Prozess. Mehr willkürlich als methodisch und planmäßig werden Komponenten des Systems (Hardware oder Software) so lange verbessert bis das System eine ausreichende Performanz zeigt.

1.2 *Ziel und Fokus der Arbeit*

❑ **Das Ziel dieser Arbeit** ist es, **Werkzeuge und Methoden für eine Performanzanalyse** für unterschiedliche Granularitäten eines Systems nach oder während der Implementierung zu entwickeln. Der Prozess der Optimierung der Performanz eines Systems soll dadurch koordiniert und systematisch erfolgen können und nicht auf einer „Schätzung“ vermutlicher Optimierungskandidaten basieren. Ziel dieser Arbeit

In dieser Arbeit wird zwischen dem **Analyseinstrumentarium** und der eigentlichen **Analyse** unterschieden. Das Analyseinstrumentarium soll als ein Werkzeug, ein Zwischenschritt, zur eigentlichen Analyse — dem Ziel dieser Arbeit — verstanden werden. **Die Analyse** ist das Ergebnis eines Experiments, **das Analyseinstrumentarium** ist die Basis auf der die Versuchsbedingungen für das Experiment variiert werden. Analyseinstrumentarium Analyse

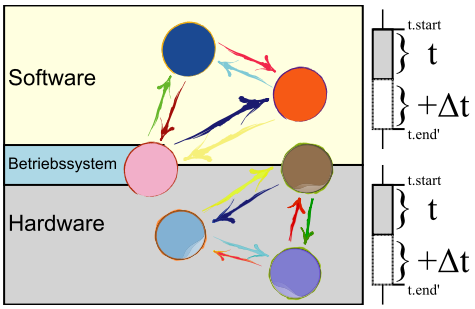


Abbildung 1.2: Das Analyseinstrumentarium als zeitliche Variation (Zeitpunkte und/oder Zeitdauer) an den Hard- oder Softwarekomponenten des Systems.

❑ **Das Analyseinstrumentarium** (Analysewerkzeug) ist eine zielgerichtete Variation von Zeitpunkten und/oder von Zeitdauern in den Abläufen eines Systems. Diese Vorgehensweise ist ein grundsätzliches Werkzeug, das in dieser Arbeit zur Performanzanalyse eingesetzt wird, vielgestaltige weitere Einsatzmöglichkeiten sind denkbar. Lösungsinstrument

Beispiel 1.2.1 Analyseinstrumentarium
Die analysierte Perspektive ist beispielsweise die Software einer IT-Anwendung. Zielgerichtet werden nun an bestimmten, ausgeführten Softwareelementen Zeitpunkte oder die Zeitdauern variiert. Möglich sind beispielsweise: Lösungsinstrument Analyseinstrumentarium

❑ **Der Startzeitpunkt** der Ausführung eines Softwaremoduls wird verändert.
❑ **Der Endzeitpunkt** der Ausführung des Softwaremoduls wird verändert.
❑ **Die Laufzeitdauer** der Ausführung wird verändert.

Eine Variation eines Zeitpunktes entspricht zu meist auch einer Variation der Zeitdauer und umgekehrt.

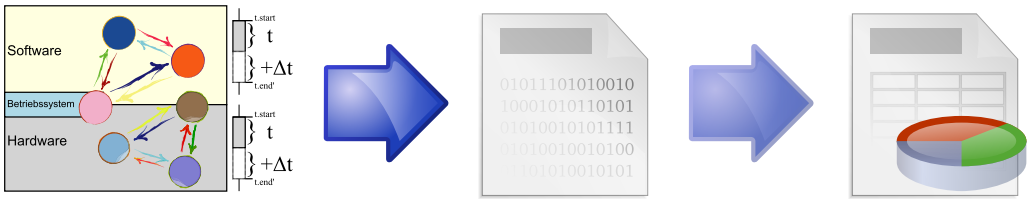


Abbildung 1.3: Empirischer Ansatz: Experiment mit dem Analyseinstrumentarium und nachfolgende Analyse der Messdaten.

Lösungsansatz

❑ **Die Analyse** erfolgt auf Basis des Analyseinstrumentariums. In einem Experiment werden die Komponenten eines ausgeführten Systems (Hardware- oder Softwarekomponenten) in ihrer Laufzeit gemessen und variiert. Durch diesen empirischen Ansatz müssen keine wesentlichen Eigenschaften abstrahiert werden und Seiteneffekte mit anderen Systembestandteilen (beispielsweise parallel laufenden Programmen oder dem Betriebssystem) werden in die Analyse integriert. Die Messdaten werden mit mathematischen Methoden weiterverarbeitet, um so Wirkzusammenhänge und Optimierungskandidaten zu identifizieren.

Beispiel 1.2.2 Analyse
In der IT-Anwendung wurden sukzessive verschiedene Module zeitlich variiert und ausgeführt, das Analyseinstrumentarium (siehe Beispiel 1.2.1) wird wiederholt angewandt. Die Ausführungszeiten der einzelnen Module werden protokolliert und subsequent mit einem Hilfsmittel wie einem Statistikprogramm (wie R oder SPSS) ausgewertet. Diese Auswertung kann beispielsweise mit einer „Faktorenanalyse“ erfolgen, die zugrundeliegende Wirkbeziehungen werden in Faktoren gruppiert. Somit erhält man Einblick in die Wirkzusammenhänge eines komplexen Systems und kann Optimierungskandidaten identifizieren.

Test auf nicht deterministisch reproduzierbare Fehler

❑ **Ein zusätzliches Ergebnis** dieser Arbeit ist eine Testmethodologie auf nicht deterministisch reproduzierbare Fehler eines Systems. Mit dem Analyseinstrumentarium werden zeitliche Abläufe beeinflusst, hierbei können (bislang unbekannte) Fehler, bedingt durch mangelnde Synchronisationsmechanismen oder basierend auf bestimmten zeitlichen Effekten, gefunden werden. Im Normalfall (während der Entwicklung oder dem Testen) werden diese nicht entdeckt, können sich aber auf unterschiedlichen Zielplattformen oder unter gewissen Umständen ereignen.

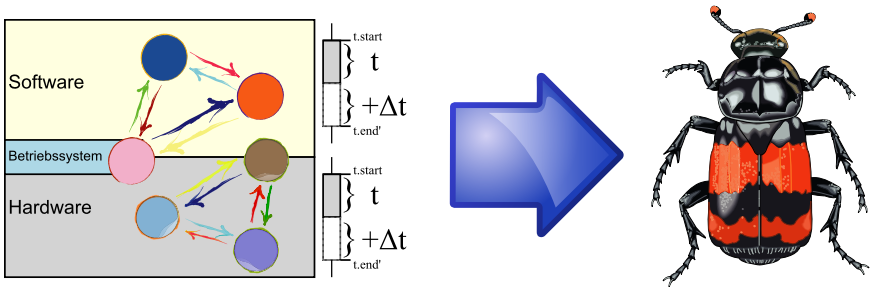


Abbildung 1.4: Das Analyseinstrumentarium kann als Technik zum Test von Systemen eingesetzt werden.

Beispiel 1.2.3 Nicht deterministisch reproduzierbare Fehler

Treten Fehler in einem Softwareprodukt auf, kann dies zu enormen Kosten und im Extremfall zu einer eventuellen Gefährdung der menschlichen Gesundheit und des menschlichen Lebens führen.

Testen

Testen ist deshalb ein integraler Bestandteil der Systementwicklung. Hierbei sollen Fehler reproduziert bzw. provoziert werden, die in einem nachfolgenden, iterativen Prozess beseitigt werden. Manche Fehler werden jedoch beim Testen nicht entdeckt, sie sind schwer zu reproduzieren, da ihr Auftreten an bestimmte zeitliche Gegebenheiten oder Abläufe gebunden ist.

Das Analyseinstrumentarium setzt aber genau hier an: für eine Performanzanalyse werden zielgerichtet Abläufe zeitlich variiert, wobei funktionale Eigenschaften beibehalten werden sollen. Werden diese nicht beibehalten, kann sich dieses nicht intendierte Verhalten auf anderen Zielplattformen oder unter gewissen Umständen wiederholen.

Berühmte und berüchtigte Beispiele für diese nicht deterministisch reproduzierbare Fehler, welche auf bestimmte zeitlichen Abläufen basieren und nicht im Testprozess entdeckt wurden, sind:

Northeast Blackout

❏ der Stromausfall in Nordamerika und Teile Kanadas im Jahr 2003 (*Northeast Blackout*), verursacht durch eine *Race Condition* [Neu06; Bos+08][PVH10, S. 245],

Patriot Missile

❏ ein Fehler bei einem *Patriot System*, basierend auf einer Alterung des Systems (Akkumulation von Rundungsfehlern), welcher zu 28 Toten und 97 Verletzten führte [LMT10, S. 125][Bal+10; Off92] (siehe Szenario 8 und Exkurs 10.4.1, Seite 271) und

Therac-25

❏ die tödliche Strahlendosis der *Therac-25* für Patienten bei einer Strahlentherapie [LT93, S. 27] (siehe Exkurs 10.3.1, Seite 267).

Tests als Konsequenz

Im Rahmen dieser Arbeit muss die Korrektheit des Analyseinstrumentariums gezeigt werden. Der Anwendungsfall zum Test auf nichtdeterministisch reproduzierbare Fehler bzw. Synchronisationsfehler als zusätzliches Ergebnis ergibt sich als Konsequenz direkt dadurch.

1.2.1 Teil- und Forschungsfragen

Teil- und Forschungsfragen

Für das Ziel (eine systematische ex post Performanzanalyse) mit diesem Lösungsansatz (Experimente mit zielgerichteter Änderung von Zeitpunkten und -dauer) ergeben sich folgende Teilfragestellungen und Forschungsfragen zur Beantwortung:

Betrachtungsfokus

❑ α) **Wo liegt der Betrachtungsfokus für diese Arbeit?**
Was ist Performanz? Was ist eine Performanzanalyse? Wie werden diese in dieser Arbeit und in der Wissenschaft definiert? Was ist der in dieser Arbeit benutzte Systembegriff? Aus was besteht ein System? Welche Termini werden zur Darstellung in dieser Arbeit benötigt? Was sind „nicht deterministisch reproduzierbare Fehler“? Was ist ein Softwaretest? An welchen Szenarios kann die in dieser Arbeit entwickelte Technik praktisch nachvollzogen und demonstriert werden?

bestehende Ansätze

❑ β) **Ist die Vorgehensweise (zum Stand der Technik und Wissenschaft) neu?**
Ist das Ziel dieser Arbeit (eine Performanzanalyse durch Variieren von Zeitpunkten und -dauern in Abläufen) innovativ? Gibt es bestehende Ansätze, die ähnlich sind? Welche Vorgehensweisen haben die etablierten Methoden zur Performanzanalyse? Können diese Ansätze, mit ihrer großen Spannbreite, kategorisiert und klassifiziert werden? Sind die bestehenden Ansätze zur Performanzanalyse im industriellen Umfeld der Systementwicklung etabliert, oder gibt es Bedarf an einer weiteren Methodologie zur Performanzanalyse.

experimenteller Ansatz

❑ γ) **Welche Vorteile ergeben sich aus dem experimentellen Ansatz und wie sieht das Analyseinstrumentarium aus?**
Welche Nachteile haben die bestehenden Methoden aus Wissenschaft und Industrie? Welche Vorteile ergeben sich aus dem hier erstellten Lösungsansatz zum Stand der Wissenschaft und Technik? Wie wird eine zeitliche Variation in einem System möglich? Welche zeitlichen Variationen sind möglich? Kann eine Verkürzung der Abläufe in einem Experiment realisiert oder müssen diese simuliert werden?

Validität

❑ δ) **Wird durch das Analyseinstrumentarium das Ergebnis einer Berechnung nicht verändert?**
Die Validität des Analyseinstrumentariums in einem Experiment muss formal verifiziert werden. Ist das Analyseinstrumentarium für eine Performanzanalyse durch ein Experiment, und demnach dem Ziel dieser Arbeit, valide (zur Validität vgl. [Alb09, S. 27])? Kann durch eine zeitliche Variation von Abläufen das Ergebnis einer (beliebigen) Berechnungen verändert werden? Mit welchen Mitteln der Informatik kann dies gezeigt werden? Was passiert, wenn diese Berechnung nicht atomar ist, sondern aus sequentiellen Teilberechnungen besteht? Verändert das Lösungsinstrument bzw. Analyseinstrumentarium das Ergebnis einer solchen Berechnung? Kann bei beliebigen sequentiellen Teilberechnungen an beliebigen Stellen zeitlich variiert werden?

❑ ϵ) **Wie sind die Wirkzusammenhänge des Analyseinstrumentariums?** Wirkzusammenhänge
Werden mit einer Variation der zeitlichen Abläufe einer Komponente andere Komponenten beeinflusst? Wenn dies der Fall ist, wie werden diese beeinflusst? Können durch diese Beeinflussung oder nachfolgenden Reaktionen Zusammenhänge erschlossen werden? Auf welchen Effekten basiert eine Reaktion anderer Komponenten? Wird die Laufzeit eines Softwaremodul variiert: reagieren andere Softwaremodule, z. B. mit einer Laufzeitvariation? Gibt es einen Zusammenhang zwischen dieser Reaktion und Optimierungskandidaten bzw. -potenzial bei anderen Komponenten des Systems?

❑ ζ) **Ist das Lösungsinstrument zielführend? Wird Optimierungspotenzial im System entdeckt?** Optimierungspotenzial
Sind diese Wirkzusammenhänge ein Indikator für Optimierungspotenzial bzw. Optimierungskandidaten? Kann so durch ein nachfolgendes Optimieren dieser Kandidaten eine bessere Performanz erreicht werden? Kann das Analyseinstrumentarium die Auswirkungen einer Verbesserung einzelner Komponenten identifizieren?

❑ η) **Kann die Variation von Ressourceneigenschaften (mittels des Analyseinstrumentariums) praktisch durchgeführt werden?** Experimentierumgebung
Um ein praktikables und effizientes Analyseinstrumentarium im Softwareentwicklungsprozess zur Verfügung zu haben, muss eine Möglichkeit geschaffen werden, effizient und zielgerichtet Abläufe einzelner Komponenten zeitlich zu variieren. Kann eine *Experimentierumgebung* konstruiert werden, mit der dieses möglich wird? Können einem Analysten Werkzeuge zur Realisation des Analyseinstrumentariums ohne eine Experimentierumgebung an die Hand gegeben werden? In welchen Komponenten bzw. Ebenen des Systems ist dieses Variieren realisierbar und zielgerichtet? Wie können die Daten des Experiments erhoben werden? Welche Möglichkeit zur Erhebung der empirischen Daten, nach dem Einsatz Analysinstrumentariums, gibt es? Eignet sich ein Logging der Methodenaufrufe? Welche Mechanismen können implementiert werden, wo liegen die Vor- und Nachteile dieser Mechanismen? Datenerhebung

❑ θ) **Wie kann die hier entwickelte Methode zur Bestimmung von Mindestanforderungen eingesetzt werden?** Hardware-Rahmenbedingungen
IT-Anforderungen haben spezielle Anforderungen an bestimmte Hardwarerahmenbedingungen. Lassen sich mit der in dieser Arbeit entwickelten Methode bzw. Experimentierumgebungen Aussagen über diese Mindestanforderungen ermitteln? Wie wäre beispielsweise der Effekt einer leistungsfähigeren, aber teueren CPU? Ist das System noch funktional bzw. ausreichend performant, wenn eine teure, aber schnelle magnetische Festplatte durch eine billigere Flash-Disc ausgetauscht wird? Lassen sich Rahmenbedingungen für einzelne Softwarekomponenten empirisch ermitteln, so dass gewünschte funktionale und nicht-funktionale Eigenschaften erhalten bleiben? Mit herkömmlichen Methoden ist dieses Testen praktisch nur sehr schwer realisierbar. Kann eine Experimentierumgebung realisiert werden, mit der solche Tests möglich werden, eventuell sogar

automatisierbar? Welche Vorteile ergeben sich daraus für den Softwareentwicklungsprozess, wie kann man Softwareentwickler unterstützen um systematisch die nichtfunktionalen Anforderungen einzuhalten? Können die hier vorgestellten Techniken in einen Testprozess einfließen?

Analyse ❑ ι) **Welche Auswertemöglichkeiten (Analysemethoden) können genutzt werden?** Wie lassen sich die Daten analysieren? Wie kann aus den erhobenen Daten eine Performanzanalyse durchgeführt werden? Welche Methoden zur Analyse der erhobenen Daten des Experiments gibt es? Diese Daten, beispielsweise Laufzeitdaten bzw. Start- oder Endzeitpunkt, enthalten Informationen aus einer Vielzahl von Variationen mittels des Analyseinstrumentariums. Können multivariate Analysetechniken, also Anleihen aus der Statistik, benutzt werden? In [Man07] wurde die Faktorenanalyse zur Gruppierung dahinterliegender Zusammenhänge der Module genutzt, gibt es ähnliche oder besser geeignete Verfahren? Welche Analysetechniken eignen sich grundsätzlich? Können diese Analysetechniken kategorisiert und klassifiziert werden? Welche Komplexität hat der Analyseprozess via eines Experimentes? Können Optimierungskandidaten und -potenzial entdeckt werden?

Testen ❑ κ) **Werden durch das Analyseinstrumentarium (eine zeitliche Beeinflussung) Fehler in einem System verursacht?**
Die Analyse wird empirisch auf Basis des Analyseinstrumentariums durchgeführt, es werden zeitliche Abläufe in einem Experiment im System variiert. Kann man beliebig, zum Zwecke einer Analyse, in diese zeitlichen Abläufe eingreifen? Kann dies zu funktionalen Fehlern führen? Ist vielleicht sogar **ein umgekehrter Ansatz** möglich: können so Fehler entdeckt werden, die mit herkömmlichen Methoden bzw. Testmethodologien nur schwer zu identifizieren sind, weil sie aufgrund zeitliche Bedingungen nicht deterministisch reproduzierbar sind? So gibt es Fehler, deren Auftreten mit den zeitlichen Abläufen in einem System zusammenhängen, z. B. bei einem Rendezvous warten Prozesse oder es gibt Fehler die mit einer Alterung des Systems zusammenhängen. Welche Fehlerklassen können gefunden werden? Wie kommen diese Fehler zustande und wie kann das Analyseinstrumentarium dazu genutzt werden, diese Fehler zu reproduzieren? Wie sieht der Prozess zum Auffinden solcher Fehler aus?

1.2.2 *Vorgehensmodell und Ergebnisse der Arbeit*

Die Bearbeitung der Aufgabenstellungen und die Lösungen der Teilfragestellungen werden durch das Vorgehensmodell, das in Abbildung 1.6 dargestellt ist, realisiert. Die Pfeile in dem Schaudiagramm visualisieren Abhängigkeiten: eine Thematik am Pfeilfuß wird zum Verständnis einzelner Aspekte der Thematik an der Pfeilspitze benötigt. Die beantworteten **Teilfragestellungen** sind entsprechend gekennzeichnet.

Pfeilsemantik im grafischen Modell

❑ **In Kapitel 2** werden die benötigten Termini für diese Arbeit definiert. Der Untersuchungsfokus wird betrachtet, es wird das zu lösende Problem, eine Performanzanalyse von Systemen, beschrieben. Die Begriffe „nicht deterministisch reproduzierbare Fehler“ und Softwaretest werden eingeführt. Anschließend werden zur Illustration zwölf Szenarios dargelegt, zwei didaktische Illustrationszenarios zum einfachen Nachvollziehen der Performanzanalyse, drei Szenarios zur Testmethodologie und sieben typische IT-Anwendungen mit unzureichender Performanz, die im Rahmen dieser Arbeit zur praktischen Demonstration der erarbeiteten Analysemethodik verwendet werden.

Termini

Szenarios

❑ **In Kapitel 3** werden aktuell bestehende und relevante Forschungsarbeiten, Projekte sowie Ansätze zur Performanzoptimierung und -analyse umfassend begutachtet. Diese werden nach der Phase im Entwicklungsprozess kategorisiert (der frühe modellbasierte Ansatz (Abschnitt 3.2), Ansätze während der Implementierung (Abschnitt 3.3) und späte messungsbasierte Ansätze (Abschnitt 3.4)). In einer abschließenden Diskussion (Abschnitt 3.5) wird aufgezeigt, dass die bestehenden Ansätze unzureichend sind und in der Praxis nicht generell akzeptiert werden.

Stand der Wissenschaft und Technik

❑ **In Kapitel 4** wird das Fundament dieser Arbeit, das Analyseinstrumentarium bzw. der Lösungsansatz, vorgestellt. Die grundlegende Idee wird demonstriert. Die zeitlichen Effekte, wie Verlängerung der Laufzeit (Prolongation), Verzögerung von Zeitpunkten (Retardation) oder Laufzeitverkürzungen (durch simuliertes Optimieren), die durch die Variation von Abläufen realisiert werden können, werden betrachtet.

Analyseinstrumentarium

❑ **In Kapitel 5** wird die Validität (interne Validität bzw. Ceterus paribus-Validität) des Analyseinstrumentariums bewiesen. Dass die Korrektheit bei einem Experiment mit dem Analyseinstrumentarium an sequentiellen Berechnungen beibehalten wird, wird an einer Turingmaschine, einem einfachen aber grundlegenden Maschinenmodell, bewiesen.

Korrektheit

❑ **In Kapitel 6** wird der durch das Analyseinstrumentarium entdeckte Wirkzusammenhang dargelegt. Dieser wird mit Hilfe eines theoretischen Modells, einer *parallelen Registermaschine*, die sich wegen ihrer Einfachheit und Nähe zu realen Systemen zur Darstellung anbietet, entwickelt und bewiesen. Anhand der *Happend-Before Relation* und *Lamport-Uhren* wird der Wirkzusammenhang, bei multiplen Prozessoren bzw. Prozessen und paralleler Ressourcennutzung, formal betrachtet.

Wirkzusammenhang

Experimentierumgebung ❑ **In Kapitel 7** wird aufgezeigt, wie die Experimente durchgeführt werden können. Abschnitt 7.1 erläutert, wie die zeitliche Variation von Software durch eine Instrumentierung möglich wird. Eine Erweiterung ist eine zeitliche Variation mittels einer virtuellen Maschine. Durch die Virtualisierung wird eine größere Kontrolle des zu untersuchenden Systems erreicht. Sie stellt einen ganzheitlichen Ansatz dar, nicht nur Code sondern auch die unterliegende Hardware wird prolongiert. Abschnitt 7.2 führt kurz in die Thematik ein und zeigt die Realisation der Experimentierumgebung.

Mindestanforderungen ❑ **In Kapitel 8** wird eine Methode vorgestellt, um systematisch Mindestanforderungen von Software an die Hardware zu eruieren. Software stellt bestimmte Mindestanforderungen an die Hardware. Sind diese nicht erfüllt kann dies bis zu einer nicht funktionalen Software führen. Die steigende Diversität bei Hardware stellt hierbei insbesondere ein großes Problem dar, Software kann nicht auf allen möglichen Plattformen ausreichend validiert werden. Das Kapitel schließt mit einigen Realszenarios bei denen Mindestanforderungen an die Hardware mit der Experimentierumgebung ermittelt werden.

Analyse ❑ **In Kapitel 9** werden Möglichkeiten zur Identifikation von Optimierungskandidaten vorgestellt. Die gemessenen Daten müssen analysiert werden, hierzu werden verschiedene Analysemöglichkeiten verglichen. Kapitel 9 behandelt die Analyse mit Hilfe multivariater Methoden und der deskriptiven Statistik. Auf dem Fundament des Analyseinstrumentariums aufbauend werden grundsätzliche Analysemethoden vorgestellt, die mit der Variation von Abläufen möglich sind. Diese werden mit anderen Methoden verglichen und Vor- und Nachteile werden evaluiert. Das Potenzial, die Vor- und die Nachteile beider Methoden werden abschließend erörtert und gegeneinander abgewägt. Unter anderem wird ein Verfahren demonstriert, um aus einem System einen Abhängigkeitsgraphen zu gewinnen. Mit dem simulierten Optimieren aus Kapitel 4 lässt sich ein Effekt einer Optimierung im System ausmessen. Welche Vorteile diese Methoden in der Softwareentwicklung und -analyse haben, wird diskutiert. An acht Szenarios wird der Analyseprozess detailliert demonstriert, es wird gezeigt, wie Optimierungskandidaten und -potenzial durch ein Experiment gefunden werden können.

Statistische Analyse

Tests ❑ **In Kapitel 10** wird eine Möglichkeit für Tests mittels der entwickelten Methoden im Softwareentwicklungsprozess dargestellt und an Szenarios exemplifiziert. Unzureichende Synchronisation von Hardware- sowie Softwarekomponenten stellt eine Fehlerquellen dar. Kapitel 10 benutzt das Analyseinstrumentarium aus Kapitel 4 und die Ergebnisse und Erkenntnisse aus Kapitel 6, um unzureichende Synchronisationsmechanismen aufzudecken. Das Kapitel behandelt das Auffinden von, im Allgemeinen nicht deterministisch reproduzierbaren Fehlern (wie *Heisenbugs*, *Races* und *Aging-related faults*) und schließt mit einer Diskussion wie dieses Vorgehen als Testmethodologie im Softwareentwicklungsprozess genutzt werden kann.

Zusammenfassung und Ausblick ❑ **In Kapitel 11** wird diese Arbeit zusammengefasst und ein Ausblick auf mögliche weiterführende Arbeiten gegeben.

Hinweise zum Layout

❑ **Am Anfang** jedes Kapitels befindet sich eine Übersicht über die einzelnen Abschnitte und der dort behandelten Aspekte und Themen. Kapitelübersicht

❑ **Am Ende** jedes Kapitels befindet sich eine Synopsis des Kapitels und eine Beantwortung von Teilfragestellungen. Kapitelsynopsis

Beispiel <Kapitelnummer.Abschnittsnummer.Nummer> Beispiele
Beispiele veranschaulichen und konkretisieren Begriffe und Ergebnisse in dieser Arbeit. Alle Beispiele sind im Register am Schluss der Arbeit unter *Beispiele* aufgeführt. Beispiele

Exkurs <Kapitelnummer.Abschnittsnummer.Nummer> Exkurse
Exkurse sind in sich geschlossene Abhandlungen über Hintergründe und Zusammenhänge. Sie dienen zur weiteren Erläuterung und sollen den Argumentationskurs der Arbeit nicht stören und wurden deshalb ausgegliedert. Alle Exkurse sind im Register am Schluss der Arbeit unter *Exkurse* aufgeführt. Exkurse

Eigennamen, *feststehende* und *englische Begriffe* sind kursiv gesetzt und in das Sachregister integriert. *Sätze* und *Lemmas* sind farbig, *Definitionen* sind grau hinterlegt und nummeriert. kursiv

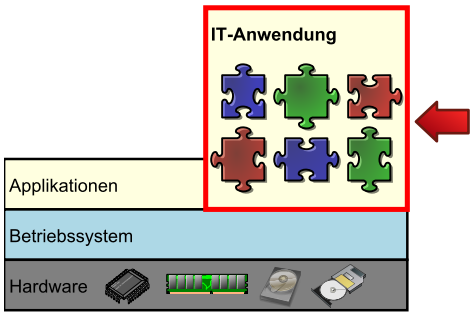


Abbildung 1.5: Eine schematische Darstellung als Überblick des Systems.
Die Schemazeichnung zeigt die Ebenen des untersuchten Systems. An ihr wird beispielsweise die diskutierte Ebene indiziert (wie hier im Bild) oder spezifische Besonderheiten dargestellt.

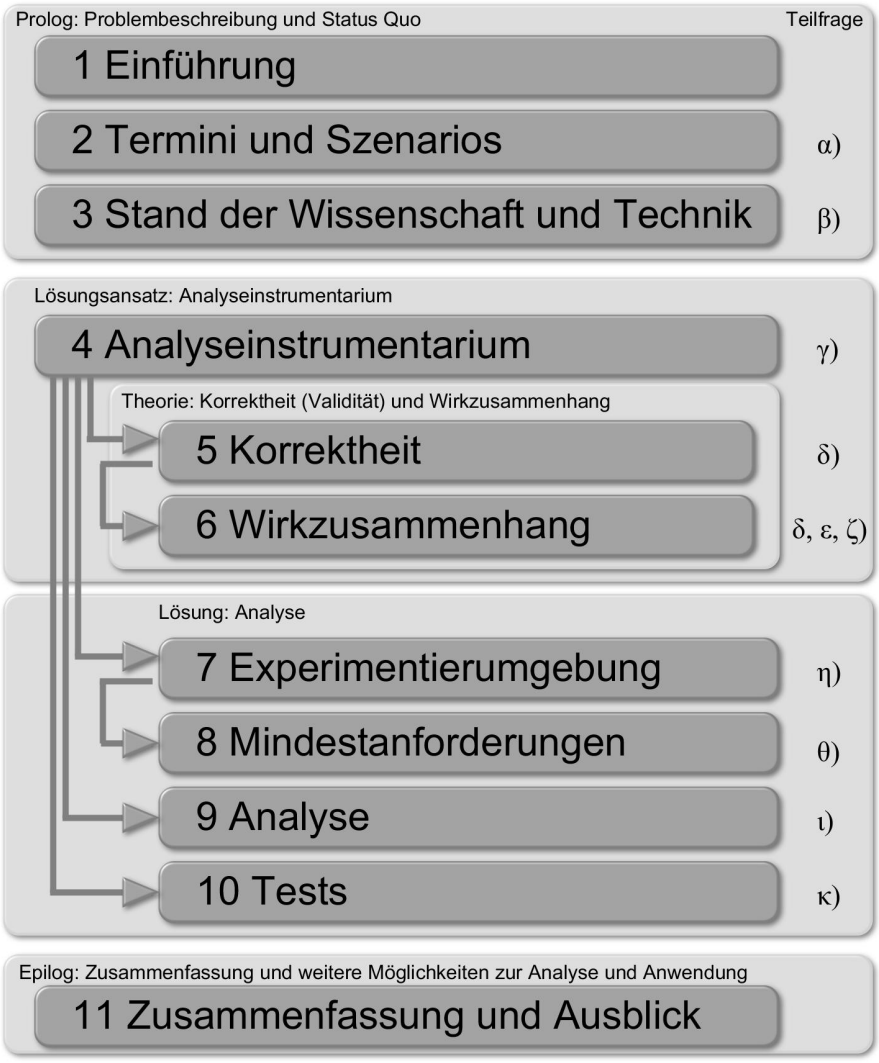


Abbildung 1.6: Vorgehensmodell der Arbeit im Überblick.
❑ **Kapitel** sind durch dunkelgraue Balken repräsentiert, beschriftet mit der Kapitelnummer und dem Titel des Kapitels.
❑ **Beantwortete Teilfragen** stehen hinter den einzelnen Kapiteln.
❑ **Pfeile** visualisieren Abhängigkeiten: ein Kapitel am Pfeilfuß wird zum Verständnis einzelner Aspekte der Thematik an der Pfeilspitze benötigt.
❑ **Thematisch und strukturell zusammengehörende Kapitel** sind mittels der hellgrauen Blöcke gruppiert.

Kapitel 2

Termini und Szenarios

„Wenn man seine Überlegungen nicht damit beginnt, daß man Definitionen gibt, also die Bedeutung der einzelnen Bezeichnungen festsetzt, so ist es, als wenn man eine Rechnung anstellen wollte, ohne den Wert der Zahlwörter eins, zwei, drei zu kennen.“

Thomas Hobbes

In diesem Kapitel werden im Abschnitt 2.1 die für den Status Quo und diese Arbeit notwendigen Begriffe definiert.
Am Ende dieses Kapitels, in Abschnitt 2.2, werden 12 Szenarios vorgestellt. Anhand dieser 12 Szenarios wird die praktische Realisierbarkeit der in dieser Arbeit entwickelten Techniken, basierend auf dem Analyseinstrumentarium, demonstriert.

Übersicht des Kapitels

2.1 Definitionen und Begriffsklärung

2.1.1 Begriffsdefinition Performanz

Das englische Wort „Performance“ besitzt eine Vielzahl von Konnotationen.
Das IEEE Standard Fachwörterbuch der Terminologie der Softwaretechnik definiert Performance als:

1 IEEE Definition Performance

IEEE Definition Performance

The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage [Ei90].

Performanz

In dieser Arbeit wird im Folgenden der, vor allem in der Praxis, breit etablierte deutsche Begriff Performanz anstelle des englischen Terminus Performance für Definition 1 benutzt.

Exkurs 2.1.1 Performance vs. Performanz

Performanz im deutschen Sprachgebrauch

Der Duden definierte 1991 den Ausdruck „Performanz“ nur im Zusammenhang des Sprachgebrauchs [Dro91]. Sprachlich korrekt müsste in dieser Arbeit das Fremdwort „Performance“ oder der deutsche Terminus „Leistung“ benutzt werden.

Aufgrund gleichen etymologischen Ursprungs, ähnlicher phonologischer Wortkontur, fast gleicher Orthographie und allgemeiner Beliebtheit von Anglizismen hat sich jedoch weitgehend das Wort Performanz im deutschen Sprachgebrauch etabliert. Insbesondere existiert kein bedeutungsgleiches Pendant für Performance in der deutschen Sprache. So kann unter Performance die Leistungsfähigkeit, Antwortzeit oder Berechnungszeit und vieles mehr subsumiert werden.

2.1.2 *Begriffsdefinition System*

Systemdefinition

Der Fokus dieser Arbeit liegt auf einer Performanzanalyse von Systemen. In dieser Arbeit wird unter einem System die Gesamtheit von verknüpften, sich eventuell beeinflussenden und interagierenden Elementen (oder Komponenten[1]) verstanden.

2 System

Definition System

Ein System ist die Summe aus folgenden interagierenden Komponenten:

- ❑ ausgeführte Software
- ❑ der angesteuerten bzw. zur Ausführung benötigten Hardware
- ❑ entsprechenden externen Faktoren (auch Umgebung genannt)

Beispiel 2.1.1 und Abbildung 2.1 illustrieren diese Definition.

Beispiel 2.1.1 System

Software besteht mindestens aus:

- Programm bzw. IT-Anwendung
- Daten

Externen Faktoren sind:

- Hardware (die CPU, die Festplatte, . . .)

[1] siehe Definition 2.1.2, Seite 18

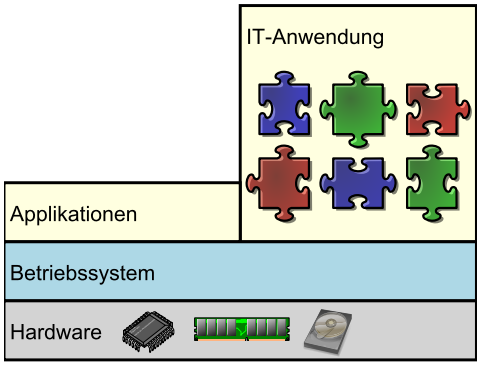


Abbildung 2.1: Ein System bestehend aus Hardware mit unterschiedlichen Hardwarekomponenten, dem Betriebssystem, parallel laufenden Applikationen und der zu untersuchenden IT-Anwendung. Die Puzzleteile symbolisieren unterschiedliche Module (verschiedenster Granularität) die miteinander interagieren. Diese schematische Abbildung des Systems wird durchgehend in dieser Arbeit als Indikator der gerade untersuchten und diskutierten Ebene benutzt.

- Betriebssystem (Daten + Code) (wie z. B. Windows 7 oder Ubuntu mit individuellen Einstellungen)
- Optionale andere Applikationen / Programme (Firewall, Antivierenprogramme, . . .)

Exkurs 2.1.2 klassische Systemdefinition in der Informatik

klassische Systemdefinition

Pionierarbeit zur Sicherheit von Computersystemen leisteten 1973 D. Elliott Bell und Leonard J. LaPadula, als sie eine Basis für entscheidende Forschung durch ein allgemeingültiges, beschreibendes Modell für ein Computersystem legten [BL73]. Hierfür machten sie Anleihen aus Konzepten und Konstrukten der Systemtheorie.

mathematische Systemdefinition

Ein System S ist demnach eine Relation von (abstrakten) Mengen X und Y. Stellt S eine Funktion dar ($S : X \to Y$), spricht man von einem funktionalen System. Zweckdienlich können hier die Elemente aus X als Eingabewerte und die Elemente aus Y als Ausgabewerte betrachtet werden. [BL73, S. 1].

Systemdefinition

In der Arbeit [BL73] wird das mathematische Fundament hinsichtlich sicherer Computersysteme gelegt. Es wird konsequenterweise mit Zustandsmengen des Computersystems weitergearbeitet.

Eine formale, mathematische Systemdefinition, wie sie D. Elliott Bell und Leonard J. LaPadula einführen [BL73], die mit ihrer Arbeit die mathematische Grundlage für sichere Computersysteme gelegt haben, ist in dieser Arbeit nicht nötig. Im Kontext dieser Arbeit können Systeme nicht derart abstrahiert werden, da konkrete Laufzeiteigenschaften der Ressourcen in die Betrachtung mit einfließen. Eine solche, formale Systemdefinition würde deshalb die Argumentation unnötig beschweren und behindern.

2.1.3 *Begriffsdefinition Systemenkomponenten*

Komponenten des Systems

Alle diese Elemente aus Definition 2, dem System, können abstrahiert als Systemkomponenten bezeichnet werden. Systemkomponenten (aus Software und externen Faktoren) können miteinander interagieren und sich gegenseitig beeinflussen. Diese Systemkomponenten werden in dieser Arbeit kurz als Komponenten bezeichnet.

Beispiel 2.1.2 Komponenten

Beispiel für Komponenten

Komponenten bezeichnen:

❑ **Hardwarekomponenten**

Hardwarekomponenten umfassen:

- Prozessoren
- Prozessorkerne
- Speicher (Festplatte, RAM, Flashspeicher, Caches ...)
- uvm.

❑ **Softwarekomponenten**

Je nach analysierter Granularitätsstufe umfassen Komponenten hier:

- Funktionen bzw. Prozeduren, Pakete, Komponenten (siehe Definition 3), etc. – alle diese Softwarekomponenten werden in dieser Arbeit als Module bezeichnet (siehe Abschnitt 2.1.4)
- Das Betriebssystem
- Andere, gleichzeitig ablaufende Programme
- uvm.

❑ **Systeme oder Systembestandteile**

Bei der Kopplung von Systemen (beispielsweise durch ein Netzwerk wie das Internet) kann bei der Analyse gar ein ganzes System (wiederum bestehend aus Hardware- und Softwarekomponenten) eine Komponente repräsentieren.

Komponente: Soft- und Hardware

Der Begriff Komponente wird in dieser Arbeit abstrakt verwendet, es wird keine Unterscheidung zwischen Hardware und Software gemacht. Insbesondere wird durch die in dieser Arbeit verwandte Virtualisierungstechnologie diese Unterscheidung hinfällig, da Hardwarekomponenten virtualisiert bzw. emuliert werden können und demnach aus Software bestehen.

Exkurs 2.1.3 Komponentenbasierten Softwareengineering

Software-Komponenten

In der Informatik ist ein jedoch auch ein anderer, enger gefasster Komponentenbegriff, der sich auf spezielle Softwarebausteine bezieht, üblich [HC01]. Diese Softwarekomponenten, die auch kurz auch als Komponenten bezeichnet werden, müssen konform zu einem Komponentenmodell sein [CS02]. Der hier in dieser Arbeit verwendete Komponentenbegriff ist weiter gefasst, subsumiert jedoch auch den Komponentenbegriff aus dem komponentenbasierten Softwareengineering.

Komponentenbegriff in dieser Arbeit

2.1.4 *Begriffsdefinition Modul*

In dieser Arbeit werden Softwareelemente der betrachteten bzw. analysierten Gliederungsgranularität als Modul bezeichnet. Der Begriff Modul wird in dieser Arbeit nach folgender Definition benutzt:

3 Modul
A module is a contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier. [YC79] zitiert nach [FP97]

Definition Modul

Der Begriff Modul fasst mit Definition 3 insbesondere unterschiedliche Analysegranularitäten einer konkreten Systemimplementierung.

Analysegranularität

Beispiel 2.1.3 Module in einem Java-System
Als ein Modul kann bei einem klassischen Java-System ein Paket, eine Klasse oder eine Methode bezeichnet werden.

Exkurs 2.1.4 Module zur Übersichtlichkeit
Auf Maschinenebene repräsentieren Programme Sequenzen von Speicherzuständen, die mittels eines Befehlsregisters durch die Chiparchitektur decodiert und ausgeführt werden. Bei der Erstellung von Software (z. B. in einer Hochsprache) wird diese für die Übersichtlichkeit zweckdienlich leicht verständlich in Module gegliedert, danach wird durch computerunterstützte Schritte der Code von der Zielplattform maschinell verarbeitbar transformiert (geparsed, kompiliert, gelinkt, etc.).

Gliederung in Module

4 Absolute Ausführungszeit eines Moduls
Die absolute Ausführungszeit bezeichnet eine von aussen gemessene Zeit (wall clock time) eines Moduls bei realen Systemen beziehungsweise die absoluten Berechnungsschritte bei Maschinenmodellen mit uniformer Berechnungszeit.

Die *wall clock time* ist die traditionelle Unixzeit die seit Beginn des *1. Januar 1970, 00:00 Uhr* mitgezählt wird [TT04]. Im Rahmen dieser Arbeit wird sie als nicht (durch die Prolongation) beeinflussbare, uniform ausserhalb des Systems verlaufenden Zeit beliebig messbarer Granularität betrachtet.

wall clock time

2.1.5 *Performanz von Systemen, Performanzanalyse*

Ausreichende Performanz von Systemen ist neben den funktionalen Anforderungen als Qualitätsmerkmal für ein System entscheidend. Voll funktionale Software ohne ausreichende Performanz des Systems ist suboptimal und wird als qualitativ schlecht beurteilt bzw. von potenziellen Benutzern nicht akzeptiert.

Qualitätskriterium Performanz

Exkurs 2.1.5 nicht-funktionale Anforderung Performanz

nicht-funktionale Anforderung

Die Performanz eines Softwaresystems wird klassisch als nicht-funktionale Anforderung in der Anforderungsanalyse klassifiziert [Chu+99, S. 217].
Dass die Performanz der Klasse der nicht-funktionalen Anforderungen zugerechnet wird ist diskussionswürdig und historisch bedingt: In der frühen, sich als ingenieurmäßige Disziplin etablierenden Softwareentwicklung lag der Fokus auf Programmcode, der die geforderten Funktionen umsetzt [Esp+06].
Unterschreitet jedoch die Performanz eines Systems einen gewissen Schwellwert, kann das System infunktional werden. Diese Situation verursacht insbesondere extreme Kosten durch den nachträglichen Aufwand in Verbesserungen und Optimierungen. So gibt es Bemühungen, Performanz parallel neben funktionalen Anforderungen in die Artefakte während des Softwareentwicklungsprozesses zu integrieren, wie z. B. durch [Theo5].

2.1.6 Begriffsdefinition Performanzanalyse

Der Prozess der Analyse von Systemen hinsichtlich ihrer Performanz wird nachfolgend als „Performanzanalyse“ bezeichnet.

5 Performanzanalyse

Performanzanalyse

Eine Performanzanalyse ist die Summe der Analysen (und der damit verknüpften Aktivitäten) während des Lebenszyklus von Software [Sin95], um ausreichende Performanz sicherzustellen oder die Performanz des Systems zu verbessern.

Diese Definition ist angelehnt an die Definition von Woodside, Franks und Petriu [WFP07, S. 1] zum *Software Performance Engineering* (*SPE*), verfeinert um den Begriff „*Lebenszyklus von Software*“ (nach [Sin95]). In jeder Phase des Softwareentwicklungsprozesses stehen unterschiedliche Informationen über das System zur Verfügung [Som95]. So unterscheiden sich die Methoden um die Performanz eines Softwaresystems zu analysieren und zu entwickeln bzw. zu optimieren. Diese werden im nächsten Kapitel umfassend betrachtet, kategorisiert und klassifiziert.

Artefakte in den Phasen

2.1.7 Begriffsdefinition Softwaretest

Testen

Testen ist ein integraler Bestandteil des Softwareentwicklungsprozesses und in allen Softwareprozessparadigmen vorhanden. Ziel von Tests ist es, Fehler in einer konkreten Implementierung eines Softwareprodukts zu finden, die in weiteren Entwicklungsschritten behoben werden. Durch diesen iterativen Prozess wird versucht, den Anteil der Fehler zu reduzieren damit das Softwareprodukt die Spezifikation und damit den Anforderungen und den Anwenderbedürfnissen entspricht.

Der *IEEE Standard 610* von 1990 definiert einen *Softwaretest* als

6 Softwaretest
A Softwaretest is „the process of operating a system or component under specified conditions, observing or recording the results and making an evaluation of some aspects of the system or component“ [Iee]

Softwaretest

Das Überprüfen auf Korrektheit mittels eines (Software-) Tests wird als Validieren bezeichnet.

Validieren

2.1.8 *Begriffsdefinition „nicht deterministisch reproduzierbare Fehler“*

In dieser Arbeit wurde in Definition 2 ein System als die Summe ausgeführter Software, der Hardwarekomponenten und entsprechender externen Faktoren eingeführt.
Auf, oder mit Systemkomponenten werden eventuell asynchrone Operationen ausgeführt, was die Komplexität in einem System erhöht (durch nicht diskrete Zustände, eigene Taktung oder gar keine Taktung, etc.) [She03]. Des Weiteren unterscheiden sich die externen Faktoren der möglichen Zielsysteme (beispielsweise unterschiedliche Hardware, Betriebssystem und unterschiedliche andere Programme etc.). Dies führt zu einer hohen Diversität der Zielsysteme.

schwer beherrschbare Systeme

Der Scheduler (ein Steuerprogramm welches die zeitliche Ausführung von nebenläufigen Abarbeitungslinien regelt) muss nicht deterministisch und kann unterbrechend (preemptiv) sein [Man10]. Durch die Ausführung eines Systems können unerwünschte Effekte wie Speicherlecks, Rundungsfehler oder beendete Prozesse, die dennoch Systemressourcen verbrauchen (*Zombies*) auftreten, die zu weiteren Seiteneffekten führen können.

Scheduler

Aus diskretem Code kann oder wird so ein nichtdeterministisches und chaotisches und dadurch nur schwer beherrschbares System [Ber+10]. Die einzelnen parallelen Abarbeitungslinien müssen zur Abarbeitung einer Aufgabe irgendwann wieder richtig synchronisiert werden, was enormes Fehlerpotential darstellt [HP04].

nichtdeterministisches System

7 Nicht deterministisch reproduzierbare Fehler
Nicht deterministisch reproduzierbare Fehler sind Fehler im System, bedingt durch Hard- oder Software, die nicht bei jedem Programmlauf mit spezifischen Eingabewerten zuverlässig reproduzierbar sind.

Die einzelnen in der Literatur klassifizierten Fehlerklassen, deren Namensursprung, ihre Entstehung und deren Auftreten werden in Kapitel 10 detailliert betrachtet.

2.2 *Szenarios*

Im Folgenden Abschnitt werden zwei einfache Illustrationsszenarios zur Präsentation der Performanzanalyse mittels des Analyseinstrumentariums vorgestellt. Diese sind di-

Illustrationsszenarios

daktische Szenarios zum Nachvollziehen und Erläutern und deshalb extrem simpel konstruiert.

Realszenarios Erweitert von sieben Realszenarios wird die Durchführung der Analyse an bestehenden IT-Anwendungen (mit unzureichender Performanz) in Kapitel 9 dargelegt. An diesen Szenarios wird die praktische Relevanz der hier entwickelten Analyse demonstriert.
Für Kapitel 10 werden drei weitere Szenarios zum Auffinden von nicht deterministisch reproduzierbaren Fehlern dargelegt, an denen Softwaretests mittels des Analyseinstrumentariums durchgeführt werden.

2.2.1 *Illustrationsszenario Methodenaufrufe*

Szenario 1 Illustrationsszenario Methodenaufrufe

didaktisches Beispiel Dieses Szenario wurde bereits zu Illustrationszwecken in [Man07] benutzt. Es hat sich jedoch wegen seiner Einfachheit zum Nachvollziehen und Erläutern bewährt. Da dieses Szenario, das Berechnen der Fakultätsfunktion, ein beliebtes Beispiel aus der Lehre ist, wird es auch in dieser Arbeit als Illustrationsszenario benutzt, insbesondere da hier die Wirkung des Analyseinstrumentariums und die Analyse leicht nachvollzogen werden können und um mit [Man07] konsistent zu sein.

Skalierbare Laufzeit Die Fakultätsfunktion benötigt iterative Berechnungen der CPU, so dass diese Funktion einiges an Zeit (CPU-Zyklen) verbraucht. Durch den zu berechnenden Funktionswert kann die Laufzeit der einzelnen Funktionen leicht skaliert werden.

Vermeidung von Messchwankungen Explizit wurde dieses Szenario so gewählt, dass auf keine Hardwareressourcen zugegriffen werden muss, so sind die Messungen nicht mit größeren Schwankungen (z. B. durch die Festplatte mit Latenzzeiten) behaftet.

Schleifenversion Es wurde der Variante mit einer Schleife Vorzug gegenüber der rekursiven Version gegeben, da bei der rekursiven Version (durch das mitprotokollieren jedes rekursiven Aufrufs) eine größere Datenmenge produziert wird. Das Szenario wurde in Java implementiert. [Man07, S. 34]

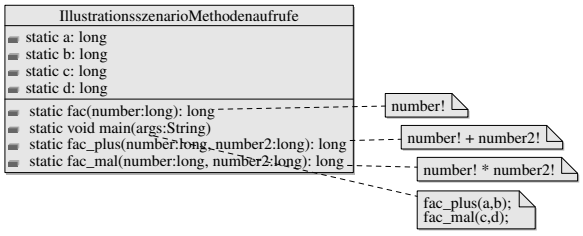


Abbildung 2.2: Klassendiagramm zum Illustrationsszenario.

```
public class IllustrationsszenarioMethodenaufrufe {
  public static long a = 1, b = 2, c = 3, d = 5;
  public static void main(String[] args) {
    long l;

    l = fac_plus(a, b);
    l = fac_mal(c, d);

    System.out.println(l);
  }

  static public long fac_plus(long number, long number2) {
    return number = fac(number) + fac(number2);
  }

  static public long fac_mal(long number, long number2) {
    return number = fac(number) * fac(number2);
  }

  static public long fac(long number) {
    long fakultaet = 1;

    for (int zahl = 1; zahl <= number; zahl++) {
      fakultaet = fakultaet * zahl;
    }
    return fakultaet;
  }
}
```

Listing 2.1: Java: Illustrationsszenario Methodenaufrufe

Szenario 2 Illustrationsszenario Methodenaufrufe für das simulierte Optimieren

Simulationsansatz

Beim simulierten Optimieren soll durch ein Experiment auf Basis des Analyseinstrumentariums keine Struktur erkannt oder ein Modell der zu untersuchenden IT-Anwendung erstellt werden, sondern, ein System wird mit einer Komponente (Methode, etc.) ausgeführt, die (relativ) performanter ist. Essentiell zur Demonstration sind hier absolute Laufzeitmesswerte und keine Struktur des Systems.
Zum einfachen Nachvollziehen des simulierten Optimierens[2] wurde die main-Methode in diesem Szenario ersetzt. Hier wird viermal die Methode fac aufgerufen, eine Methode mit gleicher Definition ist umgeschrieben (CopyOffac), diese wird mit besserer Performanz simuliert. Das Szenario hat sich bereits in [Mano7] zur Darstellung des simulierten Optimierens bewährt.

```
public class IllustrationsszenarioSimuliertesOptimieren {
  public static void main(String[] args) {
    l = fac(2);
    l = fac(2);
    l = CopyOffac(2);
    l = fac(2);
    l = fac(2);
  }

  static public long fac(long number) {
    long fakultaet = 1;

    for (int zahl = 1; zahl <= number; zahl++) {
      fakultaet = fakultaet * zahl;
    }
    return fakultaet;
  }

  static public long CopyOffac(long number) {
    long fakultaet = 1;

    for (int zahl = 1; zahl <= number; zahl++) {
      fakultaet = fakultaet * zahl;
    }
    return fakultaet;
  }
}
```

Listing 2.2: Java: Illustrationsszenario Methodenaufrufe für das simulierte Optimieren

[2] siehe Abschnitt 4.3.3, Seite 67

2.2.2 *Szenario Imageshuffle*

Szenario 3 Imageshuffle

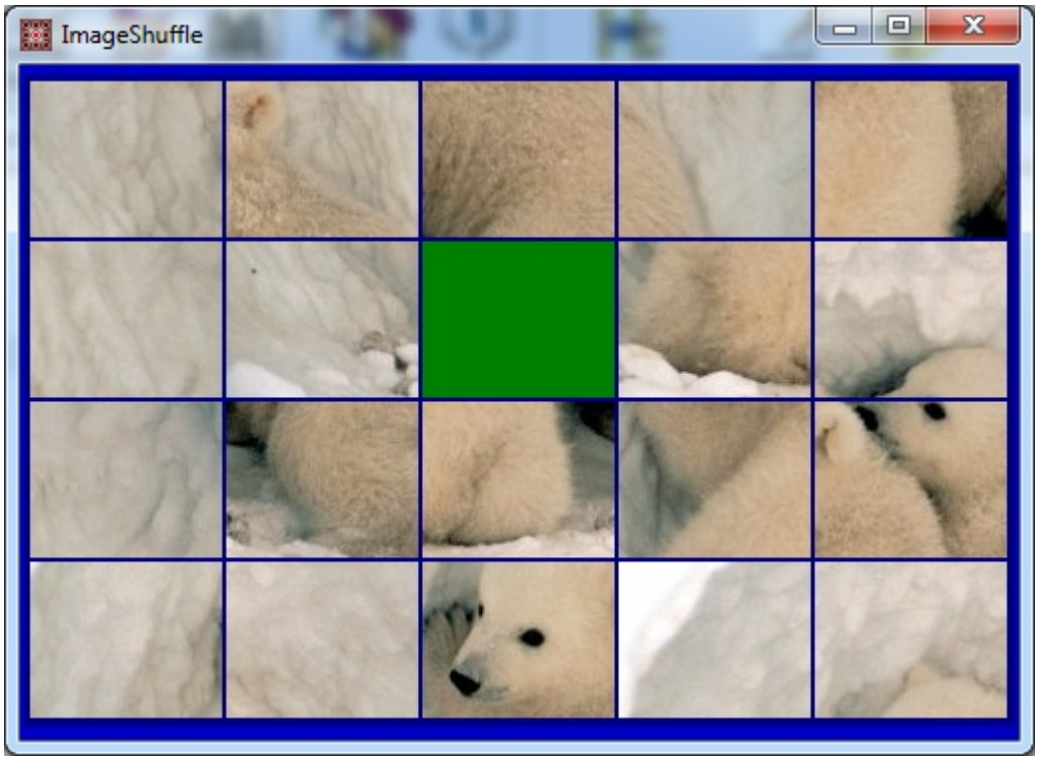


Abbildung 2.3: Screenshot – Ausführung von Imageshuffle.jar

ImageShuffle ist ein Schiebepuzzle (ein kleines Spiel) und wurde in Java implementiert. Es kann von `http://javaboutique.internet.com/ImageShuffle/` [McN02] oder `http://www.florian-mangold.com/diss/ImageShuffleSrc.zip` bezogen werden.
Es besteht aus einer einzigen *Jar-Datei* und ist nicht nebenläufig implementiert. Dieses Programm wurde bereits in [Man07] zu Illustrationszwecken genutzt.

Java Schiebepuzzle

2.2.3 *Szenario Hardwareabhängigkeit – Android Betriebssystem für Netbooks und Smartphones*

Abbildung 2.4: Netbook auf einem Notebook. Netbooks sind die kleinen, preisgünstigen, jedoch nicht ganz so leistungsfähigen Verwandten von Notebooks.

Netbooks

Das am schnellsten wachsende PC Segment sind momentan die *Netbooks* [Del09]. Begonnen hat diese Revolution mit dem *Asus Eee PC* [Bor08] Ende des Jahres 2007 [Des09]. Um den extrem günstigen Preis im Vergleich zu Notebooks zu erreichen, muss hier ein Kompromiss in der Hardware eingegangen werden. Netbooks sind demnach nicht mit den leistungsfähigsten Prozessoren und wenig Speicher ausgestattet [Des09].

Smartphones

Smartphones sind heute nicht mehr wegzudenkende mobile Endgeräte. Diese basieren auf einer Synthese von *PDA*s (*Personal Digital Assistants*) und Mobiltelefonen [HA09, S. 65]. 1990 wurde die Ära der Smartphones mit dem *Blackberry* eingeleitet, insbesondere war die *Siemens AG* ein entscheidender Pionier für Multimediafunktionen (z. B. ein Patent auf Farbdisplays [HRW98], ein Patent zur Speicherung von Musikdateien [EC00]

und ein Patent auf Mobiltelefone mit Tastatur und Kamera [Hil98]) bei mobilen Endgeräten. Im Gegensatz zu Mobiltelefonen bauen die meisten Smartphones auf einem Betriebssystem auf [AI08; LY09].

Android

Android ist ein freies, *open source* Betriebssystem oder Framework, von Google entwickelt für mobile Geräte wie die beschriebenen *Smartphones* und *Netbooks* [Sha+10]. Mit einer möglichst einfachen Entwicklungsmöglichkeit für Anwendungen („*Apps*“), freien Verfügbarkeit und Portabilität der Entwicklungsplattform hat Android definitiv das Potenzial zum Marktführer [HA09]. Insbesondere ist Android nicht an spezifische Hardware gebunden, im Gegensatz zum größten Konkurrenten *iOS* von *Apple*.

Hardwareheterogenität

Jedoch bringt der Vorteil der vielen Zielplattformen auch eine hohe Hardwareheterogenität mit sich. Durch die diversen Leistungskenngrößen der Hardwarekomponenten wird die Performanz der Software bzw. des Systems beinflusst. Mobile Endgeräte (*Netbooks*, *Smartphones*, ...) sollen produziert werden, so billig wie möglich aber dennoch so leistungsfähig wie nötig, damit beispielsweise aktuelle *Open Source Software* auf diesem System ohne Performanzprobleme lauffähig ist. Es muss eruiert werden, welche Hardwarekomponenten verbaut werden dürfen, so dass das Preis/Leistungverhältnis stimmt und Standardsoftware ausgeführt werden kann. Dass dies nicht trivial ist, sieht man an den aktuellen und akuten Performanzproblemen des Betriebssystems *iOS 4* auf dem *iPhone 3G*.

Szenario 4 Speicher mit unterschiedlichen Leistungskenngrößen

heterogene Speicherkomponenten

Für ein Android-System (bzw. allgemein übertragbar auf generelle Systeme) muss entschieden werden, welcher Speicher eingesetzt wird. Die Preise, physikalischen Realisationen und vor allem die Leistung dieser variieren jedoch stark. So unterscheidet man von der grundsätzlichen Technologie zwischen *Mobile DDR* (z. B. Mobile DDR SDRAM K4X51163PC in einem Szenario von Cho u. a. [Cho+08]), *SDRAM*, *DDR1*, *DDR2 SDRAM*, *DDR3 SDRAM* und vielen mehr.

Preis- und Leistungsverhältnis

Beispielsweise kann Speicher mit einer hoher Übertragungsrate, geringer Latenzzeit und einer kurzen Suchzeit eingesetzt werden, der preislich aber signifikant teurer ist, als eine funktional gleichwertiger Speicher mit geringeren Leistungskenngrößen. Hierbei wird ein Benchmarkprogramm eingesetzt, um die Leistungskenngrößen zu testen oder die implementierte Software wird mit der entsprechenden Hardwarekomponente getestet, ob das System die Spezifikation erfüllt.

Abbildung 2.5: Ein *Smartphone* ist eine Synthese aus Mobiltelefon und Minicomputer bzw. persönlichem Organizer (*PDA – Personal Digital Assistant*).

Szenario 5 asymmetrische Leistungskenngrößen bei einer Flashdisk

asymmetrische Leistungskenngrößen

Bei einem bestehenden System soll die Festplatte durch ein Flashlaufwerk ausgetauscht werden. Jedoch sind die Leistungskenngrößen bei einer Flashdisk im Gegensatz zu einer herkömmlichen Festplatte asymmetrisch, so benötigt ein Schreiben länger als ein Lesen von der Disk.

Es ist nun fraglich, ob die Technologie einfach ausgetauscht werden kann und welche Leistungskenngrößen die Flashdisk mindestens besitzen muss, damit das System fehlerfrei und ansprechend reagiert.

Benchmarking

Hierbei werden unterschiedliche Benchmarkprogramme eingesetzt um die Leistungskenngrößen zu testen oder die implementierte Software wird mit der entsprechenden Hardwarekomponente getestet, ob das System die Spezifikation erfüllt.

2.2.4 *Szenario Simpler Data Race*

Szenario 6 Simpler Data Race

Illustration zu einem Data Race

In Listing 2.3 ist ein einfach nachvollziehbarer *Data Race* als Illustrationsszenario angegeben. `NUMBEROFTHREADS` Threads werden gestartet, beide inkrementieren in einer Schleife das gemeinsame Objekt `count` um den Zahlenwert `INCREMENTS`, wobei sie den Zahlenwert in eine lokale Variable zwischenspeichern, was zum *Data Race* führen kann. So sollte die Variable `count` am Ende des Programms (zu meist) den Wert `NUMBEROFTHREADS * INCREMENTS` haben. Sequentialisiert würde sich dieser Wert analog ergeben.

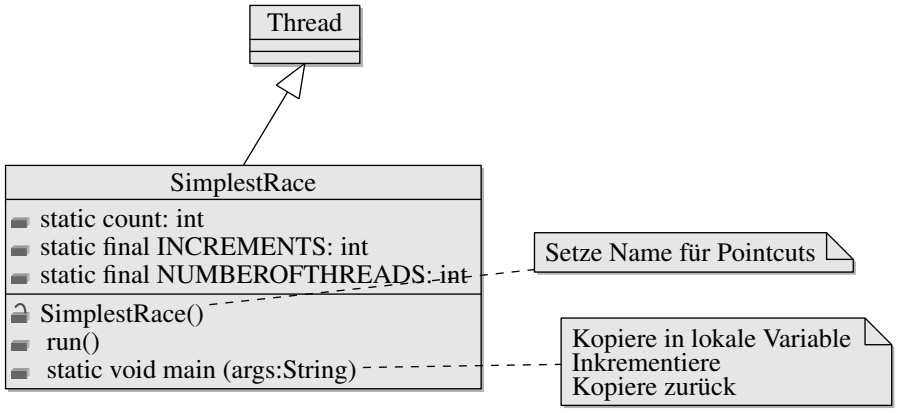


Abbildung 2.6: Klassendiagramm: SzenarioSimplestRace – ein einfach nachvollziehbarer *Data Race*.

```
public class SzenarioSimplestRace extends Thread {
  public static int count = 0; // shared object
  public static final int NUMBEROFTHREADS = 5;
  public static final int INCREMENTS = 100;

  SzenarioSimplestRace(String name) {
    super(name);
  }

  public void run() {
    {
      int y;
      for (int i = 0; i < INCREMENTS; i++) {
        y = count; // read shared object
        y++; // increment shared object
        count = y; // save shared object
      }
    }
  }

  public static int startSimplestRace() {
    SzenarioSimplestRace[] t = new SzenarioSimplestRace[NUMBEROFTHREADS];
    for (int i = 0; i < NUMBEROFTHREADS; i++) {
      t[i] = new SzenarioSimplestRace(i + "");
      t[i].start(); // start this Thread (run Method)
      t[i].setName(i + ""); // Pointcut-Identifier
    }
    for (int i = 0; i < NUMBEROFTHREADS; i++) {
      try {
        t[i].join(); // wait 4all Threads
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    return count;
  }

  public static void main(String args[]) {
    count = 0;
    System.out.println(startSimplestRace() + "\t"); // start Scenario
  }
}}
```

Listing 2.3: Java: Simpler Data Race

2.2.5 *Szenario Heisenbug*

Szenario 7 Heisenbug

Ein Beispiel für einen Heisenbug ist eine Kommunikation zwischen unterschiedlichen Schichten eines Systems. So erfolgt beispielsweise ein Senden einer Nachricht der einen Schicht bevor die andere Schicht empfangen kann. Ein berühmtes und berüchtigtes Beispiel war der *Texas-Bug* der *Therac-25* bei der eine erfahrene Benutzerin die Daten zu schnell änderte, die Empfängerschicht konnte die Änderung nicht verarbeiten (siehe Exkurs 10.3.1 auf Seite 267).

Kommunikation zw. Systemschichten

Ein simples Beispiel für einen Heisenbug ist in Listing 2.4 implementiert. Hierbei ist ein Peripheriegerät *Memory-Mapped*, d. h. ein *I/O Register* des Gerätes ist auf eine Adresse des Hauptspeichers abgebildet. Mittels Mikroinstruktionen zum Speicherzugriff kann so das Gerät angesteuert werden. Ein berühmtes Beispiel für ein *Memory Mapping* ist die *Upper memory area* der *IBM PCs* für das *Video RAM* und das *Bios* [CCG03; Fra03].

Memory-Mapped

```
...
Wait4device:
  mov bx, [Portnummer]     ; Port is memory mapped to Portnummer
  and bx, 1                ; Logical AND with 1
  cmp bx, 0                ; Device was ready?
  je Wait4device           ; Not yet ready!
  ...
Write2device:              ; Write to the Device
  ...
Calculation:               ; Calculate very important Data
  ...
main:
  jmp Calculation          ; Calculate
  ; jmp Wait4device        ; Wait until the device is ready
  jmp Write2device         ; When ready, write to the device
  ; Test if the programm has finished
  je main                  ; When not finished, start again
```

Listing 2.4: x86 Assembler: Berechnung und Ausgabe mit Polled I/O

Bei Listing 2.4 kann es zu einem Heisenbug kommen. Hier wird kein *Polled I/O* gemacht, die entsprechenden Routinen bzw. Prozeduren dafür sind auskommentiert (`Write2device`). Abhängig von der Zeitdauer der Berechnung kann das entsprechende Peripheriegeräte (nicht) bereit sein. Ist es nicht bereit, geht die gesendete Nachricht verloren, es kann ein Heisenbug auftreten. Der Effekt des Heisenbugs kann unterschiedlich sein, von „verschluckten“ Pixeln bis zum *Texas-Bug* (siehe Exkurs 10.3.1 zur *Therac-25* auf Seite 267).

Polled I/O

2.2.6 Szenario Aging-related fault – Patriot Missile

Szenario 8 Aging-related fault – Patriot Missile

Patriot Missile Aging-related Fault Szenario

Analog zum realen *Aging-related fault* des Patriot Missile Systems [GMT08; Bal+10; Off92][LMT10, S. 125] (siehe Seite 10.4.1) gibt es bei diesem Szenario eine Akkumulation von Rundungsfehlern. Am 25. Februar 1991 konnte im Golfkrieg in Saudi Arabien eine *SCUD* Rakete von einem *Patriot Missile* System nicht abgefangen werden, was zu 28 Toten und fast 100 Verletzen führte [Bal+10].

Akkumulation von Rundungsfehlern

Der *Aging-related Fault* war eine Akkumulation von Rundungsfehlern. Intern wurden Ganzzahlwerte in reelle Zahlen durch eine Multiplikation mit $0,1$ mit einem 24-Bit (Festkomma-)Register konvertiert [KT07].
Der Zahlenwert $0,1$ ist $0,0001100110011001100110011\ldots_2 = 1/2^4 + 1/2^5 + 1/2^8 + 1/2^9 + 1/2^{12} + 1/2^{13} + \ldots$ und damit periodisch und endet nicht.
Mit einem 24 Bit Festkommaregister ergibt sich folgender Zusammenhang:

$$0,1_{10} \mathrel{\widehat{=}} 0,000110011001100110011001_2 = \tfrac{209715}{2097152}$$

Somit ergibt sich eine Fehlergröße von:

$$\tfrac{1}{10} - \tfrac{209715}{2097152} = 0,000000095367431640625 = \tfrac{1}{10485760}$$

Der Zahlenwert wird jede Zehntelsekunde inkrementiert, was zu folgender Akkumulation der Rundungsfehler führt [Cha08]:
Somit ergibt sich eine Fehlergröße ***pro Stunde (x)*** von:

$$\left(\tfrac{1}{10} - \tfrac{209715}{2097152}\right)(3600 \cdot 10 \cdot x) = \tfrac{225}{65536}x$$

Nach bereits 8 Stunden kann eine *SCUD Rakete* durch diesen Fehler nicht mehr geortet werden. Bei der geschätzten Laufzeit von hundert Stunden am 25. Februar 1991 ergibt sich ein Fehler von $0,34332275390625$ Sekunden.

angelehntes Beispielsprogramm

Listing 2.5 zeigt ein C-Programm angelehnt an diesen Fehler. Alle Zehntelsekunde wird die Variable t vom Typ **float** inkrementiert. Nach 100 Stunden wird dieser Wert mit der tatsächlich vergangenen Zeit verglichen. Auf einem normalen System (mit *wall clock time*) müssten 100 Stunden abgewartet werden, um den Fehler zu reproduzieren. Durch das Analyseinstrumentarium soll der Fehler „schneller“ und zielgerichteter reproduziert werden können.

```c
#include <stdio.h>
#include <time.h>

void wait ( int tenthOfASecond )
{
  clock_t endwait;
  endwait = clock () + tenthOfASecond * CLOCKS_PER_SEC / 10 ;
  while (clock() < endwait) {}
}

int main ()
{
  int n;
  float t=0;
  clock_t starttime,endtime;
  double deltatime;

  starttime = clock();

  printf ("Starting countdown ...\n");
  for (n=10*60*60*100; n>0; n--) /*100hours=1sec*60*60*100*/
  {
    wait (1); /* wait tenth of second */
    t += 0.1;
  }
  printf ("FIRE!!!\n");

  endtime = clock();
  /* Calculate time difference */
  deltatime = ((double) (endtime - starttime)) / CLOCKS_PER_SEC;

  printf ("Time passed: %f\n" , deltatime );
  printf ("Patriot Missle Time: %f\n" , t );
  printf ("Delta: %f seconds \n" ,   deltatime - (double) t );
  return 0;
}
```

Listing 2.5: C: Patriot Missle – Aging-related fault

2.2.7 *Szenario Webcrawler – WebFrame*

Szenario 9 Webcrawler WebFrame

Java Web Crawler

1998 veröffentlichten Blum u. a. [Blu+98] online den Artikel „*Writing a Web Crawler in the Java Programming Language*“, der beschreibt, wie mit der damals neuartigen Java-Technologie ein *Webcrawler* implementiert werden kann. Dieser Artikel ist immer noch das erste Ergebnis wenn nach den Schlüsselwörtern *Java* und *Webcrawler* in Suchmaschinen (z. B. *Google*) gesucht wird und wird gerne als Vorlage und Beispiel für einen Webcrawler genommen.

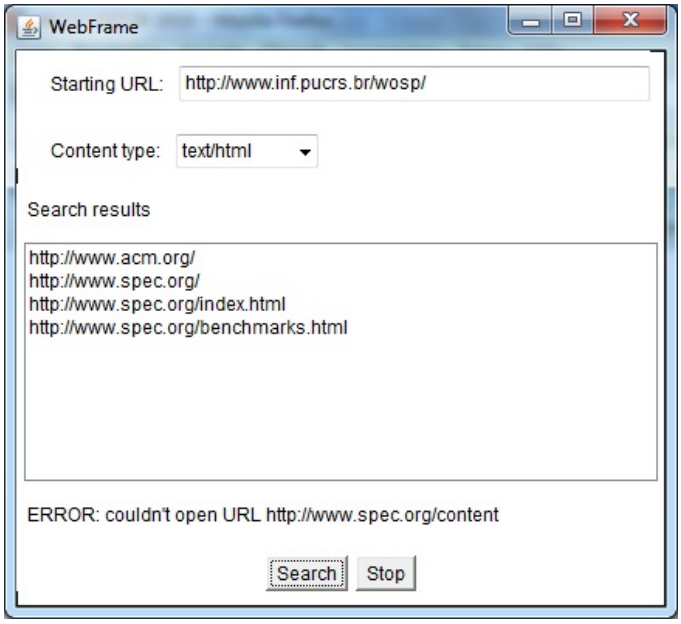


Abbildung 2.7: Screenshot: WebFrame – ein Java Webcrawler

veraltete Methoden

Dieses Demo wurde mittels *JDK 1.1.3* implementiert und ist so, wie auf der Website angegeben, nicht mehr funktional. So muss das Paket `List` importiert werden (**import** java.awt.List;) und `InputStream` muss durch einen `BufferedInputStream` ersetzt werden da ansonsten die Methode `guessContentTypeFromStream` den Inhalt des Streams nicht richtig „schätzen“ kann und immer **NULL** zurückliefert.
Der Originalcode kann von `http://java.sun.com/developer/technicalArticles/ThirdParty/WebCrawler/WebCrawler.java` heruntergeladen werden, die momentan lauffähige Version, mit den erwähnten Veränderungen, kann von `http://www.florian-mangold.com/diss/WebCrawler.java` bezogen werden.

2.2.8 *Szenario Component-based Model Checker – cmc*

Szenario 10 Component-based Model Checker – cmc

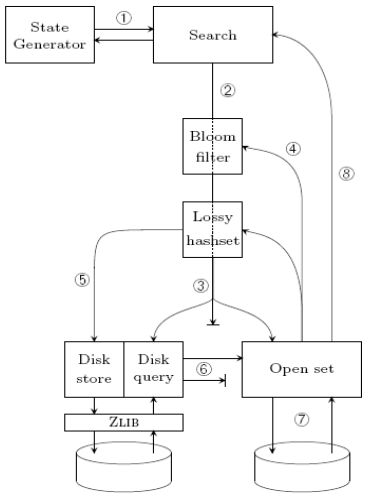


Abbildung 2.8: Kommunikationsfluss des cmc-Modelcheckers (CMC - Component-based Model Checker). Pfeile indizieren den Kommunikationsfluss (der gecheckten Zustände) der Komponenten. Grafik zitiert nach [HW06b].

Modelchecker

Ein expliziter Modelchecker realisiert eine Suche in einem (zum Teil extrem großen) Graphen, welcher das Transitionssystem eines modellierten Programms darstellt [Man07, S. 68]. Von Hammer und Weber [HW06b] wurde in Abschnitt 4 bis 5 der Arbeit „To Store Or Not To Store Reloaded: Reclaiming Memory On Demand“ dieser Modelchecker detailliert beschrieben. Durch eine Auslagerung auf die Festplatte soll die Zustandsexplosion bewältigt werden [HW06a]. So können Modelle mit mehr als 10^9 Zuständen „gechecked“ werden. Der Modelchecker kann von [Ham06] online bezogen werden[3], die Dissertation von Hammer [Ham09] beschreibt den Modelchecker als Fallstudie in Kapitel 5 ausführlich.

interessanter Anwendungsfall

In [Man07] wurde dieser explizite Modelchecker der von Hammer und Weber [HW06b], in *C++* implementiert wurde, mit strukturentdeckenden Methoden untersucht. Durch die Kombination einer externen Ressource (die Festplatte) und berechnungsintensiver Komponenten (die Hashfunktion) ist dies ein sehr interessanter und durch die Veröffentlichungen vor allem gut dokumentierter Anwendungsfall für eine Performanzanalyse aus der Praxis, an dem die Analyseergebnisse didaktisch gut präsentiert werden können.

[3] `http://www.pst.ifi.lmu.de/~hammer/cmc/cmc-0.1.tar.bz2`

2.2.9 *Szenario industriellen Steuerungssystems – SBT FS20*

Szenario 11 industriellen Steuerungssystems – SBT FS20

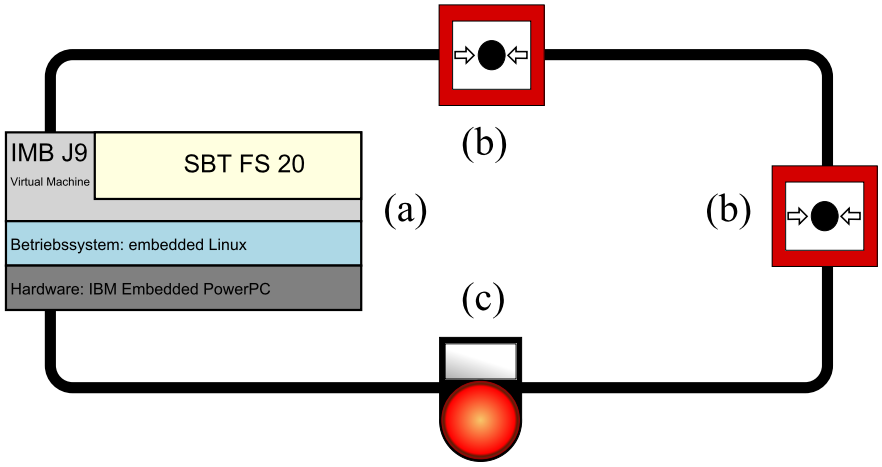


Abbildung 2.9: Sinteso^{TM} industrielles Steuerungssystems der Siemens Building Technology mit Systemkomponenten, realisiert als räumlich verteiltes Client-Server System mittels eines verbindenden Netzes (Fire Control Network).
❑ **(a)** bezeichnet die Ausführungsplattfom.
❑ **(b)** bezeichnet entsprechende Sensoren: von hochentwickelten Spezial- und Brandmeldern bis zum gängigen Handfeuermelder.
❑ **(c)** bezeichnet entsprechende Alarmierungsmittel, z. B. Sirenen, optische Signale etc.

Brandmelde-System

Sinteso^{TM} ist ein Brandmelde-System der *Siemens Building Technology* (SBT) [KM06]. Ausführungsplattform ist ein *embedded PowerPC* von *IBM* [Ali+04] mit *embedded Linux* [Yag03] als Betriebssystem und einer IBM J9 virtual machine [Helo2]. [Mano6] Das System basiert auf einer *Client-Server-Architektur*, verknüpft durch ein *Fire Control Network* (*FCN*) (vgl. [Lono6]), auf dem sich räumlich verteilte Komponenten (z. B. Sensoren) *subscriben* oder *unsubscriben* können (siehe Abbildung 11). [Mano6, S. 4] Der Server wurde nebenläufig mittels Java Threads implementiert, um auf Nachrichten oder Events der verteilten Komponenten (Clients/Sensoren) reagieren zu können [Mano6, S. 4]. Das Projekt *SB FS20* spiegelt den Status Quo vieler Performanzprobleme wieder. Ein großes System, verteilt entwickelt, soll in seiner Performanz verbessert werden. Der Code liegt nur teilweise offen, zur Analyse stehen wenig Informationen und Dokumentation zur Verfügung. Eine weitere Diskussion über die Optimierung des Szenarios befindet sich in Exkurs 3.3.1 (Seite 48). In Exkurs 4.3.2 (Seite 70) wird dargelegt, wie dieses Szenario zur Entwicklung der Analysetechnik beigetragen hat.

Client-Server-Architektur

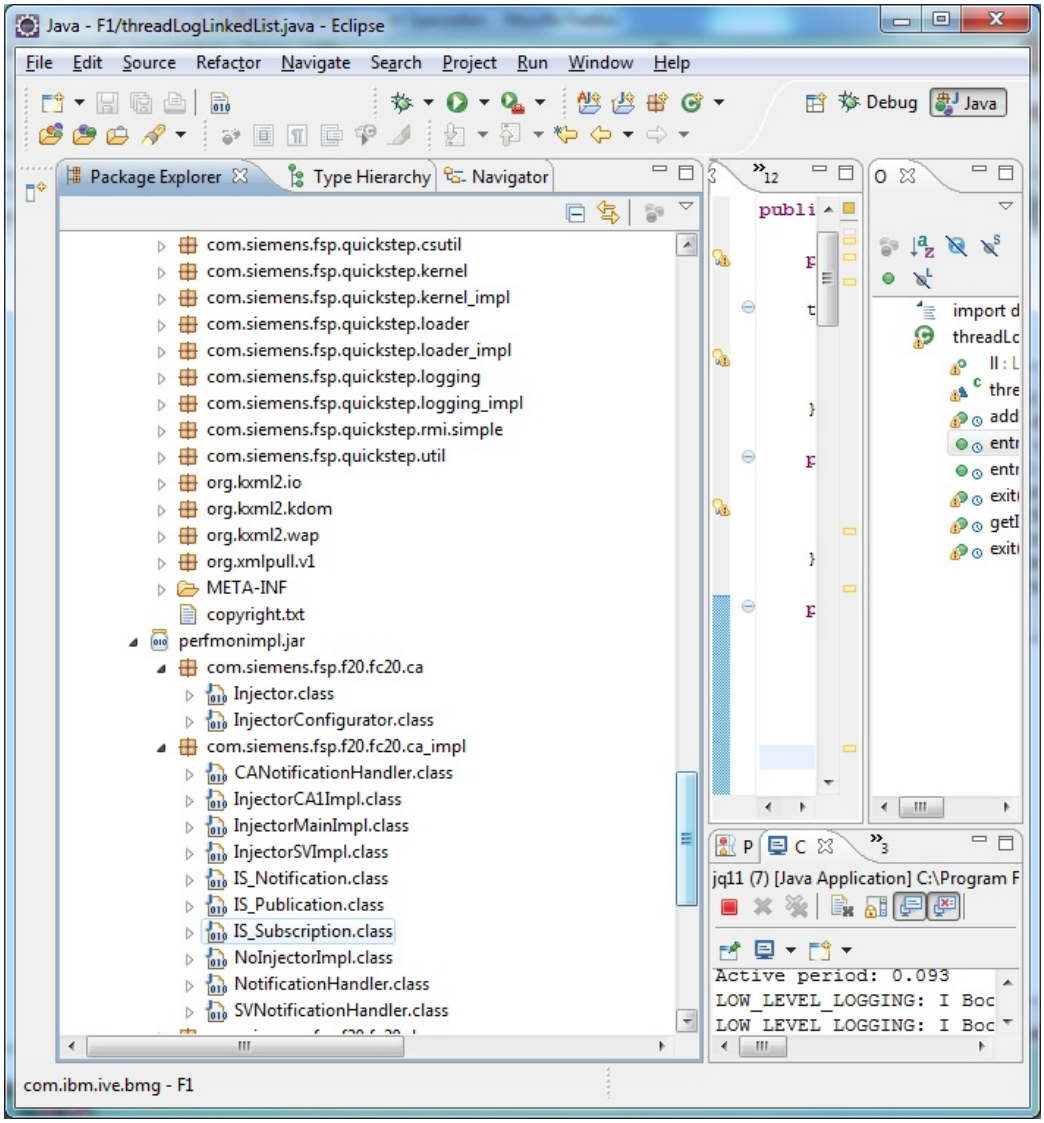


Abbildung 2.10: Ein Screenshot der Projektstruktur von Szenario 11 – industrielles Steuerungssystem. Das Projekt wurde verteilt entwickelt, baut auf externen Jar-Files auf und ist mehrere Megabyte groß. Der Code ist, bzw. war, zur Analyse nicht zugänglich.

2.2.10 *Szenario komponentenbasierter Webcrawler*

Szenario 12 komponentenbasierter Webcrawler

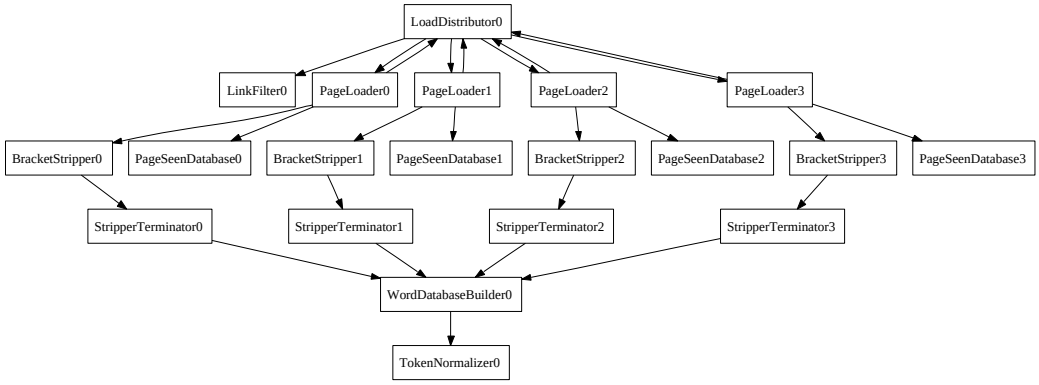


Abbildung 2.11: Komponenten des Webcrawlers. Pfeile indizieren den Kommunikationsfluss.

Testapplikation Der komponentenbasierte Webcrawler ist eine einfache Testapplikation, geschrieben mit einem Framework zur Entwicklung von parallelen, komponentenbasierten Softwaresystemen in Java. Die Architektur ist in Abbildung 2.11 gezeigt.

Nadelöhr Das Nadelöhr eines Webcrawler ist der Zugriff auf die Webseiten via Internet, deshalb ist ein Webcrawler parallel (via Threads) implementiert.

Arbeitsweise Beginnend mit einer initialen Webseite werden verlinkte Seiten geladen und die Links extrahiert. Die Bezeichnungen werden „normalisiert“, d. h. der lexikalische Kern wird ermittelt. Abschließend werden die so normalisierten Linkbezeichnungen in einer Datenbank gespeichert.

Szenario	Stichwort	S.	Durchführung	S.
Szenario 1	Illustrationsszenario 1 – strukturentdeckend	22	Abschnitt 9.5	217
Szenario 1	Illustrationsszenario 1 – modellbildend	22	Abschnitt 9.6	223
Szenario 2	Illustrationsszenario 2 – Simulation	24	Abschnitt 9.7	228
Szenario 3	Schiebepuzzle	25	Abschnitt 9.8	231
Szenario 4	Zugriffsgeschwindigkeit	27	Abschnitt 8.3.2	191
Szenario 5	Assymetrische Zugriffsgeschwindigkeit	28	Abschnitt 8.3.3	196
Szenario 6	Java Data Race	29	Abschnitt 10.2.4	263
Szenario 7	Heisenbug durch ein Memory Mapping	31	Abschnitt 10.3.4	270
Szenario 8	Patriot Missile	32	Abschnitt 10.4.1	273
Szenario 9	Webcrawler WebFrame	34	Abschnitt 9.9	234
Szenario 10	CMC Modelchecker	35	Abschnitt 9.10	238
Szenario 11	Steuerungssystem SBT FS 20	36	Abschnitt 9.11	243
Szenario 12	komponentenbasierter Webcrawler	38	Abschnitt 9.12	248

Tabelle 2.1: Die Durchführung der Szenarios. Die letzten beiden Spalten geben die jeweiligen Abschnitte und die Seite an, hier werden Performanzanalysen, Tests oder Simulationen durchgeführt.

2.3 *Zusammenfassung und Beantwortung von Teilfragestellung α*

Definition der Termini

- In Abschnitt 2.1.5 wurde der deutsche Begriff „*Performanz*“ abstrakt und sehr technisch aufbauend auf einer *IEEE Definition* für diese Arbeit präzisiert.
- Darauf aufbauend wurde der *Performanzanalyse*, angelehnt an eine aktuelle wissenschaftliche Veröffentlichung, expliziert.
- Zuvor wurde ein System sehr technik- und praxisnah definiert (als Summe von ausgeführter Software inklusive von Daten, der Hardware und externer Faktoren). Durch ein Beispiel wurde der Begriff exemplifiziert.
- Die in dieser Arbeit untersuchten Elemente dieser Systeme, wie Module, Softwarekomponenten oder Hardwarekomponenten, wurden beschrieben und durch Beispiele veranschaulicht.
- Der Begriff *Softwaretest* wurde nach einer IEEE Definition eingeführt.
- Darauf aufbauend wurden „*nicht deterministisch reproduzierbare Fehler*“ definiert. Das sind Fehler die nicht zuverlässig reproduziert werden können und deshalb in einem *Softwaretest* oft nicht gefunden werden können.

Die Vorteile der in diesem Kapitel gegebenen abstrakten und sehr technischen Definitionen sind die breite Anwendbarkeit der in dieser Arbeit entwickelten Methodologie in der Praxis.

Szenarios

Zur Demonstration dieser Anwendbarkeit wurden 12 Szenarios dokumentiert:

- Zwei Illustrationsszenarios zum Nachvollziehen und
- sieben Szenarios aus der Praxis, ein breites Spektrum von IT-Anwendungen anhand deren die in dieser Arbeit entwickelte Methodologie exemplifiziert wird,
- drei Szenarios zum Test auf *nicht deterministisch reproduzierbare Fehler* für Kapitel 10.

Teilfragestellung α ✔

Somit wurde der Betrachtungsfokus und die Termini dieser Arbeit präzisiert und zwölf Darstellungs- und Praxisszenarios dokumentiert, was Teilfragestellung α beantwortet.

Kapitel 3

Stand der Wissenschaft und Technik

„Wenn die anderen glauben, man ist am Ende, so muß man erst richtig anfangen."

Konrad Adenauer

Übersicht des Kapitels

Das folgende Kapitel 3 zeigt einen umfassenden Überblick über die bestehenden Ansätze zur Performanzanalyse. Hierbei werden die Methodologien in Abschnitt 3.1 nach der Phase im Entwicklungsprozess kategorisiert und der frühe modellbasierte Ansatz in Abschnitt 3.2, der implementierungsbasierte Ansatz in Abschnitt 3.3 und der späte messungsbasierte Ansatz in Abschnitt 3.4 behandelt. In einer Diskussion in Abschnitt 3.5 wird aufgezeigt, dass die bestehenden Ansätze teilweise unzureichend sind und, v.a. im industriellen Umfeld, nicht generell akzeptiert werden.

3.1 Kategorisierung der Ansätze

Performanzanalyse im Softwarelebenszyklus

Woodside et al. [WFP07] unterscheiden zwei generelle Ansätze zur Performanzanalyse von Software. Diese Klassifikation ist im Allgemeinen akzeptiert. Es wird unterschieden

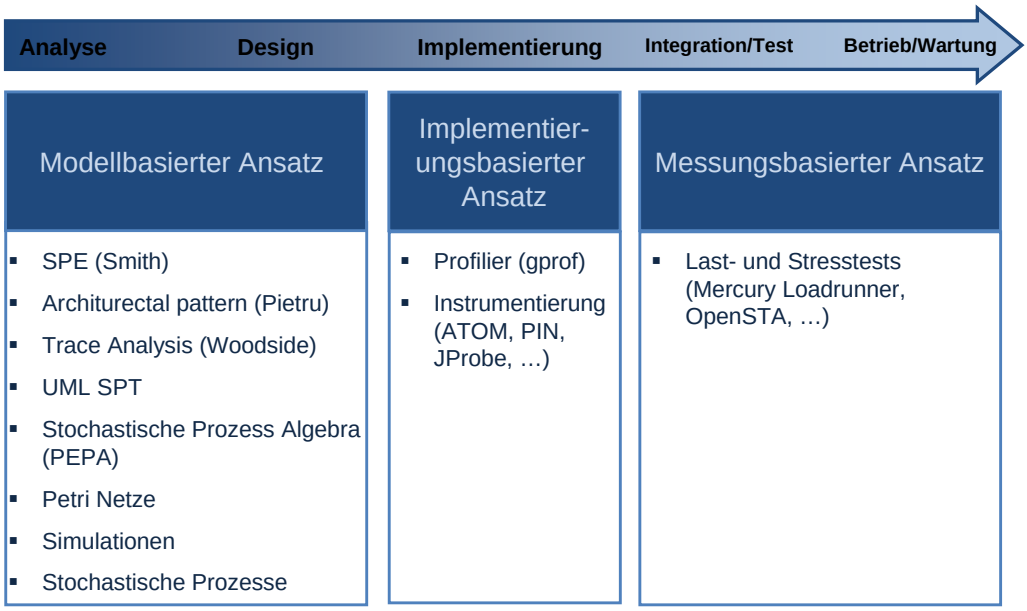


Abbildung 3.1: Stand der Wissenschaft und Technik: Je nach Phase im Softwarelebenszyklus werden unterschiedliche Methodologien der Performanzanalyse angewandt. Horizontal ist der *Software Life Cycle* abgebildet, die einzelnen Methodologien sind vertikal geordnet und anhand der Phasen kategorisiert. Die einzelnen Ansätze sind im Text weiter ausgeführt.

zwischen einem **modellbasierten**, frühen Ansatz im Softwareentwicklungsprozess und einem **messungsbasierten**, späten Ansatz (im Softwarelebenszyklus).

fließende Unterschiede

Die Unterschiede der einzelnen Methoden sind teilweise fließend. So integriert der modellbasierte Ansatz auch Messungen in späten Phasen des Softwaresystems um die Modelle zu validieren. Der messungsbasierte Ansatz geht teilweise auch von Modellen aus.

implementierungs-basierter Ansatz

Analog zum späten messungsbasierten Ansatz und zum frühen modellbasierten Ansatz gibt es auch Bestreben währen der Implementierung des Systems Performanzanalysen durchzuführen. Dieser **implementierungsbasierte** Ansatz stellt einen wichtigen Aspekt in der Performanzanalyse dar. Dieser Ansatz wird zu meist übersehen, weil er so offensichtlich ist. Dieser ist aber sogar der weit verbreiteste Ansatz, weil er die intuitivste Methode darstellt. Deshalb wird die klassische und generell akzeptierte Klassifizierung, die von Woodside et al. [WFP07] vorgestellt wird für diese Arbeit um diesen Ansatz erweitert und damit subklassifiziert.

3.2 Modellbasierte Ansätze

Früher Ansatz in der Entwicklung

Die grundlegende Idee für den modellbasierten Ansatz der Performanzanalyse ist naheliegend. Früh im Software Lebenszyklus, in der Design- und Analysephase eines Softwareprojekts werden die Artefakte aus der Entwicklung des Gesamtsystems benutzt um ein Modell zu erstellen. Dieses Modell wird während des ganzen Lebenszyklusses des Softwareprojekts zur Performanzanalyse genutzt.

Vorhersagen des Performanzverhaltens

Da dieses Performanz-Modell aus verschiedenen quantitativen und qualitativen Eigenschaften wie geschätzter Zeit, der Softwarearchitektur, Ressourcennutzung etc. basiert, sind unter anderem Vorhersagen bezüglich der zu erwarteten Performanz möglich (*Performance Predictions*), bevor das System erstellt wird.

Überblick

Balsamo et al. [Bal+04] geben einen exzellenten Überblick über modellbasierte Performanz Prognosen (*Performance Predictions*). In dieser Arbeit werden die unterschiedlichen Ansätze nach dem zugrundeliegenden Formalismus zur Modellierung gruppiert.

Subklassifikation der modellbasierten Methodologien

So subklassifizieren Balsamo u. a. [Bal+04] Methodologien zur Performanz Vorhersage basierend auf dem Software Performance Engineering Ansatz nach Smith [Poo92; SWo2] (*SPE Approach*), Methoden basierend auf Software Architektur Mustern (*Architectural Pattern-based*), Methodologien basierend auf einer Analyse von Trace Daten (*Trace-Analysis*) und der UML Erweiterung für Performance (*UML extensions for Performance*).
Anschließend werden mathematisch fundierte Methodologien zur Performanz Vorhersage, wie der Prozess Algebra (*Process Algebra*) und Ansätze basierend auf Petri-Netzen (*Petri-Net Approaches*) betrachtet. Abgeschlossen werden mit Methodologien basierend auf Simulationen (*simulation*) und stochastischen Prozessen (*stochastic processes*).

Vollständiger und profunder Überblick

Balsamo u. a. [Bal+04] präsentieren eine äußert profunde, aktuelle und vollständige Kategorisierung der existierenden Ansätze zur modellbasierten Performanzanalyse, deshalb wird diese Kategorisierung für diese Arbeit übernommen und weitgehend, insbesondere durch aktuelle Ansätze, ergänzt.

3.2.1 SPE - Software performance engineering

Smith als Pionierin der *SPE*

Die Pionierin dieses Ansatzes, der hauptsächlich mit *SPE* abgekürzt wird, ist *Connie U. Smith* [Poo92; SWo2; Smi90; BDo6]. Sie ist auch bei aktuellen Erweiterungen und Ergänzungen des Ansatzes (z. B. *PMIF* als Austauschformat für unterschiedliche Werkzeuge) beteiligt [MS10; SLP10a; SLP09; SLP10b].

In diesem Ansatz werden zwei seperate Modelle benutzt:

software execution model
: Das Ausführungsmodell der Software repräsentiert ihr dynamisches Verhalten.

system execution model
: Ein Modell für die Zielplattform inklusive Hard- und Softwarekomponenten.

Die Analyse des *software execution model* plus der Hardwareeigenschaften sind die Eingabewerte für das *system execution model*, welches dann das ganze System repräsentiert.

3.2.2 *Software Architektur Muster basierte Methoden – Architectural pattern-based methods*

Architectural Patterns

Aus der Architektur eines Systems wird mittels von Architekturmustern (*architectural patterns*) ein Performanzmodell abgeleitet, wie beispielsweise von Petriu et al. [PW00; PS02; PW05; TP10; Woo+09] demonstriert wird.

3.2.3 *Analyse von Trace Daten - Trace Analysis*

Trace Analysis

Aus einem dynamischen Modell des Systems (wie beispielsweise Nachrichten-Reihenfolge-Diagrammen (*message sequence charts* [BS01])) wird ein Modell abgeleitet.
Bedeutende Beiträge in diesem Gebiet wurden von *Woodside* geleistet [PW02; Woo+01].
Insbesondere haben diese Arbeiten nachhaltig aktuelle Werkzeuge (wie beispielsweise *Scalasca*) beeinflusst [Wyl10; Gei+10; Bec10a; MWW10].

3.2.4 *UML SPT Profil*

UML Unterstützung

Ein sehr vielversprechender Ansatz für eine strukturierte Performanz Entwicklung ist das UML Profile für *Scheduling, Performance and Time* [The05]. Mittels diesem UML Profils ist es möglich, Aspekte bezüglich der Performanz Vorherzusagen und zu analysieren [Ger+01; GK09; SBL08].

3.2.5 *Stochastische Prozess Algebra - Stochastic Process Algebra*

PEPA

Mittels der stochastischen Prozess Algebra können nicht nur funktionale sondern auch Charakteristiken bzgl. der Performanz analysiert werden [Hil96; Tri10]. Ein gutes Beispiel hierfür ist die PEPA Workbench [GH94; Gil+04] oder das Pepa Eclipse Plugin [TDG09; Tri07].

3.2.6 *Petri Netze - Petri-Nets*

Petri Netze

Ähnlich der stochastischen Prozessalgebra können Petri Netze auch dazu benutzt werden, Modelle zur Vorhersage der Performanz von Systemen zu erstellen.
Einen methodischen und modernen Ansatz dazu gibt López-Grao et al. [LGMC04]. Praxisrelevant dargestellt ist es in der Arbeit von Rana und Shields [RS00]. Auch aktuelle

Arbeiten nutzen immer noch die von *Carl Adam Petri* in den 1960er Jahren eingeführten Petri Netze zur Modellierung performanzrelevanter Vorgänge (z. B. [DSP10; PPM10; Hap+10]).

3.2.7 *Simulationen - Simulation Methods*

Simulationen

Aus UML Diagrammen werden mittels *Simulations*-Paketen Modelle erstellt. Diese Modelle werden subsequent simuliert [Hen+04; Mig+00; AS00; CMG08; AP10].

3.2.8 *Stochastische Prozesse - Stochastic Processes*

Semi-Markov Prozesse

Bei diesem Ansatz werden Semi-Markov Prozesse verwandt. Dazu werden sehr detaillierte Informationen auf Design Ebene benötigt. Ein Beispiel geben Lindemann et al. [Lin+02] und Berardinelli, Cortellessa und Marco [BCM10].

3.2.9 *Weitere Ansätze und Evaluation modellbasierter Ansätze*

Mathematische Formalismen und Werkzeuge

Bolch u. a. [Bol+06] geben einen guten Überblick über die benutzten mathematischen Modellierungswerkzeuge und -formalismen, im speziellen *queing networks*, *layered queues* und unterschiedlichen Klassen von Petri Netzen.

Mischformen und andere Ansätze

Da die Performanzanalyse bzw. die Erstellung ausreichend performanter Software ein wichtiger Aspekt und ein breites Forschungsgebiet ist, gibt es hier neue, völlig unterschiedliche Methoden bzw. Mischformen aus den bestehenden Ansätzen.

Palladio Komponentenmodell Framework für komponentenbasierte Softwareentwicklung

Reussner et al. implementierten z. B. mit dem *Palladio* Komponentenmodell ein zeitgenössisches und akutelles Framework zur Vorhersage von Performanz für komponentenbasierter Softwareentwicklung [Reu+07; KR08; BKR09; Bec10b; Bec+10].

Eine exzellente Evaluation der unterschiedlichen, modellbasierten Performanzanalysen gibt Koziolek [Koz04], deshalb wird im Rahmen dieser Arbeit darauf verzichtet.

3.3 *Implementierungsbasierte Ansätze zur Performanzanalyse*

Refaktorisierung und Reimplementierung

Refaktorisierung beziehungsweise die Reimplementierung von spezifischen Modulen eines Programms um ihre Performanz zu steigern ist auch eine Performanzanalyse bzw. Performanzoptimierung. Dies wird Tuning oder Feuerwehrperformanzanalyse genannt [SW02]. Diese wird aber zumeist bei der Kategorisierung in der wissenschaftlichen Literatur übersehen, da diese so naheliegend erscheint. Für diese Optimierung wird eher handwerkliche Geschicklichkeit des Programmierers benötigt was eine wissenschaftliche Forschung mit einem absoluten Ansatz obsolet macht.

Optimierung von Modulen

Hierbei werden der Code und die Algorithmen bestimmter Teile der Software verbessert, so dass das individuelle Modul oder Programmelement (*Unit*, Komponente, Paket, etc.) die erforderliche Performanz aufweist.

loop optimization

Dies wird durch Austausch und Veränderung des zugrundeliegenden Algorithmus erreicht oder beispielsweise durch eine Schleifenoptimierung (*loop optimization*) [Ove+05; BD06; Fow99; Wu+08].

Performanz bei Softwaretests

Insbesondere sollte jedes Modul bzw. jede Softwareeinheit unter Performanzaspekten getestet werden, wenn Unit- oder Modultests durchgeführt werden.

Performanzfokus auf Softwareeinheiten

In dieser Phase werden zu meist die gleichen Werkzeuge und Methoden wie im messungsbasierten Ansatz verwandt. Der Fokus dieser Phase liegt jedoch auf einzelnen Modulen und Softwareeinheiten – das System als Ganzes wird nicht betrachtet und ist zu meist auch **noch nicht** als Gesamtsystem integriert.

3.3.1 *Profiling zur Performanzanalyse*

historisch erster Lösungsansatz

Profiler sind historisch der erste Lösungsansatz zur Performanzanalyse und Performanzoptimierung. Die allerersten Ansätze zum Profiling waren zu meist vom Programmier codiert und nutzten *Timer Interrupts* um eine Probe (*Sampling*) von *Code Hot Spots* zu erstellen. Code wurde manuell instrumentiert unter der Zuhilfenahme von den in den Rechnern vorhandenen Timern.

eigenständige Applikationen

Profiler als eigenständige Applikationen kamen in den späten Siebzigern auf. Ein gutes Beispiel hierfür ist der bekannte Profiler *prof* [Ker+79], der Name *prof* wurde als eine Abkürzung für Profiler gewählt.

Erste Veröffentlichung

1982 wurde die erste wissenschaftliche Arbeit die Profiler behandelte publiziert, hier wurde das Design und die Verwendung des Gnu Profilers gprof beschrieben [Pet82].

Call Graph Profiling

gprof ist ein *call graph profiler*, also ein Profiler der wichtige Informationen aus dem Aufrufbaum integriert. Während der Ausführung der analysierten Anwendung werden wichtige Daten wie Aufrufhäufigkeit (*call counts*), Dauer des Aufrufs (*call execution time*) und Aufrufsequenzen (*call sequences*) gesammelt und nach dem Lauf in einem dynamischen Graphen visualisiert [Pet82]. Somit können Bereiche für eine Optimierung identifiziert werden. Profiling wird als hauptsächliches und wichtigstes Optimierungswerkzeug genutzt [Alt+10; Con+09]. Die Methode ist jedoch kaum Gegenstand aktueller Forschung, eine Ausnahmen stellt [DSZ10] als Kombination mit der Virtualisierungstechnologie dar.

3.3.2 *Instrumentierung zur Performanzanalyse*

Instrumentierung von Systemen

Der erste Ansatz zur Instrumentierung von Systemen wurde mit *ATOM* präsentiert [SE94a]. Beide Arbeiten (gprof [Pet82] und ATOM [SE94b] wurden 2004 vom PLDI

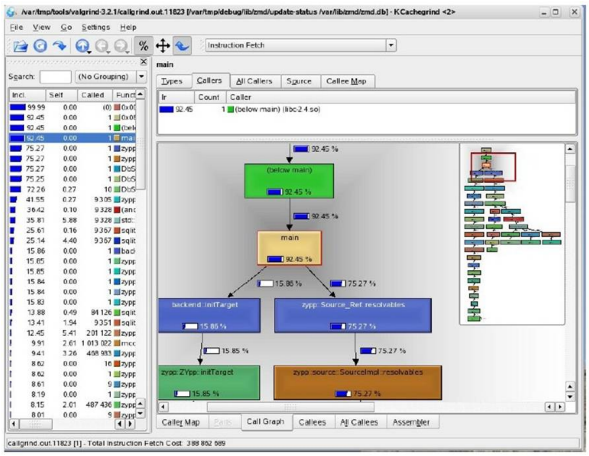


Abbildung 3.2: Screenshot von Valgrind, zitiert von [Bra06, S. 4]. Valgrind ist ein Tool zur Analyse von Programmen. Insbesondere eignet es sich zum Eingrenzen und Analysieren bei schlechter Performanz. Es stellt ein Werkzeug für den implementierungsbasierten Ansatz dar.

(*„ACM SIGPLAN Conference on Programming Language Design and Implementation“*) als eine der zwanzig einflussreichsten Arbeiten honoriert [McK04].

Instrumentierung als Erweiterung

Instrumentierung von Code ist eine Erweiterung von Profiling, zusätzliche Anweisungen werden in den Quellcode integriert. Um ein System zu profilen geben diese zusätzlichen Anweisungen beispielsweise die aktuellen, internen Timerwerte aus. In dieser Arbeit wird eine Instrumentierung zum Zwecke einer Performanzanalyse, wie im Allgemeinen Jargon üblich, als „Profiling“ bezeichnet.

Dynamische Injektion

Ein wichtiger Ansatz zur Instrumentierung von Systemen ist zum Beispiel *PIN* als das Standardwerkzeug in der Lehre und Ausbildung. *PIN* benutzt einen neuartigen Ansatz, der auf dynamischer Injektion von Code (*dynamic injection*) beruht. Mittels der dynamischen Injektion kann die Quellcodeveränderung extern und unkompliziert erfolgen. Damit ist insbesondere ein *„application steering“* [Luk+05] möglich. Während der Laufzeit können softwarebedingte Modifikationen am System durchgeführt werden, so dass das Programm unterschiedliche Ausführungspfade durchläuft. Insbesondere ergibt dies eine andere Ressourcennutzung.

application steering

So kann ein Entwickler oder Systemanalyst wichtige Informationen ohne langwierige Reimplementierung und erneute Compilierung des Systems bekommen. Wichtige Werkzeuge in diesem Gebiet sind JProbe [Sof07] und DynInst. Sarkar und Mukherjee [SM10] evaluiert Spezialwerkzeuge für das Tuning von verteilten Anwendungen.

Exkurs 3.3.1 Tuning von Szenario 11 – industrielles Steuerungssystem

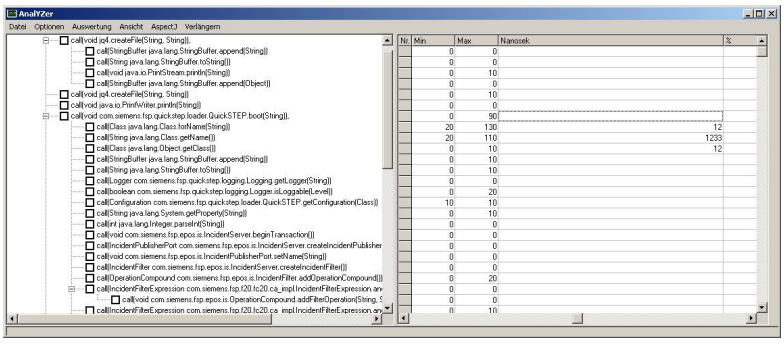


Abbildung 3.3: AnalYzer mit dem Aufrufbaum und der korrelierenden Liste zur Darstellung des Traces. Hier in der Abbildung kann im protokollierten Programmlaufs von Szenario 11 nach minimaler und maximaler Methodenlaufzeit und den benötigten Nanosekunden protokollierten Programmlaufs von Szenario 11 sortiert werden [Mano6].

Im Rahmen eines Fortgeschrittenenpraktikums (FoPra) sollte 2006 bei der *Siemens CT SE 1*, vom Autor betreut von Dr. Michael Pönitsch, bei Szenario 11, dem industriellen Steuerungssystem, **das Auswirken einzelner Komponenten auf die Gesamtperformanz** untersucht werden. Es war jedoch keine Dokumentation vorhanden, das Projekt groß (6,5 MB an *Jarfiles*) und der gesamte Code nicht zugänglich. Dadurch war kein Profiling bzw. ein *Code-Review* möglich. Mittels des inzwischen nicht mehr gepflegten *Eclipse Profiler Plugins* wurde versucht die Performanzprobleme der Anwendung aufzufinden [Mano6].
Da die Anwendung stark nebenläufig war, waren die einzelnen Interaktionen der Threads nur sehr schwer nachzuvollziehen, was eines der Hauptprobleme beim Profiling nebenläufiger Anwendungen ist (siehe Diskussion im Abschnitt 3.5.2 auf Seite 53). Deswegen wurde hierzu die Anwendung mittels AspectJ für ein Tracing instrumentiert und zwei Anwendungen geschrieben, erst AnalYzer in Visual Basic und später LogView in Java, mit denen versucht wurde, die protokollierten Daten (das Trace) besser zu gruppieren, um so das Zusammenwirken der Komponenten zu verstehen [Mano6].
In Abbildung 3.3.1 ist das Tool abgebildet. Links ist der Call-Tree als TreeView-Element implementiert, rechts eine Liste mit der nach dem Methodennamen, dem Thread, der Aufrufhäufigkeit, der minimalen und maximalen Dauer der Methodenausführung und vielem mehr sortiert werden konnte. Die Daten konnten im Tool mittels des Excel Diagram Steuerelements als Diagramme angezeigt werden (siehe Abbildung 3.3.1) [Mano6].
Jedoch brachten auch diese Gruppierungen keinen Erkenntnisgewinn. Deswegen wurde das Tool LogView konzipiert, das den Trace ähnlich einem Sequenzdiagramm darstellt. Wie in Abbildung 3.3.1 gut zu sehen ist, wird der Aktivierungsbalken im „Sequenzdiagramm“ proportional zur Zeit der Methodenausführung dargestellt, die Aktivierungsbalken sind spezifisch nach dem Thread des Objekts in bestimmten Farben eingefärbt. [Mano6]

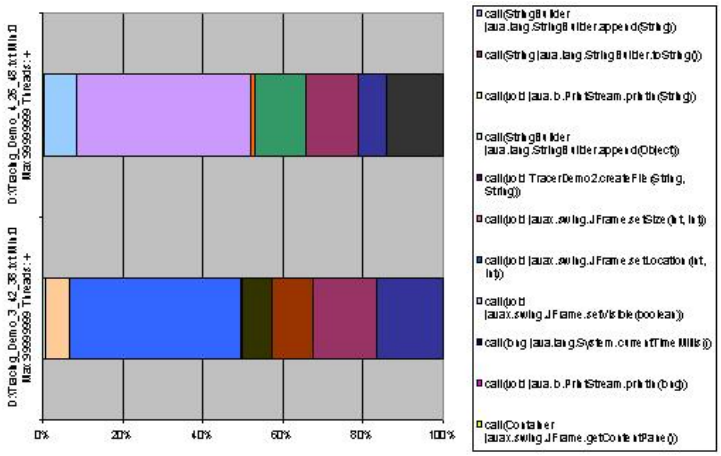


Abbildung 3.4: Univariates Auswertung eines protokollierten Programmlaufs von Szenario 11 (industrielles Steuerungssystem) im Tool AnalYzer

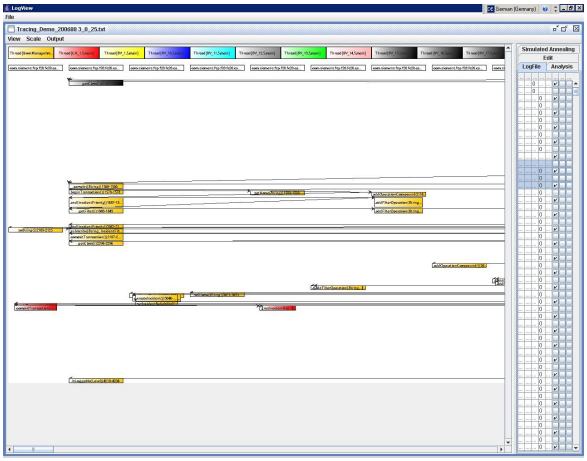


Abbildung 3.5: LogView: Darstellung eines protokollierten Programmlaufs von Szenario 11 als Sequenzdiagramm. Auch diese Ansicht ist, durch die komplexe Ausführung von Szenario 11, unübersichtlich. Ohne ausreichende Expertise können keine Zusammenhänge, Wechselwirkungen beziehungsweise Optimierungspotenzial und -kandidaten erkannt werden.

Abbildung 3.3.1 zeigt ein Trace der Ausführung von Szenario 11. Aber auch hier sind die Ausführungen so komplex, dass, ohne ausreichende Expertise (z. B. durch eine Beteiligung bei der Entwicklung des Systems), keine Zusammenhänge erkannt werden konnten.

3.4 Messungsbasierte Ansätze zur Performanzanalyse

Last- und Streßtests

Am Ende des Lebenszyklusses von Software werden Werkzeuge und Methoden für Last- und Stresstests (load- and stresstests) und Monitoring Tools eingesetzt. Der Fokus liegt hier im Verifizieren oder Testen ob das System die geforderte Performanz und demnach die Spezifikation des Systems (falls gegeben) erfüllt bzw. nutzbar ist.

industrielles Umfeld

Messungsbasierte Ansätze werden vor allem im industriellen Umfeld genutzt, während modellierungsbasierte Ansätze vor allem im akademischen Bereich großen Anklang finden.

3.4.0.1 Last- und Stresstests – Load and stress tests

Scott Barber beschreibt in einer Serie von Artikeln den momentanen Stand der Technik im Testen der Performanz im industriellen Umfeld [Bar04b; Bar04a].

automatische Lastgenerierung

Erwähnenswert ist der Ansatz, sich automatisch Last (*workload*) für die zu testende Applikation generieren zu lassen [Kri06]. Analog soll das Nutzerverhalten mit stochastischen Mitteln modelliert werden [Dra+06].

Toolunterstützung

Hierzu ist ein breites Spektrum von Applikationen verfügbar, wie die von *HP Software & Solutions* (ehemals *Mercury*, 2006 von *Hewlett-Packard* übernommen) im industriellen Umfeld führenden Loadgeneratoren *Loadrunner* [Cor04] und *Winrunner* [San+07] bis hin zu Open Source Projekten wie *OpenSTA* [Mun+02]. Das Thema hat eher eine geringe Betrachtung in der Wissenschaft [ZFL10].

Lösung von Performanzproblemen

Abhängig vom System (ob im embedded Umfeld oder Webserver (vergleiche z. B. [MA00; MA98; Sav01]) gibt es unterschiedliche Ansätze wie das Problem unzureichender Performanz gelöst wird.

KIWI – kill it with iron Ansatz

So kann das Problem beispielsweise bei Server Applikationen durch ein Upgrade mit performanterer Hardware gelöst werden. Dieser Ansatz wird oft humoristisch als „kill it with iron“ oder abgekürzt als der *KIWI-Approach* bezeichnet [Theo7, S. 15]. So wird der Effekt – unzureichende Performanz – durch das Aufrüsten mit Hardware beseitigt, nicht jedoch die eigentliche Ursache der Performanzprobleme.

Exkurs 3.4.1 Probleme bei Hardwareskalierung

Probleme mit Hardwareskalierung

Dieser Ansatz (kill it with iron) funktioniert demnach nur in bestimmten Bereichen, in denen Hardware ausgetauscht werden kann. So sind in bestimmten Industriesparten die Stückzahlen extrem hoch und die Gewinnspanne klein (z. B. Automobilbranche) oder leistungsfähigere Hardware kann durch andere Nebeneffekte nicht verbaut werden (wie z. B. notwendige, zusätzliche Kühlung durch leistungsfähigere Hardware).

3.5 Evaluation des Stands der Technik und Wissenschaft

Alle drei Herangehensweisen zur Software Performanzanalyse, namentlich der modellbasierte, der implementierungsbasierte und der messungsbasierte Ansatz, sind unzureichend und haben Mängel.

Nachteile des Status Quo

3.5.1 Evaluation des modellbasierten Ansatzes

Der bereits früh im Lebenszyklus der Software ansetzende modellbasierte Ansatz ist extrem wichtig, primär um harte Anforderungen an die Performanz und die Leistungsziele des System zu erreichen (vgl. z. B. [AK+08]).

Modellbasierte Ansatz

Viele Performanzprobleme liegen inhärent in der Architektur eines Systems. Viele Entwickler haben eine sogenannte „fix-it-later"-Einstellung [Smi90] – dabei wird erst das System implementiert, etwaige Performanzprobleme werden später gelöst. Dieses wird von Smith [Smi90] als „Tuning" bezeichnet. Zusätzliche Kosten für frühe Performanzanalyse für typische Softwaresysteme beziffert Smith [Smi90] mit 1 bis 3% der gesamten Projektkosten und argumentieren, dass diese geringer sind als die Kosten für ein Tuning.

fix-it-later

Tuning

❑ **Die Vorteile des modellbasierten Ansatzes** liegen klar in der Integration von Performanzabhängigkeiten und -kopplungen.

❑ **Die Nachteile des modellbasierten Ansatzes** sind jedoch:

❑ **Modelle sind teuer, zeitintensiv und benötigen Expertise**

- ❑ Um ein Modell zu erstellen werden Experten benötigt.
- ❑ Ausreichendes Wissen über Anwendungsfälle (*Use Cases*) und Szenarien der Software muss vorhanden sein.
- ❑ Die spezifische Nutzung einer Software unterscheidet sich von Anwender zu Anwender, so dass es zu einer Diskrepanz zwischen intendierten und tatsächlichen Anwendungsfällen kommen kann.
- ❑ Bei Wiederverwendung von Code kann zur betreffenden Bibliothek zu meist kein Modell gebaut werden.
- ❑ Die Modelle können fehlerhaft und ungenau sein.

❑ **Modelle sind Annäherungen**

- ❑ Wichtige Details, wie unterschiedliche Hardwareumgebungen, müssen abstrahiert werden, um überhaupt ein Modell erstellen zu können.
- ❑ So tendieren Modelle dazu, durch Schätzungen und Abstraktion, unpräzise zu sein [KF05]. Dies kann dazu führen, dass die Prognosen unzutreffend

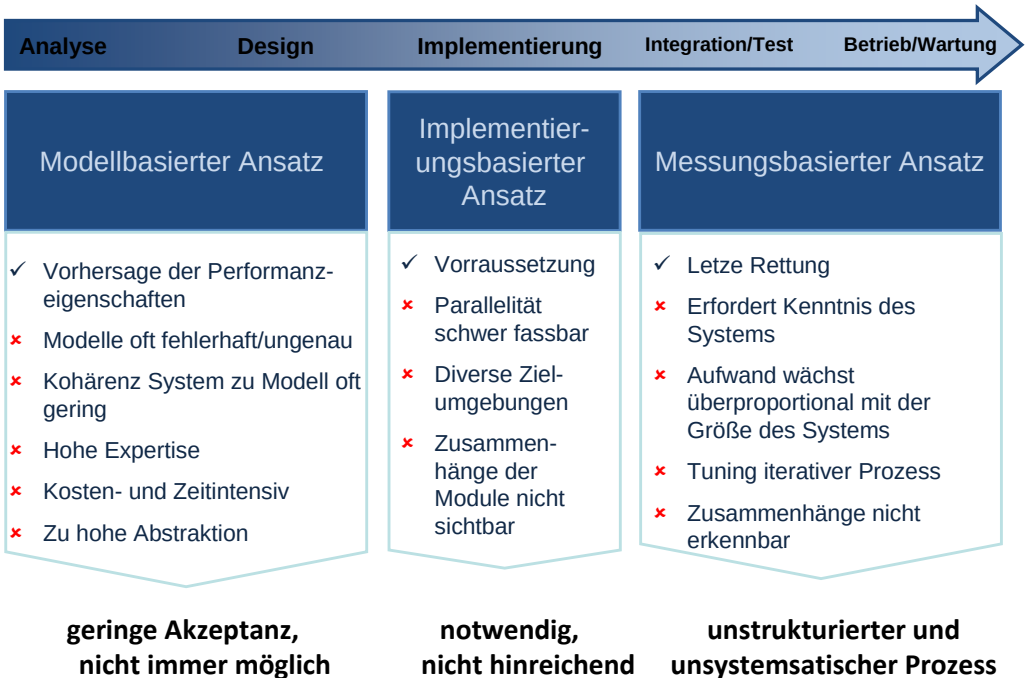


Abbildung 3.6: Evaluation der Methodenansätze:
❑ Horizontal ist der *Software Life Cycle* abgebildet.
❑ Die Vor- und Nachteile der Ansätze sind vertikal angeordnet.
❑ Ein Haken (✓) identifiziert einen Vorteil.
❑ Ein rotes „✘“ zeigt einen Nachteil an.
❑ Die Beschriftung am Fuß der einzelnen Balken fasst ein entsprechendes Fazit zur Performanzanalyse der Methodenansätze zusammen.

sind. Nicht selten führt dies zu Unterschieden von prognostiziertem zu realem Performanzverhalten des Systems.

- Softwaresysteme entwickeln sich weiter, jedoch kann dies dazu führen, dass die zu Beginn getätigten Annahmen nicht mehr gelten. Zu meist werden die entsprechenden Performanzmodelle nicht aktualisiert.

- **Unzureichendes Wissen über die Komponenten (bei der Modellbildung)**
  Um qualitative bessere Software in kürzeren Zeitperioden zu erstellen, wird Software nicht (mehr) monolithisch erstellt. Das Rad soll nicht ständige neu erfunden werden. Anstelle dessen soll Software zu einem hohen Grad modular erstellt werden. Dabei werden vorhandene Komponenten und Bibliotheken umfangreich benutzt. Diese Softwarebauteile sind zu meist von Drittanbietern und es sind keine detaillierte Informationen über die jeweiligen Internas verfügbar.

3.5.2 *Evaluation des implementierungsbasierten Ansatzes*

Das Tuning und die Optimierung auf Modulebene sind alleine nicht ausreichend, da Abhängigkeiten und Performanzkopplungen mit anderen Modulen nicht betrachtet werden können.

- **Performante Module sind notwendig aber nicht hinreichend**
  Die Analyseperspektive liegt auf einem singulären Modul, dies ist für eine ausreichende Performanz des Systems notwendig, aber nicht hinreichend, da Abhängigkeiten von anderen Komponenten des Systems (z. B. durch Aufrufe, Wartezeiten uvm.) beim Profiling nicht betrachtet werden.
- **Parallele Interaktion, parallele Ressourcennutzung**
  Vor allem die wachsende Parallelität von Systemen verschärft durch die gleichzeitige Ressourcennutzung das Problem der Abhängigkeiten. Insbesondere diese Parallelität ist beim Profiling schwer wahrnehmbar.
- **Wiederverwendung von Code**
  Wiederverwendung von Code (z. B. mittels Komponenten, Paketen, uvm.) wird propagiert, jedoch liegt dieser Code teilweise nicht vor, weil er beispielsweise von Drittanbietern erworben wurde. Zusätzlich ist fremder Code beim Profiling oft schwer zu verstehen [Sta84; WS00].

notwendig, nicht hinreichend

Ausreichend performante Module, Komponenten und Softwareeinheiten jeder Granularität (z. B. Module, Prozeduren, Funktionen, Pakete, Komponenten, etc.) sind eine notwendige Bedingung aber alleine nicht hinreichend für ein ausreichend performantes Gesamtsystem, da Kopplungen und Abhängigkeiten im System, die die Gesamtperformanz beeinträchtigen, nicht untersucht werden können. Teilweise sind Softwareeinhei-

ten in dieser Phase nur partiell implementiert. Die Nutzung von Ressourcen wie der Hardware wird hierbei nicht beachtet.

Nebenläufige Beeinflussung der Performanz

Dies ist ein besonderes Problem mit den sich weitgehenden etablierenden und geforderten parallel arbeitenden Systemen. Da hier Code parallel ausgeführt wird können die Seiteneffekte einzelner Soft- und Hardwareeinheiten auf die Performanz nur schwer betrachtet werden, der Untersuchungsfokus liegt auf der sequentiellen Abarbeitung der analysierten Softwareeinheit.

3.5.3 *Evaluation des messungsbasierten Ansatzes*

Performanz ausreichend

Messungsbasierte Ansätze zur Performanzanalyse zeigen ob die Performanz eines Systems adäquat ist, aber nicht die unterliegende Ursache von Performanzproblemen und Performanzabhängigkeiten der einzelnen Komponenten. Die Ausgabe besteht nur aus absoluten Daten, die gegenseitige Beeinflussung wie aus den Modellen aus dem messungsbasierten Ansatz kann nicht eruiert werden.

Lösung des Performanzproblems

Zur eigentlichen Behebung des Performanzproblems wird nun der *KIWI-Approach* angewandt, Hardware wird also unverhältnismäßig skaliert (was weitere, iterative Last- und Streßtests erfordert), oder aus früheren Phasen des Softwarelebenszklus müssen zur Analyse (modell- oder implementierungsbasierte) Methoden verwandt werden, um entsprechende weitere Aktionen (z. B. Refaktorisierung und Redesign) einleiten zu können.

3.6 *Performanzanalysen in der Praxis*

Performanzanalysen in der Praxis

Zur Verbreitung von Performanzanalysen in der Praxis (genauer: Industrie) zitieren Woodside et al. in ihrer Studie „*The future of Software Performance Engineering*“ [WFP07] ein im Jahre 2006 in Auftrage gegebenes Gutachten des US-amerikanischen Softwareunternehmens *Compuware*. Diese im Juli 2006 durchgeführte Online Befragung von *Forrester Consulting* an *Senior IT Managern* aus großen europäischen Organisationen ergab aus den 150 Antworten folgendes Ergebnis [Com06]:

3.6.1 *Regelfall Performanzproblem*

- Mehr als die Hälfte aller befragten Organisationen gaben an, dass 20% oder mehr ihrer ausgelieferten Applikationen unerwartete Performanzprobleme aufwiesen [Com06, S. 2].
- 52% aller befragten Manager gaben an, dass sie am Tage der Auslieferung der Applikation mit einer Wahrscheinlichkeit höher als 50% Performanzprobleme erwarten [Com06, S. 3, Frage 5] (siehe Abbildung 3.7).

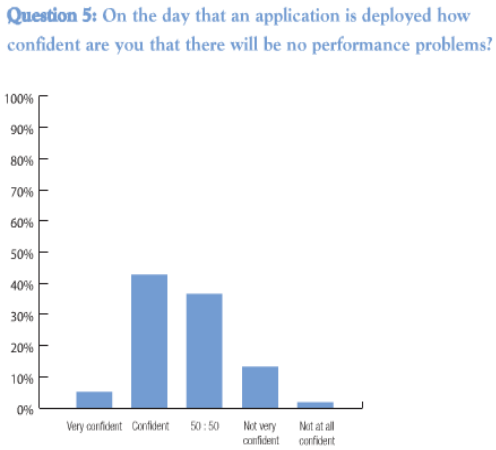


Abbildung 3.7: Mehr als die Hälfte aller befragten Manager gaben an, dass sie in mehr als 50% der Fälle Performanzprobleme bei ausgelieferten Applikationen erwarten. Quelle: zitiert von „*Compuware, Applied Performance Management Survey*“, Oct. 2006 [Como6, S. 3, Frage 5].

- 71% aller befragten Organisationen gaben an, dass Performanz Probleme vom *End User* gefunden werden, wenn dieser den *Help Desk* kontaktiert [Como6, S. 1].

Performanzprobleme sind im industriellen Umfeld der Softwareentwicklung, nach den Ergebnissen dieser Studie, eher die Regel als die Ausnahme.

3.6.2 *Akzeptanz des modellbasierten Ansatzes*

- Nur 14% aller Antworten (Frage 1) konstatieren, dass sie konsistent einen modellbasierten Ansatz zur Performanzanalyse nutzen [Como6, S. 2, Frage 1] (siehe Abbildung 3.8).
- 30% aller Firmen benutzen selektiv einen modellbasierten Ansatz zur Performanzanalyse [Como6, S. 2, Frage 1] (siehe Abbildung 3.8).

modellbasierte Ansätze in der Industrie

Modellbasierte Ansätze zur Performanzanalyse sind im industriellen Umfeld nicht generell akzeptiert. Dies ist auf die diskutierten Nachteile des modellbasierten Ansatzes[1] zurückzuführen.

[1] siehe Abschnitt 3.5.1, Seite 51

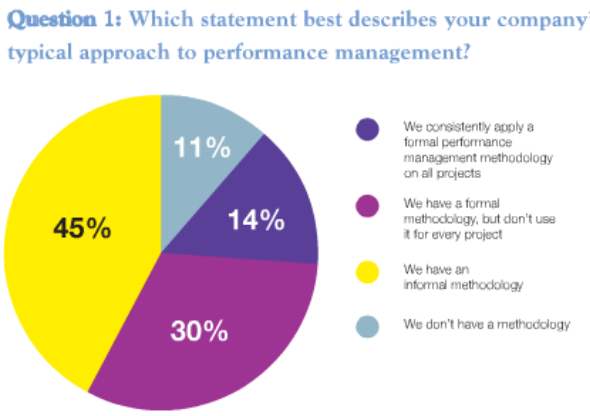


Abbildung 3.8: Nur ein geringer Bruchteil der Antworten des Gutachtens konstatiert, dass formale Methoden zur Performanzanalyse konsistent im Entwicklungsprozess genutzt werden. Quelle: zitiert von „*Compuware, Applied Performance Management Survey*“, Oct. 2006 [Com06, S. 2, Frage 1].

3.6.3 *Akzeptanz des messungsbasierten Ansatzes*

- Ungefähr die Hälfte aller Organisationen wartet bis Performanz Probleme beim Testen oder in der Produktion auftauchen bevor sie Daten des Systems messen [Com06, S. 4, Frage 9]. 8% aller Organisationen sammeln überhaupt keine Daten [Com06, S. 4, Frage 9].
- 47% der Befragten gaben an, dass Endnutzer über unzureichende Performanz monieren, auch wenn messungsbasierte Ansätze (mittels *monitoring tools*) ausreichende Performanz bestätigen [Ven+09].

messungsbasierte Ansätze in der Industrie

Messungsbasierte Ansätze zur Performanzanalyse werden im industriellen Umfeld, wie die modellbasierten Ansätze, nicht generell eingesetzt.[2] Zusätzlich zeigt das Ergebnis der Studie, dass immer noch, trotz eines messungsbasierten Ansatzes Performanzprobleme auftreten können.

Die diskutierten Nachteile des messungsbasierten Ansatzes[3] zeigen die Ursachen hierfür auf.

[2] siehe Abschnitt 3.5.1, Seite 51
[3] siehe Abschnitt 3.5.3 auf Seite 54

3.6.4 *Akzeptanz des implementierungsbasierten Ansatzes*

- 56% aller Organisationen gaben an, dass sie Profiling als kritischen Schritt sehen [Como6, S. 3, Frage 7a].
- Diese 56% gaben des Weiteren an, dass vom Profiling die Performanz in über 80% der Fällen profitiert [Como6, S. 3, Frage 7b].

Profiling in der Industrie

Diese Frage 7a bestätigt den hohen Stellenwert des implementierungsbasierten Ansatzes für die gute Performanz von Applikationen bzw. Systemen. Profiling ist und bleibt, trotz aller elaborierten modellbasierten Ansätze, essentiell zur Performanzanalyse von Systemen. In Abschnitt 3.5.2 (Seite 53) wurden die Nachteile des implementierungsbasierten Ansatzes diskutiert.

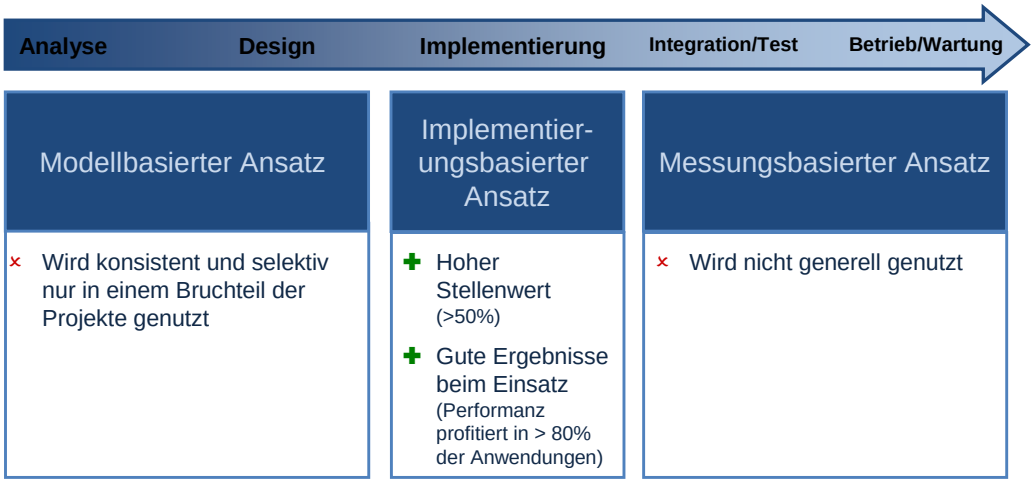


Abbildung 3.9: Akzeptanz der unterschiedlichen Ansätze in der SW Industrie nach [Ven+09; Como6]. Horizontal ist der *Software Life Cycle* abgebildet, eine kurze Bilanz zur Akzeptanz ist vertikal angeordnet. Ein grünes „✚“ identifiziert positive Akzeptanz, ein rotes „✘“ schlechte Annahme des Methodenansatzes in der Industrie.

3.6.5 *Herausforderung wachsende Komplexität von Systemen*

Herausforderung wachsende Komplexität

58% aller Antworten identifizieren die wachsende Komplexität bei Systemen als hauptsächliche Herausforderung bei der Performanzanalyse. Ein Problem, welches nach der Prognose von *Forrester Consulting*, durch den Einsatz von Service-orientierten Architekturen (SOA) und Virtualisierungstechnologien weiter zunehmen wird [Ven+09].

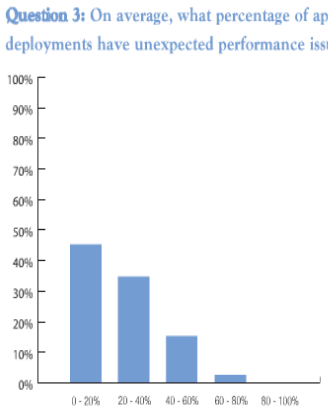


Abbildung 3.10: Beim Enduser treten oft unerwartete Performanzprobleme auf. Quelle: zitiert von „*Compuware, Applied Performance Management Survey*“, Oct. 2006 [Como6, Frage 3].

Parallelität, Kopplung

Hier ist vor allem die steigende Parallelität sowie die zunehmenden Kopplungen mit anderen Systemen und Subsytemen zu nennen, die die Komplexität und die Abhängigkeit bezüglich der Performanz von anderen Komponenten erhöhen. Die Parallelität kann nur schwer überblickt werden und die Internas der gekoppelten Komponenten sind meist für eine Performanzanalyse des gesamten Systems nicht zugänglich.

3.6.6 Performanzprobleme beim Endnutzer

Hardwareheterogenität

Die Zielsysteme für Applikationen und Systeme unterscheiden sich drastisch. Durch die hohe Hardwareheterogenität wird Software auf einer anderen Hardwareplattform und damit mit unterschiedlichen Eigenschaften ausgeführt, die die Performanz des Systems beeinflussen können.

Einfluß der Hardware

Bislang ist keine Methodologie bekannt, die den Einfluß der Hardwarekomponenten auf die Performanz des Systems eruiert bzw. eine automatisierte Methode, die Mindestanforderungen an die Hardware bestimmt.
Compuware fordert deswegen ein proaktives, holistisches Vorgehen, schlägt aber ihre eigene modellbasierte Performanzanalyse vor.

3.6.7 *Zusammenfassung der Studie*

Zusammengefasst zeichnet die Studie [Com06; Ven+09] folgendes Bild:

- Performanzprobleme sind im industriellen Umfeld der Softwareentwicklung, nach den Ergebnissen dieser Studie, eher die Regel als die Ausnahme.
- Modellbasierte Ansätze zur Performanzanalyse werden im industriellen Umfeld nicht generell eingesetzt.
- Messungsbasierte Ansätze zur Performanzanalyse werden im industriellen Umfeld nicht generell eingesetzt.
- Zusätzlich zeigt das Ergebnis der Studie, dass trotz eines messungsbasierten Ansatzes Performanzprobleme auftreten können.

3.7 *Zusammenfassung, Beantwortung von Teilfragestellung β und Fazit*

bestehende Ansätze

Die nach dem Stand der Technik und Wissenschaft bestehenden Ansätze zur Performanzanalyse wurden in diesem Kapitel präsentiert. Die in der Wissenschaft und Industrie generell akzeptierte Klassifizierung mit dem frühen, modellbasierten Ansatz und dem späten, messungsbasierten Ansatz wurde übernommen und mit dem Bindeglied des implementierungsbasierten Ansatzes subklassifiziert. Abschnitt 3.2 kategorisiert, anhand der entsprechenden, zugrundeliegenden Vorgehensmethodik, die frühen modellbasierten Techniken. Abschnitt 3.3 zeigt die Techniken und Werkzeuge des implementierungsbasierten Ansatzes (Profiling und Instrumentierung). Abschnitt 3.4 beschreibt Programme und Methodiken mit denen der späte messungbasierte Ansatz via Last- und Stresstests bei den zu analysierenden IT-Anwendungen durchgeführt wird. Eine zitierte Erhebung validiert in Abschnitt 3.5, dass alle Ansätze in der Industrie nicht weit verbreitet sind. Als Konsequenz stellt unzureichende Performanz in IT-Applikationen eher den Regelfall als die Ausnahme dar. Die Zusammenfassung der Studie befindet sich in Abschnitt 3.6.7 auf Seite 59.

Ansätze im SW-Lifecycle

Performanzanalyse in der Praxis

Die betrachteten Ansätze zur Performanzanalyse basieren:

modellbasiert

❑ **modellbasiert** – auf einer Modellierung und subsequenten (mathematischen) Evaluation,

implementierungsbasiert

❑ **implementierungsbasiert** – auf der manuellen Identifikation und dem Refaktorisieren von (performanzmindernden) Codeelementen,

messungsbasiert

❑ **messungsbasiert** – auf dem Messen und dem künstlichen Erzeugen von Last- oder Stress im System, oder – genereller – das quantitative Variieren von Eingabewerten in einem Experiment.

Kein Ansatz benutzt ein Experiment mittels des hier vorgeschlagenen Analyseinstrumentariums (dem zielgerichteten Variieren von Zeitpunkten oder Laufzeitdauern der Komponenten eines Systems).

Teilfragestellung β ✔ Somit kann Teilfragestellung β positiv beantwortet werden:
Aktuell wird, nach dem Stand der Wissenschaft und Technik, kein Verfahren benutzt, das bei einem System oder teilweise implementierten System Wechselwirkungen mittels eines Experiments (basierend auf zeitlicher Variation einzelner Systemelemente) eruiert.

neuer Ansatz Die Vorgehensweise ist neu.

Kapitel 4

Analyseinstrumentarium

„We must use time as a tool, not as a couch."

John F. Kennedy

Übersicht des Kapitels

In Abschnitt 4.1 wird der für diese Arbeit gewählte Ansatz der empirischen Versuchsreihen präsentiert.
Abschnitt 4.2 zeigt die Vorteile des empirischen Ansatzes zur Performanzanalyse gegenüber den bestehenden Ansätzen, mit ihren jeweiligen Nachteilen, aus Kapitel 3 auf.
Abschnitt 4.3 definiert, bewußt abstrakt und ohne konkrete Realisation, das Analyseinstrumentarium und zeigt einzelnen Effekte, die damit realisiert werden können (Prolongation, Retardation und simuliertes Optimieren).

4.1 Empirischer Ansatz zur Performanzanalyse

Im Rahmen dieser Arbeit wird eine Methodologie erarbeitet, um die Auswirkungen der Performanz einzelner Komponenten auf andere Komponenten oder das System in einem Experiment zu eruieren. So können Optimierungspotenzial und -kandidaten in einem System gefunden werden. Es wird zwischen dem Analyseinstrumentarium bzw. den Analyseinstrumentarien, als Lösungsansatz, und der eigentlichen Analyse, dem Ziel dieser Arbeit unterschieden.

Werkzeug zeitliche Variation

❑ **Der Lösungsansatz – das Analyseinstrumentarium**

Lösungsansatz dieser Arbeit ist es, zum Zwecke einer Performanzanalyse, zielgerichtet die Abläufe einzelner Komponenten eines Systems zeitlich zu variieren. Das hier eingeführte Analyseinstrumentarium sollen als ein Werkzeug verstanden werden, das für unterschiedlichste Aufgaben eingesetzt werden kann.

❑ **Das Ziel – die Analyse**

Mit dem Analyseinstrumentarium als Lösungsansatz soll es möglich werden, nachfolgend aus gemessenen Reaktionen der Systemkomponenten bei einer Ausführung Rückschlüsse auf die Beschaffenheit des Systems zu ziehen. Insbesondere sollen Abhängigkeiten, Zusammenhänge und Wechselwirkungen der Systemkomponenten erkannt werden.

empirische Versuchsreihe

Dieses Vorgehen ähnelt einer empirischen Versuchsreihe. Bestimmte Eigenschaften, hier zeitliche Abläufe, eines Systems werden bei empirischen Versuchen kontrolliert verändert um Änderungen und Auswirkungen zu beobachten. Mittels des Analyseinstrumentariums werden zielgerichtet die Versuchsbedingungen verändert, die Ausführung des zeitlich variierten Systems entspricht einem Experiment, die Analyse ist eine Auswertung des protokollierten Versuchsergebnisses.

Versuchreihen in technischen Disziplinen

Experimente sind in der Informatik, die ihre Wurzeln aus der Mathematik hat, nicht weit verbreitet. Da die junge Informatik immer mehr Anleihen aus Ingenieurswissenschaften machen muss, um die immer größer werdende Komplexität von Systemen und die Anforderungen an diese zu bewältigen, sind Versuchsreihen, analog wie sie in den angewandten Disziplinen der Mechanik, Elektrotechnik u.ä. üblich sind, naheliegend [SDJ07; Per08; BSH86; Bas+05; RBS93; Agr92; Fuc92; Mar+10].

4.2 *Vorteile der empirischen Versuchsreihe gegenüber bestehenden Ansätzen*

Seiteneffekte

Der empirische Ansatz hat immense Vorteile – die hochkomplexe Umgebung kann mit zur Analyse einbezogen werden. So können Seiteneffekte zum Betriebssystem, parallel arbeitenden Programmen beziehungsweise zur Umgebung allgemein identifiziert werden.

Es ergeben sich dadurch zum Stand der Technik[1] der Performanzanalyse und der Evaluation zur Akzeptanz dieser [Com06; Ven+09][2] folgende Vorteile:

❑ **Vorteile gegenüber modellbasierten Ansätzen**

Die in dieser Arbeit vorgeschlagene Methodologie integriert, analog wie die modellbasierten Ansätze[3], verschiedene quantitative und qualitative Eigenschaften

[1] siehe Abschnitt 3, Seite 41

[2] siehe Abschnitt 3.6.7, Seite 59

[3] siehe Abschnitt 3.2, Seite 43

des Systems, z. B. die Hardwareumgebung, das Betriebssystem und parallele Abläufe. Diese werden jedoch durch empirische Experimente am laufenden System mit in die Analyse einbezogen, sie müssen also nicht a priori abgeschätzt werden, was zu Fehlern bzw. Ungenauigkeiten führen kann.

❑ **Vorteil als eine Erweiterung zu implementierungsbasierten Ansätzen**

Die in dieser Arbeit entwickelte Methode kann eine Erweiterung der implementierungsbasierten Ansätze[4] darstellen. Insbesondere können Profiler oder entsprechende Tools um die Analyseinstrumentarien ergänzt werden. Somit ist eine Vorselektion vor dem Profiling zur Performanzanalyse und -optimierung möglich. So können bei Performanzproblemen abhängige oder verursachende Komponenten, Module und Ursachen identifiziert werden – ohne dass die Internas der Ursachen bei der Vorselektion vorliegen müssen. Insbesondere ermöglicht der Ansatz die (automatisierte) Identifikation von Abhängigkeiten bei parallelen Abläufen[5], was mittels eines klassischen Profilings meist nur schwer bzw. sehr mühsam möglich ist.

❑ **Vorteile gegenüber messungsbasierten Ansätzen**

Messungsbasierten Ansätze[6] (Last- oder Stresstests) können auch als empirische Versuchsreihe gesehen werden. Hier werden die Eingabewerte variiert um Last zu generieren bzw. das System unter Stress zu setzen. Beispielsweise werden mit entsprechenden Tools mehr User simuliert, die auf eine Webapplikation zugreifen. Somit wird das zu erwartende Verhalten des Systems bei einer entsprechenden Last ausgemessen. Jedoch eignen sich messungsbasierte Ansätze nicht zur Identifikation von Bottlenecks bzw. Optimierungskandidaten.

❑ **Vorteile des Lösungsansatzes bei komplexen Systemen**

Insbesondere die immer komplexeren, teilweise integrierten und zusammenwachsenden Systeme stellen ein großes Problem bei der Performanzanalyse dar.[7] Durch die erwähnten automatisierbaren, empirischen Versuchsreihen wird die Komplexität des Systems automatisch in die Analyse mit integriert. Die Technik skaliert: so kann von höhergranularen Einheiten (z. B. Softwarepakete) in einem iterativen Prozess auf Optimierungspotenzial in einer niedergranularen Untermenge (z. B. Module oder Funktionen) geschlossen werden.

[4] siehe Abschnitt 3.3, Seite 45
[5] siehe Abschnitt 6, Seite 89
[6] siehe Abschnitt 3.4, Seite 50
[7] siehe Abschnitt 3.6.5, Seite 57

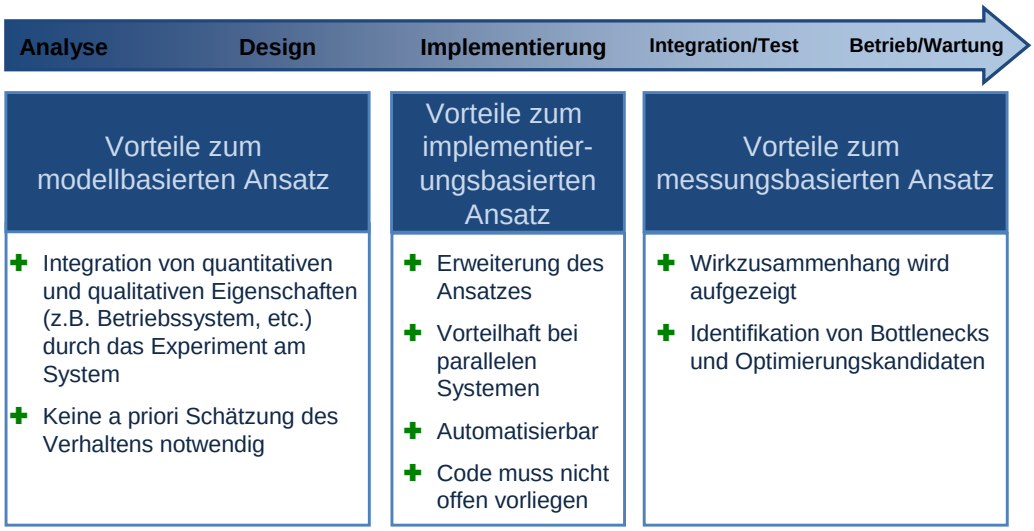


Abbildung 4.1: Vorteile des in dieser Arbeit entwickelten empirischen Ansatzes zur Performanzanalyse gegenüber den bestehenden Methodenansätzen.
Horizontal ist der *Software Life Cycle* abgebildet, die einzelnen Vorteile zu den Methodologien sind vertikal geordnet, mit einem grünen „✚“ gekennzeichnet und anhand der Phasen kategorisiert. Die einzelnen Vorteile sind im Text weiter ausgeführt.

4.3 *Analyseinstrumentarien*

Exkurs 4.3.1 Analyseinstrumentarium vs. Analyseinstrumentarien
Der Singular „Analyseinstrumentarium“ bezeichnet die Idee der zeitlichen Variation von Abläufen im Allgemeinen. Der Plural „Analyseinstrumentarien“ bezeichnet die verschiedenen Ausprägungen des Analyseinstrumentariums. So sind ein Verzögern oder ein Verlangsamen alles zeitliche Variationen von Abläufen, aber unterschiedliche Techniken bzw. Analysewerkzeuge, die auch unterschiedlich realisiert werden müssen.

automatisierbare Performanzoptimierung

Ziel der Performanzoptimierung ist, die Performanz eines Systems zu optimeren, es also „schneller“ (performanter) zu machen. Es ist jedoch meist umöglich bei einem System, bestehend aus Hardware und Software, eine Komponente durch eine performantere Komponente auszutauschen um die empirischen Experimente durchzuführen und die Änderungen und Auswirkungen zu beobachten.

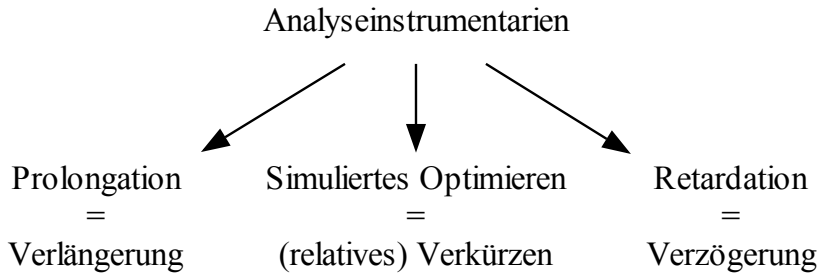


Abbildung 4.2: Ausprägungen des Analyseinstrumentariums.
Die in diesem Kapitel definierten Analyseinstrumentarien:
❑ Die Prolongation – die Verlängerung von Abläufen.
❑ Das simulierte Optimieren – ein (relatives) Verkürzen von Abläufen.
❑ Die Retardation – eine Verzögerung von Abläufen bzw. eine Verzögerung des Startzeitpunktes.
❑ Weitere Ausprägungen sind denkbar.

algorithmisch verlangsamen und verzögern

Jedoch ist ein anderer, auf den ersten Blick paradoxer, Ansatz möglich. Bei Systemen bestehend aus Hardware und Software ist es algorithmisch möglich, bestimmte Komponenten zielgerichtet zu verlangsamen oder den Startzeitpunkt zu verzögern. In dieser Arbeit werden zielgerichtet Performanzeigenschaften (insbesondere Startzeitpunkt, Laufzeitdauer und Endzeitpunkt einer Komponente sowie sinnvolle Kombinationen) verändert. Dies konterkariert eigentlich die Idee der Optimierung des Systems, der Ansatz steht diametral dazu, da hier „künstlich“ verlangsamt oder verzögert wird. Dass diese Idee jedoch als Instrumentarium für eine Performanzanalyse und eine nachfolgende Performanzoptimierung eingesetzt werden kann, wird nachfolgend in dieser Arbeit gezeigt.

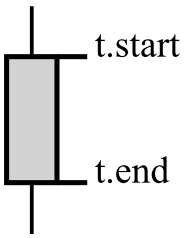


Abbildung 4.3: Abstrakte, schematische Darstellung der Komponentenausführung mit Startzeitpunkt und Endzeitpunkt einer Komponente.
In [Man06] wurde zur Illustration eine ein informelle Darstellung, angelehnt an ein Sequenzdiagramm, eingeführt. Die Länge des Aktivierungsbalken ist proportional zur Länge der (Methoden-)Ausführung.
Analog wird auch in dieser Arbeit, wie auch in [Man07], diese Darstellung gewählt.

4.3.1 Prolongation

Die Prolongation bezeichnet den Ansatz, die Performanz einer Komponente (Hard- oder Software) zu verringern [Man07].

8 Prolongation
Eine Prolongation bezeichnet ein zielgerichtetes Verlängern der Ausführungszeit einzelner Komponenten bzw. ein Verzögern des Endzeitpunktes einer Ausführung.

Die konkrete Realisation der Prolongation kann auf Hardwareebene[8] oder mittels Instrumentierung[9] geschehen.

[8] siehe Abschnitt 7.2 auf Seite 162
[9] siehe Abschnitt 7.1 auf Seite 139

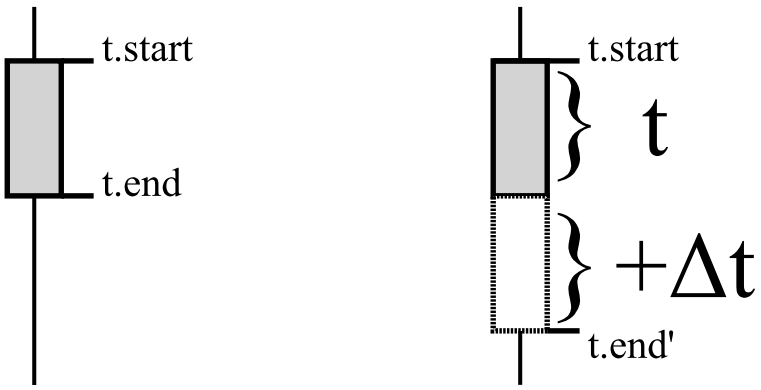


Abbildung 4.4: Prolongation.
Die Ausführungszeit des Moduls wird um Δt verlangsamt. Links ist ein Teil des Sequenzdiagramms für das nicht prolongierte Modul dargestellt, rechts das um Δt prolongierte Modul. Das Modul benötigt Δt mehr Zeit zur Abarbeitung seiner Funktion.

4.3.2 *Retardation*

Die Retardation bezeichnet in dieser Arbeit den Ansatz, den Startzeitpunkt einer Komponente (Hard- oder Software) zu verzögern.

9 Retardation
Eine Retardation bezeichnet ein zielgerichtetes Verzögern des Startzeitpunktes einer Ausführung.

4.3.3 *Simuliertes Optimieren*

Beispiel 4.3.1 Synchronisationskonzepte und Optimierungspotenzial
Bei Optimierungskandidaten ist noch nicht klar, wie sich ein System nach einer Optimierung verhält, ob sich eine Optimierung lohnt? Die Prozesse P_1 bis P_m und P_n laufen gleichzeitig. Die Prozesse P_1 bis P_m schreiben Daten in einen gemeinsamen Speicher, Prozess P_n soll diese Daten lesen. Um ein *schmutziges Leses* zu vermeiden, müssen hier Synchronisationskonzepte genutzt werden. Prozess P_n darf erst nach dem Schreiben aller Prozesse P_1 bis P_m lesen. Das gemessene Nadelöhr ist Prozess P_n, jedoch wird eine Optimierung durch das notwendige Warten kein Optimierungspotenzial bringen. [Mano7]

Durch die Prolongation können einzelne Abläufe verlängert werden. Dies impliziert, dass alle Abläufe verlangsamt werden können. Dies ist vergleichbar mit einem *Throttling*.[10][Mano7]

[10]Bei einem *Throttling* werden bei Berechnungen Takte ausgelassen, z. B. zum Schutz vor zu hohen Temperaturen

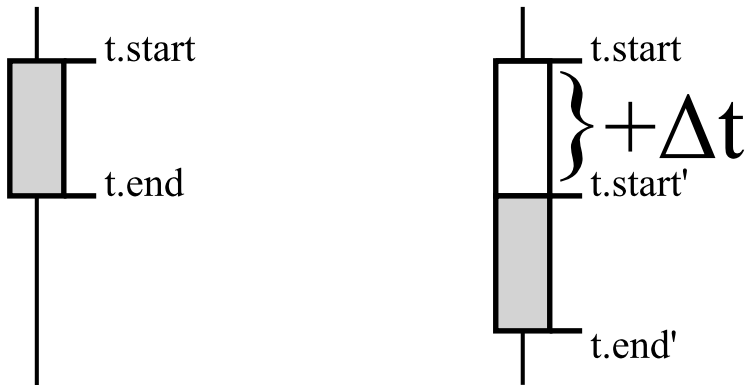


Abbildung 4.5: Retardation.
Der Anfangszeitpunkt t des Moduls wird um Δt verzögert. Links ist ein Teil des Sequenzdiagramms für das nicht retardierte Modul dargestellt, rechts das um Δt retardierte Modul. Das Modul beginnt um Δt verzögert mit der Abarbeitung seines Codes.

relative Optimierung

Verlangsamt man nun alle Abläufe, **bis auf einen Ablauf**, um einen bestimmten (Multiplikations)-Faktor, erscheint dieser Ablauf um den Faktor (relativ) schneller. Die entsprechenden Messwerte müssen für diese relative Verkürzung um diesen Faktor geteilt werden. Kapitel 6 von [Mano7] beschreibt das simulierte Optimieren ausführlich.
Das simulierte Optimieren bezeichnet den Ansatz, die Laufzeit einer Komponente (relativ) zu erhöhen bzw. zu verschnellern.

10 simuliertes Optimieren
Eine simuliertes Optimieren bezeichnet ein zielgerichtetes, relatives Verkürzen der Ausführungszeit einzelner Komponenten.

time dilation

Gupta u. a. [Gup+05] präsentieren mit der Arbeit „*To Infinity and Beyond: Time-Warped Network Emulation*“ einen ähnlichen Ansatz. Mittels einer virtuellen Maschine wird hier eine Beschleunigung des Netzwerk für einen empirischen Test simuliert. In der Arbeit wird eine Technik namens „*time dilation*“ entwickelt – dem Betriebssystem und den Applikationen wird eine langsamere Zeitrate als die physikalische Zeit „vorgegaukelt“. Physikalische Ereignisse erscheinen dadurch schneller. [Gup+05]
Beispielsweise vergeht für das Bestriebssystem durch Anwendung dieser Technik o.B.d.A. nur eine Sekunde für 10 Sekunden „wall clock time“. Daten die an der Netzwerkschnittstelle ankommen erscheinen zehnmal schneller. Die physikalischen Rate von 1 Gbps (*giga bits per second*) erscheint für das Betriebssystem wie 10 Gbps. [Gup+05]
Das simulierte Optimieren ist aber eine Stufe genereller, dadurch dass beliebe Abläufe simuliert werden können.

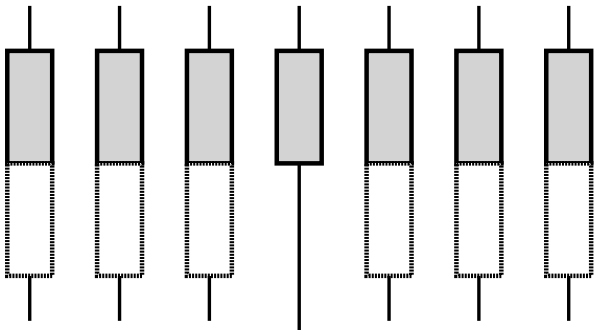


Abbildung 4.6: **Simuliertes Optimieren:**
❑ Im Bild sind fünf parallele Ausführungsstränge zu sehen, alle Ausführungsstränge bis auf den mittleren wurden prolongiert.
❑ Der Ausführungsstrang in der Mitte erscheint resultierend **relativ** zu den anderen Ausführungssträngen verkürzt.
❑ Analog kann dies auf sequentielle Ausführungen erweitert werden. Eine Prolongation aller Module einer sequentielle Ausführung bis auf ein bestimmtes Modul um einen Faktor, lässt dieses Modul um diesen Faktor schneller erscheinen. [Mano7, vgl. Kapitel 6]

4.3.4 Diskussion und weitere Ausprägungen des Analyseinstrumentariums

Basis der Analyse

Die Prolongation, die Retardation sowie das simulierte Optimieren dienen mit ihrer Möglichkeit zur Beeinflussung von zeitlichen Eigenschaften von Komponenten als Basis zur Analyse von Systemen bzw. als Basis des Experiments.

Basis Prolongation

In Kapitel 6 wird gezeigt, dass alle hier behandelten Ausprägungen des Analyseinstrumentariums (*Prolongation*, *Retardation* und *simuliertes Optimieren*) auf die *Prolongation* zurückgeführt werden können (siehe Seite 114).

sinnvolle Unterscheidung von Ausprägungen

Trotzdem ist es sinnvoll, zwischen der Prolongation und der Retardation auf „höherer Ebene“ zu unterscheiden. So ist es ein Unterschied ob eine Komponente (z. B. eine Festplatte) langsam ist (*Rotationsgeschwindigkeit*) oder noch nicht zur Verfügung steht (Köpfe geparkt).

weitere Ausprägungen

Im Prinzip sind auf der Basis der Prolongation weitere Ausprägungen des Analyseinstrumentariums möglich. Beispielsweise wäre eine Verschiebung des Startzeitpunktes nach „vorne“, eine „*Verfrühung*“ durch ein simuliertes Optimieren realisierbar. Ein Verschiebung des Endzeitpunktes nach „vorne“ kann durch ein simuliertes Optimieren realisiert werden.

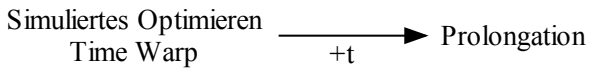


Abbildung 4.7: Das simulierte Optimieren im Vergleich zu einer Prolongation. Die Prolongation steht diametral zum simulierten Optimieren.
❑ Das *t* bezeichnet die Änderung der Zeit in den Abläufen.
❑ Mittels des simulierten Optimierens oder des Time Warps werden Abläufe relativ performanter gestaltet.
❑ Die Prolongation ist eine Verlängerung von Abläufen.

Exkurs 4.3.2 Entwicklung der Idee – Historie zur Arbeit
Da der implementierungsbasierte Ansatz beim Tuning bzw. bei der Feuerwehrperformanzanalyse von Szenario 11 (Seite 36) kein Erkenntnisgewinn ermöglichte (siehe Exkurs 3.3.1 auf Seite 48), wurde vom Autor die Idee der Prolongation am System ausprobiert. Teile des Systems wurden mittels AspectJ prolongiert.[11] So wurde bei geschickten Konstellationen eine Laufzeitverkürzung festgestellt (siehe Exkurs 7.1.1 auf Seite 152). Das Ergbnis wurde im Fortgeschrittenenpraktikum „Faktoranalyse“ [Mano6] veröffentlicht und von Dr. Michael Pönitsch patentieren lassen [Pöno6; Pöno9].
Dem Autor kam während der Versuche die Idee, diese gemessenen Auswirkungen, die einen Wirkzusammenhang darstellen, mittels statistischer Methoden[12] auszuwerten (der Titel des Fortgeschrittenenpraktikums Faktoranalyse als Singular von Faktorenanalyse verdeutlicht dies). Mitte 2007 wurde bei der Siemens CT SE 1, betreut von Dr. Moritz Hammer von der LMU, Dr. Bernhard Kempter und Dr. Harald Rölle von der Siemens AG die Diplomarbeit [Mano7] zu diesem Thema begonnen.
Die statistische Auswertung prolongierter Module ermöglichte, wie erhofft, eine Identifikation von Optimierungskandidaten in einem System [MHo8a; MHo8b; MHo8c]. Die Idee des simulierten Optimierens[13] war als eine Erweiterung schnell klar [MPKo8; MKPo7]. Um die Anzahl der Versuche bzw. die Komplexität zu reduzieren wurde der „Resource Dependence Graph“ oder „Ressource Dependence Graph“ [14] entwickelt [Man+o8c; Man+o8d; Man+o7; Man+o8a; Man+o8b].
Im November 2007 wurde mit der Dissertation zu dem Thema begonnen, betreut von Professor Dr. Martin Wirsing (LMU) und Dr. Harald Rölle (Siemens AG). Insbesondere sollte hier der intuitiv

[11] siehe Abschnitt 4.3.1, Seite 66
[12] siehe Abschnitt 9, Seite 201
[13] siehe Abschnitt 4.3.3, Seite 67
[14] siehe Abschnitt 9.4, Seite 215

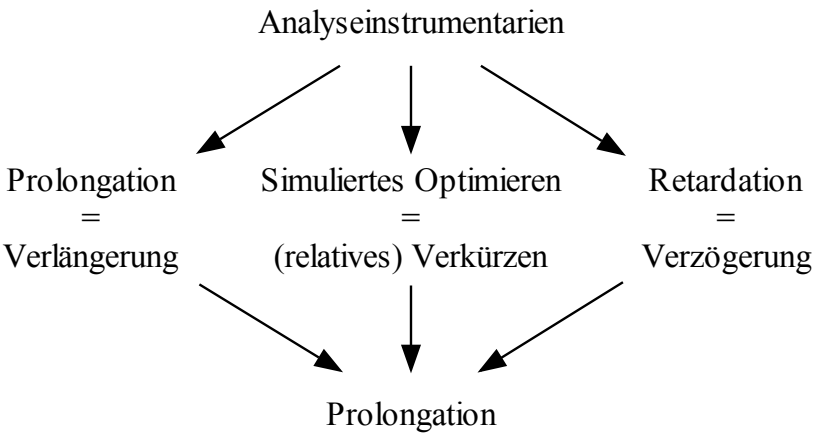


Abbildung 4.8: Die Basis für alle in diesem Kapitel definierten Analyseinstrumentarien ist die Prolongation.
❑ Das simulierte Optimieren ist die Prolongation aller Abläufe bis auf einen, resultierend erscheit dieser relativ gekürzt.
❑ Die Retardation ist die Prolongation bestimmter Ereignisse – siehe dazu Kapitel 6
❑ Weitere Ausprägungen auf Basis der Prolongation sind denkbar.

verständliche Wirkzusammenhang[15] dargelegt werden. Die Grenzen der Anwendbarkeit bzw. die Korrektheit des Analyseinstrumentariums[16] [MR10b; MR10a; MR09d; MR09c; MR09b; MR09a] sowie der Wirkzusammenhang mussten gezeigt werden.

[15] siehe Abschnitt 6, Seite 89
[16] siehe Abschnitt 5, Seite 73

4.4 Zusammenfassung und Beantwortung der Teilfragestellung γ

Lösungsansatz Experiment

In Abschnitt 4.1 wurde der Lösungsansatz für diese Arbeit präsentiert, ein empirischer Ansatz zur Peformanzanalyse. Dieser Ansatz ist ähnelt einem Experiment, wie sie beispielsweise in angewandten Ingenieurdisziplinen gebräuchlich und üblich sind.

Vorteile

Abschnitt 4.2 identifiziert die Vorteile eines Experimentes gegenüber den bestehenden Ansätzen zur Performanzanalyse (dem frühen modellbasierten Ansatz, dem implementierungsbasierten Ansatz und dem späten messungsbasierten Ansatz).
Ein Exkurs schildert die Historie und die Entwicklung der Idee zu dieser Arbeit.

Analyseinstrumentarien

Abschnitt 4.3 führt die verschiedenen, in dieser Arbeit realisierten und genutzten, Analyseinstrumentarien ein. Die *Prolongation*, die *Retardation* und das *simulierte Optimieren* werden als zeitliche Operationen definiert.

❑ Die *Prolongation* bezeichnet eine Verlängerung einer Ausführungszeit.
❑ Das *simulierte Optimieren* steht diametral dazu, es bezeichnet eine Verkürzung der Ausführungszeit.
❑ Die *Retardation* bezeichnet das Verschieben des Startzeitpunkts.

Der ähnliche Ansatz zum simulierten Optimieren von Gupta u. a. [Gup+05], die „*time dilation*“, mit einem anderem Anwendungsfall, wurde vorgestellt.

Die drei hier eingeführten zeitlichen Operationen lassen sich zurückführen auf die Prolongation. Weitere sinnvolle Kombinationen aus Veränderung des Start- oder Endzeitpunktes und der Laufzeitdauer sind möglich, beispielsweise ein nach „vorne“ schieben einer Ausführung bzw. eines Startzeitpunktes.

Teilfragestellung γ ✔

Somit konnte Teilfragestellung γ beantwortet werden:

- die Vorteile des empirischen Ansatzes wurden zum Stand der Wissenschaft und Technik (Abschnitt 4.1) betrachtet.
- wichtige Realisationen des Analyseinstrumentariums bzw. der Analyseinstrumentarien (*Prolongation*, *Retardation* und das *simulierte Optimieren*) wurden definiert.
- alle Realisationen des Analyseinstrumentariums lassen sich zurückführen auf die Prolongation.
- die Prolongation wurde, basierend auf Abläufen im generellen, abstrakt eingeführt. Wenn die Abläufe eines beliebigen Systems (welches nicht notwendigerweise aus der Informatik sein muss) verzögert werden können, können dieser in der Arbeit entwickelte Ansatz zur Analyse genutzt werden.

Kapitel 5

Korrektheit

„Die Wahrheit hat nichts zu tun mit der Zahl der Leute, die von ihr überzeugt sind."

Paul Claudel

Übersicht des Kapitels

Abschnitt 5.1 erklärt die Bedeutung der *internen Validität* oder *Ceteris paribus-Validität* bei einem Experiment und zieht den Vergleich zur *Korrektheit* von Algorithmen. Um Experimente mit dem Analyseinstrumentarium durchführen zu können, muss diese Validität gegeben sein, die Korrektheit muss während des Experiments beibehalten werden. Abschnitt 5.2 definiert die Prolongation bei einer Turingmaschine. Abschnitt 5.3 behandelt die Einzel-Prolongation einer Überführungsfunktion, es wird ein Zwischenschritt eingefügt. Abschnitt 5.4 behandelt die allgemeine Prolongation einer Überführungsfunktion, hier werden n Zwischenschritte eingefügt. Abschnitt 5.5 behandelt die Prolongation einer Turingmaschine, unterschiedliche Startzustände (z_1 bis z_m) können um beliebig viele Schritte (n_1 bis n_m) prolongiert werden.

5.1 Korrektheit in einem Experiment

Interne Validität (oder Ceteris paribus-Validität [Bou+04, S. 136]) liegt bei einem Experiment vor, wenn die beobachteten Wirkzusammenhänge nur durch die Änderung der

interne valide

variierten (unabhängigen) Variablen verursacht werden [Alb09]. Dies wird als intern valide bezeichnet [Alb09].

Korrektheit

Bei einer IT-Anwendung ist in einem Experiment die Korrektheit [Bas85] essentiell – wenn bestimmte Eigenschaften von Komponenten zielgerichtet für ein Experiment variiert werden, sollte die angedachte bzw. spezifizierte Funktionalität der IT-Anwendung beibehalten werden. Anstelle von intern validen Experimenten wird deswegen in dieser Arbeit von der Korrektheit des Analyseinstrumentariums gesprochen. Die Grenzen zwischen mathematisch formulierten Algorithmen (Korrektheit) und Experimenten (intern valide) verschwimmen in diesem Anwendungsfall des Analyseinstrumentariums.

Semantik

Es ist nicht offensichtlich, dass durch das Variieren von der Laufzeitdauern oder Zeitpunkten nicht die Berechnungen (die Semantik) eines Systems verändern. Dies wird in diesem Kapitel an einem sehr abstrakten, aber für die Informatik grundlegenden Modell, der Turingmaschine, dargestellt und bewiesen.

5.2 *Prolongation bei einer Turingmaschine*

Exkurs 5.2.1 historische Anmerkungen zur Turingmaschine

Alan Turings Turingmaschine

Die 1936 veröffentlichte Arbeit „On Computable Numbers, with an application to the Entscheidungsproblem“ mit der resultierenden Turingmaschine [Tur36] von Alan Turing diente als Vorbild für die ersten Rechner. Insbesondere war Turing entscheidend am Bau der Turing-Bombe, einer der ersten Computer, beteiligt deren logisches Design er 1939 fertigstellte [Teu04, S. 318]. Wegen ihrer hohen Abstraktion sind mit der Turingmaschine universale Beweisführungen möglich.

Gegeben ist eine deterministische Turingmaschine M mit ihrer Konfiguration K.

Sei $\delta : Z \times \Gamma \longrightarrow Z \times \Gamma \times \{L, R, N\}$ die Überführungsfunktion dieser deterministischen Turingmaschine, sei λ die Anzahl der Schritte die die Turingmaschine braucht um die Eingabe zu akzeptieren [Sch01, S. 81].

11 Prolongation bei einer Turingmaschine

Eine Prolongation bei einer Turingmaschine bezeichnet ein Ändern der Überführungsfunktion der Turingmaschine, so dass die Turingmaschine für bestimmte Eingaben in denen die Turingmaschine in bestimmte, vorher definierte Zustände geht, anstelle von λ, $\lambda + \mu$ Schritte benötigt, um diese zu akzeptieren.

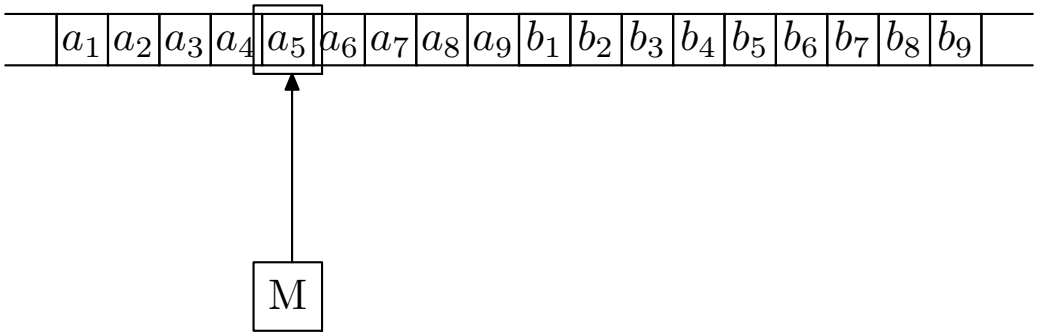


Abbildung 5.1: Turingmaschine mit unterschiedlichen Datenbereichen

5.3 *Einzel-Prolongation einer Überführungsfunktion*

12 Abkürzende Notation – die Prolongationsersetzung
Sei $\delta_{\langle z_0,a_0,1\rangle z_0}$ die Ersetzung der Überführungsfunktion δ in eine Überführungsfunktion, so dass für ein fest definiertes Bandzeichen $a_0 \in \Gamma$ und einem fest definiertem Zustand $z_0 \in Z$ gilt:

1. $\delta_{\langle z_0,a_0,1\rangle}(z,a)$ und δ für $(z \neq z_0 \vee a \neq a_0)$ mathematisch die gleiche Funktion definieren.
2. Die Ausführung von $\delta_{\langle z_0,a_0,1\rangle}(z_0,a_0)$ einen Schritt mehr braucht als die Ausführung von $\delta(z_0,a_0)$.

Dieses Ersetzen wird durch $\oplus_{\langle z_0,a_0,1\rangle}$ gekennzeichnet, wobei bei $\langle z_0,a_0,1\rangle$ z_0 den Zustand und a_0 das gelesene Bandsymbol der prolongierten Überführungsfunktion bezeichnen, 1 die prolongierte Schrittanzahl angibt.

Diese Ersetzung wird im folgenden abkürzend *Prolongationsersetzung*$_{\langle z_0,a_0,1\rangle}$ genannt.

13 Einzel-Prolongation einer Überführungfunktion
Eine **Einzel-Prolongation** einer Überführungfunktion im Zustand z_0 und gelesenem Bandsymbol a_0 bezeichnet die *Prolongationsersetzung*$_{\langle z_0,a_0,1\rangle}$ der Überführungsfunktion $\delta(z_0, a_0)$.

$$\delta(z_0, a_0) = (z_0', b_0, x)\ (x \in \{L, R, N\})$$

wird ersetzt durch

$$\delta(z_0, a_0) = (z_{Px1}, a_0, N) \text{ und } \delta(z_{Px1}, a_0) = (z_0', b_0, x) = \delta(z_0, a_0)$$

Die neue Überführungsfunktion der einzel-prolongierten Turingmaschine $T_{\langle z_0,a_0,1\rangle}$ ist wie folgt definiert:

$$\delta_{\langle z_0,a_0,1\rangle}(z, a) = \begin{cases} (z_{Px1}, a_0, N), & \text{falls } z = z_0, a = a_0 \\ \delta(z_{Px1}, a_0) = \delta(z_0, a_0), & \text{falls } z = z_{Px1}, a = a_0 \\ \delta(z, a), & \text{sonst } (z \neq z_0 \vee a \neq a_0) \end{cases}$$

wobei

a ein sich unter dem Schreib-Lesekopf befindliche Bandzeichen bezeichnet.

a_0 das sich unter dem Schreib-Lesekopf befindliche Bandzeichen bezeichnet, **bei dem die Überführungsfunktion prolongiert werden soll.**

b ein geschriebenes Bandzeichen bezeichnet.

b_0 das geschriebene Bandzeichen der prolongierten Überführungsfunktion bezeichnet.

$x \in \{L, R, N\}$ die Richtungsangabe des Kopfes darstellt.

T $(= (Z_T, \Sigma, \Gamma, \delta, z_{start}, \square, E))$ [Scho1, S. 81] die original Turingmaschine darstellt,

$T_{\langle z_0,a_0,1\rangle}$ $(= (Z_T \cup \{z_{Px1}\}, \Sigma, \Gamma, \delta\oplus_{\langle z_0,a_0,1\rangle}, z_{start}, \square, E))$ die einzel-prolongierte Turingmaschine bezeichnet, wobei

$z_{0,Px1}$ ein hinzugefügter Zustand ist ($z_{0,Px1} \notin Z_T$),
die Zustände z_0 und z_0' bereits vorhanden sind.

$Z_T \cup \{z_{0,Px1}\}$ wird als $Z_{0,PX1}$ bezeichnet.

$\delta\oplus_{\langle z_0,a_0,1\rangle}$ wird als $\delta_{\langle z_0,a_0,1\rangle}$ bezeichnet.

Der Index P deutet immer auf eine prolongierte Version hin.

Eine Überführungsfunktion wird um einen weiteren, „leeren“ Schritt erweitert, dadurch wird die Turingmaschine prolongiert. Die Überführungsfunktion im Zustand z_0 und

gelesenem Bandzeichen a_0 bewegt das Band bzw. den Lesekopf nicht weiter ($x = N$), die Überführungsfunktion des hinzugfügten Folgezustanden Zustand bewegt das Band bzw. den Lesekopf zu der ursprünglich angedachten Bandposition und schreibt das in der ursprünglich angedachte Bandzeichen. Ein Bearbeitungsschritt wird also durch zwei Bearbeitungsschritte ersetzt (für jedes Program, dass den Zustand z_0 bei gelesenem Bandzeichen a_0 erreicht).

5.3.1 Satz zur Einzelprolongation

Satz 1 Akzeptierte Sprachen einer einzelprolongierten Turingmaschine

Die Turingmaschine $T_{\langle z_0,a_0,1\rangle}$ akzeptiert genau die selbe Sprache wie die Turingmaschine T

$$T(M) = T_{\langle z_0,a_0,1\rangle}(M) = \{x, y \in \Sigma^* | z_{Start} x \vdash_* zy; y \in \Gamma^*; z \in E\}$$

Folgende Behauptung wird gezeigt:

$$\vdash_1^T \qquad \widehat{=} \qquad \vdash_{\leq 2}^{T_{\langle z_0,a_0,1\rangle}}$$

Dies wird bewiesen, in dem die Konfigurationen beider Turingmaschinen betrachtet werden und gezeigt wird, dass beide Turingmaschinen die selbe Sprache akzeptieren.

Lemma 1

$$\begin{aligned}&\forall \\ &\quad w, w', w'' \in (Z_T \cup \Gamma)^+, \\ &\quad a \in (Z_T \cup \Gamma), \\ &\quad z \in Z_T : \\ &wz_0aw' \vdash_1^T w'' \Leftrightarrow wz_0aw' \vdash_{\leq 2}^{T_{\langle z_0,a_0,1\rangle}} w''\end{aligned}$$

Wir vergleichen die Überführungsfunktionen beider Turingmaschinen:

Die Überführungsfunktion der original Turingmaschine T im Zustand z_0:

$$\delta(z_0, a_0) = (z_0', b_0, x)$$

Die Überführungsfunktion einer Turingmaschine $T_{\langle z_0,a_0,1\rangle,1}$ mit der Ersetzung, der einzel-prolongierten Überführungsfunktion:

$$\delta_{\langle z_0,a_0,1\rangle}(z,a) = \begin{cases} (z_{0,Px1}, a_0, N), & \text{falls } z = z_0, a = a_0 \\ \delta(z_{0,Px1}, a_0) = \delta(z_0, a_0), & \text{falls } z = z_{0,Px1}, a = a_0 \\ \delta(z,a), & \text{sonst } (z \neq z_0 \vee a \neq a_0) \end{cases}$$

5.3.2 Beweis

Die Konfiguration K sei $wzaw'$, wir führen den Beweis durch Fallunterscheidung durch:
Fall a) $z = z_0, a = a_0$

Fall $x = N$

linke Seite (original Turingmaschine T)
$wz_0a_0w' \vdash_1^T wz_0'b_0w'$
rechte Seite (einzel-prolongierte Turingmaschine $T_{\langle z_0,a_0,1\rangle,1}$)
$wz_0a_0w' \vdash_1^{T_{\langle z_0,a_0,1\rangle}} wz_{0,Px1}a_0w' \vdash_1^{T_{\langle z_0,a_0,1\rangle}} wz_0'b_0w'$

Fall $x = L$
Sei $w = uy$ mit $u \in \Sigma^\star$ und y das letzte Zeichen von w bzw. $y = \square$, falls $w = \epsilon$
linke Seite (original Turingmaschine T)
$uyz_0a_0w' \vdash_1^T uz_0'yb_0w'$
rechte Seite (einzel-prolongierte Turingmaschine $T_{\langle z_0,a_0,1\rangle,1}$)
$uyz_0a_0w' \vdash_1^{T_{\langle z_0,a_0,1\rangle}} uyz_{0,Px1}a_0w' \vdash_1^{T_{\langle z_0,a_0,1\rangle}} uz_0'yb_0w'$

Fall $x = R$ folgt analog
linke Seite (original Turingmaschine T)
$wz_0a_0w' \vdash_1^T wb_0z_0'w'$ falls $w' \neq \epsilon$
$wz_0a_0\square \vdash_1^T wb_0z_0'\square$ falls $w' = \epsilon$
rechte Seite (einzel-prolongierte Turingmaschine $T_{\langle z_0,a_0,1\rangle,1}$)
$wz_0a_0w' \vdash_1^{T_{\langle z_0,a_0,1\rangle}} wz_{0,Px1}a_0w' \vdash_1^{T_{\langle z_0,a_0,1\rangle}} wb_0z_0'w'$ falls $w' \neq \epsilon$
$wz_0a_0\square \vdash_1^{T_{\langle z_0,a_0,1\rangle}} wz_{0,Px1}a_0\square \vdash_1^{T_{\langle z_0,a_0,1\rangle}} wb_0z_0'\square$ falls $w' = \epsilon$

Fall b) $z \in Z_T$ **oder** $a \neq a_0$

Dann ist $\delta(z,a) = \delta(z,a)$

linke Seite (original Turingmaschine T)

$wz_0aw' \vdash_1^T wz_0'bw'$

rechte Seite (einzel-prolongierte Turingmaschine $T_{\langle z_0,a_0,1\rangle,1}$)

$wz_0aw' \vdash_1^{T_{\langle z_0,a_0,1\rangle}} wz_0'bw'$

Fall c) $z = z_{0,Px1}$

Kann nicht auftreten, da $z \in T_T$ sein muss.

5.4 *Prolongation einer Überführungsfunktion*

Lemma 2 Eine $Prolongationsersetzung_{\langle z_0,a_0,n\rangle}$ einer Überführungfunktion im Zustand z_0 und gelesenem Bandzeichen a_0 bezeichnet abkürzend das n-fache iterative Anwenden einer $Prolongationsersetzung_{\langle z,a,1\rangle}$ für $z \in \{z_0 \cup z_{PX1} \cup \ldots \cup z_{PXn-1}\}$ (siehe die Anmerkungen zur Notation 12, Seite 75).

Dabei wird der Zustand z_0 und die Zustandsmenge $Z_{\langle z_0,a_0,n-1\rangle}$ zur neuen Zustandsmenge $Z_{\langle z_0,a_0,n\rangle}$ vereinigt, insgesamt werden nach einer $Prolongationsersetzung_{\langle z_0,a_0,n\rangle}$ n neue Zustände zur ursprünglichen Zustandsmenge Z hinzugefügt, die o.B.d.A. mit $z_{0,Px1}, \ldots, z_{0,Pxn}$ fortlaufend nummeriert werden.

Sei $\delta_{\langle z_0,a_0,n\rangle}$ die Ersetzung der Überführungsfunktion $\delta_{\langle z_0,a_0,n-1\rangle}$ in eine Überführungsfunktion, so dass für ein fest definiertes Bandzeichen $a_0 \in \Gamma$ und einem fest definiertem Zustand $z_0 \in Z \cup \{z_{0,Px1}, \ldots, z_{0,Pxn-1}\}$ gilt:

1. $\delta_{\langle z_0,a_0,n\rangle}(z,a)$ und δ für $(z \neq z_0 \vee a \neq a_0)$ mathematisch die gleiche Funktion definieren
2. Die Ausführung von $\delta_{\langle z_0,a_0,n\rangle}(z_0,a_0)$ n Schritte mehr braucht als die Ausführung von $\delta(z_0,a_0)$ bzw. einen Schritt mehr braucht als $\delta_{\langle z_0,a_0,n-1\rangle}(z_0,a_0)$

Dieses Ersetzen wird durch $\oplus_{\langle z_0,a_0,n\rangle}$ gekennzeichnet, wobei bei $\langle z_0,a_0,n\rangle$ z_0 den Zustand und a_0 das gelesene Bandsymbol der prolongierten Überführungsfunktion bezeichnen, n die prolongierte Schrittanzahl angibt.

Diese Ersetzung wird im folgenden abkürzend $Prolongationsersetzung_{\langle z_0,a_0,n\rangle}$ genannt.

14 Prolongation einer Turingmaschine

Eine **Prolongation** einer Überführungfunktion im Zustand z_0 und gelesenem Bandzeichen a_0 bezeichnet die *Prolongationsersetzung*$_{\langle z_0,a_0,n\rangle}$ der Überführungsfunktion $\delta(z_0, a_0)$.

Nach Anwendung einer *Prolongationsersetzung*$_{\langle z_0,a_0,n\rangle}$ wurde die Überführungsfunktion

$$\delta(z_0, a_0) = (z_0', b_0, x) \text{ mit } x \in \{L, R, N\}$$

ersetzt durch

$\delta(z_0, a_0) = (z_{0,Px1}, a_0, N)$, $\delta(z_{0,Px1}, a_0) = (z_{0,Px2}, a_0, N) \ldots \delta(z_{0,Pxn}, a_0) = (z_0', b_0, x)$

Die Überführungsfunktion der Turingmaschine mit einer prolongierten Überführungsfunktion ($T_{\langle z_0,a_0,n\rangle}$) ist wie folgt definiert:

$$\delta_{\langle z_0,a_0,n\rangle}(z, a) = \begin{cases} (z_{0,Px1}, a_0, N), & \text{falls } z = z_0, a = a_0 \\ (z_{0,Pxi+1}, a_0, N), & \text{falls } z = z_{0,Pxi}, a = a_0, 1 \leq i < n \\ (z_0', b_0, x), & \text{falls } z = z_{0,Pxn}, a = a_0 \\ \perp, & \text{falls } z = z_{0,Pxi}, a \neq a_0, 1 \leq i \leq n \\ \delta(z, a) & \text{falls } z \in Z \setminus \{z_0\} \text{ oder } a \neq a_0 \end{cases}$$

wobei

a ein sich unter dem Schreib-Lesekopf befindliche Bandzeichen bezeichnet.

a_0 das sich unter dem Schreib-Lesekopf befindliche Bandzeichen bezeichnet, **bei dem die Überführungsfunktion prolongiert werden soll.**

b ein geschriebenes Bandzeichen bezeichnet.

b_0 das geschriebene Bandzeichen der prolongierten Überführungsfunktion bezeichnet.

$x \in \{L, R, N\}$ die Richtungsangabe des Kopfes darstellt.

T $(= (Z_T, \Sigma, \Gamma, \delta, z_{start}, \square, E))$ [Scho1, S. 81] die original Turingmaschine darstellt,

$T_{\langle z_0,a_0,n\rangle} (= (Z_T \cup \{z_{0,Px1}, \ldots, z_{0,Pxn}\}, \Sigma, \Gamma, \delta\oplus_{\langle z_0,a_0,n\rangle}, z_{start}, \square, E))$ die Turingmaschine mit der prolongierten Übergangsfunktion bezeichnet

$z_{0,Px1}, \ldots, z_{0,Pxn}$ n hinzugefügte Zustände sind ($(z_{0,Px1} \cup \cdots \cup z_{0,Pxn}) \notin Z_T$)

$Z_T \cup \{z_{0,Px1}, \ldots, z_{0,Pxn}\}$ wird als $Z_{\langle z_0,a_0,n\rangle}$ bezeichnet.
$\delta\oplus_{\langle z_0,a_0,n\rangle}$ wird als $\delta_{\langle z_0,a_0,n\rangle}$ bezeichnet.

Die Prolongation einer Überführungsfunktion bewegt beim Zustand z_0 und Bandsymbol a_0 den Schreib-Lesekopf nicht und „zählt iterativ die Zustände hoch“. Beim n-ten zusätzlichen Schritt wird dann der Lesekopf in die eigentliche Richtung weiterbewegt und das ursprünglich angedachte Symbol auf das Band geschrieben. Die Ausführung der ursprünglichen Überführungsfunktion ist hier am Ende der Kette definiert worden.

Wir vergleichen die Überführungsfunktionen beider Turingmaschinen:

Die Überführungsfunktion bei der ursprünglichen Turingmaschine T:

$$\delta(z_0, a_0) = (z_0', b_0, x)$$

Die Überführungsfunktion einer Überführungsfunktion prolongierten Turingmaschine $T_{\langle z_0,a_0,n\rangle}$:

$$\delta_{\langle z_0,a_0,n\rangle}(z,a) = \begin{cases} (z_{0,Px1}, a_0, N), & \text{falls } z = z_0, a = a_0 \\ (z_{0,Pxi+1}, a_0, N), & \text{falls } z = z_{0,Pxi}, a = a_0, 1 \leq i < n \\ (z_0', b_0, N), & \text{falls } z = z_{0,Pxn}, a = a_0 \\ \perp, & \text{falls } z = z_{0,Pxi}, a \neq a_0, 1 \leq i \leq n \\ \delta(z,a) & \text{falls } z \in Z \setminus \{z_0\} \text{ oder } a \neq a_0 \end{cases}$$

5.4.1 *Satz zur Prolongation einer Überführungsfunktion*

Satz 2 Akzeptierte Sprachen bei der Prolongation einer Überführungsfunktion

Die Turingmaschine $T_{\langle z_0,a_0,n\rangle}$ akzeptiert genau die selbe Sprache wie die Turingmaschine T

$$T(M) = T_{\langle z_0,a_0,n\rangle}(M) = \{x, y \in \Sigma^* | z_{Start} x \vdash_* zy; y \in \Gamma^*; z \in E\}$$

Dies wird bewiesen, in dem die Konfigurationen beider Turingmaschinen betrachtet werden und gezeigt wird, dass beide Turingmaschinen die selbe Sprache akzeptieren [Scho1, S. 83].

Lemma 3

$\forall$

$w, w', w'' \in (Z_T \cup \Gamma)^+$,
$a, a_0 \in (Z_T \cup \Gamma)$,
$z \in Z_T$:

1. $wz_0a_0w' \vdash_1^T w'' \Leftrightarrow wz_0a_0w' \vdash_{n+1}^{T_{\langle z_0,a_0,n\rangle}} w''$
2. $waw' \vdash_1^T w'' \Leftrightarrow waw' \vdash_1^{T_{\langle z_0,a_0,n\rangle}} w''$ falls $z \notin Z_0 \vee a \neq a_0$

Der Fall 2 ($n = 1$) wurde in Abschnitt 5.3 wurde auf Seite 78 gezeigt. Zu zeigen ist jetzt der allgemeine Fall 1 für $n > 1$.
Für den Beweis führen wir folgendes Lemma ein:

Lemma 4 Sei
$wz_0a_0w' \vdash_1 w''$ bei der Turingmaschine T
Dann gilt für alle $n \in \mathbb{N}, i < n$ bei der Turingmaschine $T_{\langle z_0,a_0,n\rangle}$:
$wz_{0,Px(n-i)}a_0w' \vdash_{i+1} w''$ mit $0 \leq i < n$

Beweis durch Induktion:

Induktionsanfang $i = 0$

$wz_{0,Px(n-0)}a_0w' \vdash_1 w''$

Dies gilt nach Definition 14 da $z_{0,Px(n-0)} = z_{0,Px(n)}$

Induktionsschritt $i \Rightarrow i + 1$

Zu zeigen:

$wz_{0,Px(n-i-1)}a_0w' \vdash_{i+2} w'' \quad \forall i < n$ bzw.
$wz_{0,Px(n-(i+1))}a_0w' \vdash_{i+2} w'' \quad \forall i < n$

Nach Induktionsvoraussetzung

$$wz_{0,Px(n-(i+1))}a_0w' \vdash_1 \underbrace{wz_{0,Px(n-i)}a_0w' \vdash_{i+1} w'' \quad \forall i < n+1}_{\text{Induktionsvoraussetzung}}$$

$$\widehat{=}$$

$$wz_{0,Px(n-(i+1))}a_0w' \vdash_{(i+1)+1} w''$$

■

5.4.2 *Beweis*

Die Konfiguration K sei $wzaw'$, wir führen den Beweis durch Fallunterscheidung durch:

Fall a) $z = z_0, a = a_0$

Fall $x = N$

linke Seite (original Turingmaschine T)
$wz_0a_0w' \vdash_1^T wz_0'b_0w' = w''$

rechte Seite (Turingmaschine mit einer prolongierten Überführungsfunktion $T_{\langle z_0,a_0,n\rangle}$)
$wz_0a_0w' \vdash_1^{T_{\langle z_0,a_0,n\rangle}} wz_{0,Px1}a_0w' \vdash_1^{T_{\langle z_0,a_0,n\rangle}} wz_{0,Px2}a_0w' \vdash_1^{T_{\langle z_0,a_0,n\rangle}} \ldots \vdash_1^{T_{\langle z_0,a_0,n\rangle}}$
$wz_{0,Pxn}a_0w' \vdash_1^{T_{\langle z_0,a_0,n\rangle}} wz_0'b_0w' = w''$
$\underbrace{\underbrace{wz_0a_0w' \vdash_1^{T_{\langle z_0,a_0,n\rangle}} wz_{0,Px1}a_0w'}_{\text{Nach Definition}} \vdash_n^{T_{\langle z_0,a_0,n\rangle}} w''}_{\text{Nach Lemma 4}}$

Fall $x = L$

Sei $w = uy$ mit $u \in \Sigma^\star$ und y das letzte Zeichen von w bzw. $y = \square$, falls $w = \epsilon$

linke Seite (original Turingmaschine T)
$uyz_0a_0w' \vdash_1^T uz_0'yb_0w' = w''$

rechte Seite (Turingmaschine mit einer prolongierten Überführungsfunktion $T_{\langle z_0,a_0,n\rangle}$)
$uyz_0a_0w' \vdash_1^{T_{\langle z_0,a_0,n\rangle}} uyz_{0,Px1}a_0w' \vdash_1^{T_{\langle z_0,a_0,n\rangle}} uyz_{0,Px2}a_0w' \vdash_1^{T_{\langle z_0,a_0,n\rangle}} \ldots \vdash_1^{T_{\langle z_0,a_0,n\rangle}}$
$uyz_{0,Pxn}a_0w' \vdash_1^{T_{\langle z_0,a_0,n\rangle}} uz_0'yb_0w' = w''$
$\underbrace{\underbrace{uyz_0a_0w' \vdash_1^{T_{\langle z_0,a_0,n\rangle}} uyz_{0,Px1}a_0w'}_{\text{Nach Definition}} \vdash_n^{T_{\langle z_0,a_0,n\rangle}} w''}_{\text{Nach Lemma 4}}$

Fall $x = R$ folgt analog

Fall b) $z \neq z_0, a \neq a_0$

Dann ist $\delta(z,a) = \delta(z,a)$

linke Seite (original Turingmaschine T)
$wz_0aw' \vdash_1^T wz_0'bw'$

rechte Seite (Turingmaschine mit einer prolongierten Überführungsfunktion $T_{z_0,n}$)

$$wz_0aw' \vdash_1^{T_{\langle z_0,a_0,n\rangle}} wz_0'bw'$$

Fall c) $z = z_{0,Px1} \vee z = z_{0,Px2} \vee \ldots \vee z = z_{0,Pxn}$ ist implizit in Fall a gezeigt worden. Auf der linken Seite kann $z_{0,Px1}$ bis $z_{0,Pxn}$ nicht auftreten, da $z_{0,Px1}$ bis $z_{0,Pxn} \notin Z_T$.

■

5.5 *Prolongation einer Turingmaschine*

Lemma 5 Eine *allgemeine Prolongationsersetzung*$_{\langle z_1,a_1,D_1\rangle,\ldots,\langle z_m,a_m,D_m\rangle}$ bezeichnet abkürzend das iterative Anwenden einer *Prolongationsersetzung*$_{\langle z,a,1\rangle}$ für $z \in \{Z_T \cup z_{PX1,1} \cup \ldots \cup z_{PXm,D_m}\}$ und $a \in \Sigma$ (siehe die Anmerkungen zur Notation 12, Seite 75), wobei jede prolongierte und dadurch transitive Kette der Überführungsfunktion (für $a_0 \in \Sigma$) von $z_i \in Z_T$ bis zu $z_i' \in Z_T$ als zusätzlicher Index $\langle z_i, a_0, D_i\rangle$ angegeben wird, D_i bezeichnet die hinzugefügte zusätzliche Schrittanzahl in der transitiven Kette.

Es gilt, dass:

1. $\delta_{\langle z_1,a_1,D_1\rangle,\ldots,\langle z_m,a_m,D_m\rangle}(z,a)$ und δ für $z \notin \{Z_T \cup z_{PX1,1} \cup \ldots \cup z_{PXm,D_m}\}$ oder a nicht für die Prolongation angedachtes Bandsymbol mathematisch die gleiche Funktion definieren
2. Die Ausführung von $\delta_{\langle z_1,a_1,D_1\rangle,\ldots,\langle z_m,a_m,D_m\rangle}(z_i,a_i)$ für $z_i \in \{Z_T \cup z_{PX1,1} \cup \ldots \cup z_{PXm,D_m}\}$ und a_i als für die Prolongation gewähltes Bandsymbol D_i Schritte mehr braucht als die Ausführung von $\delta(z_i,a_i)$

15 Prolongation einer Turingmaschine Eine **Prolongation** einer Turingmaschine bezeichnet die Prolongation der Überführungsfunktion bei $m \in \mathbb{N}$ Zuständen ($\delta(z_1,a_1)$ bis $\delta(z_m,a_m)$ im Zustand z_i und gelesenem Bandzeichen a_i um zusätzliche $D_i + 1$ Schritte ($1 \leq i \leq m$)).

Dieses Ersetzen wird durch $\oplus_{\langle z_1,a_1,D_i\rangle,\ldots,\langle z_m,a_m,D_m\rangle}$ gekennzeichnet, wobei bei $\langle z_i, a_i, D_i\rangle$ z_i den Zustand und a_i das gelesene Bandsymbol der prolongierten Überführungsfunktion bezeichnen, D_i die prolongierte Schrittanzahl angibt.

Die so erhaltene Turingmaschine wird als $T_P = T_{\langle z_1,a_1,D_1\rangle,\ldots,\langle z_m,a_m,D_m\rangle}$ bezeichnet.

5.5.1 Satz zur Prolongation einer Turingmaschine

Satz 3 Eine prolongierte Turingmaschine akzeptiert genau die selbe Sprache wie eine nicht prolongierte Turingmaschine

Eine prolongierte Turingmaschine $T_{\langle z_1,a_1,D_1\rangle,\ldots,\langle z_m,a_m,D_m\rangle}$ akzeptiert genau die selbe Sprache wie eine nicht prolongierte Turingmaschine T.

Lemma 6

$\forall$
$w, w', w'' \in \left(Z_{\langle z_1,a_1,D_i\rangle,\ldots,\langle z_m,a_m,D_m\rangle} \cup \Gamma\right)^+$,
$a \in \Gamma$,
$z \in Z_T$:

1. $wz_ia_iw' \vdash_1^T w'' \Leftrightarrow wz_ia_iw' \vdash_{D_i+1}^{T_{\langle z_1,a_1,D_1\rangle,\ldots,\langle z_m,a_m,D_m\rangle}} w''$
2. $wzaw' \vdash_1^T w'' \Leftrightarrow wzaw' \vdash_1^{T_{\langle z_1,a_1,D_1\rangle,\ldots,\langle z_m,a_m,D_m\rangle}} w''$
   falls $z \notin \{(z_1,a_1),\ldots,(z_1,a_1)\}$

Der Fall 2 ($n = 1$) wurde in Abschnitt 5.3 wurde auf Seite 78 gezeigt. Zu zeigen ist jetzt der allgemeine Fall 1 für $D_i > 1$.
Der Fall 1 für $m = 1$ wurde in Abschnitt 2 auf Seite 81 gezeigt. Zu zeigen ist jetzt der allgemeine Fall 1 für $m > 1$.

5.5.2 Beweis

Der Beweis folgt direkt aus dem allgemeinen Lemma , wir führen den Beweis durch Induktion über m durch.

Induktionsanfang $m = 1$

Der Induktionsanfang wurde mit Satz 2 auf Seite 81 gezeigt. Die Prolongation einer Überführungsfunktion ändert nichts an der akzeptierten Sprache der Turingmaschine.

$T(M) = T_{\langle z_1,a_1,D_i\rangle}(M) = = T_{\langle z_0,a_0,1\rangle}(M) = \{x, y \in \Sigma^* | z_{Start}x \vdash_* zy; y \in \Gamma^*; z \in E\}$

Induktionsvoraussetzung:

$i = 1 \ldots m$
$\forall\ w, w', w'' \in \left(Z_{\langle z_1,a_1,D_1\rangle,\ldots,\langle z_{i-1},a_{i-1},D_{i-1}\rangle} \cup \Gamma\right)^+$, $a \in \Gamma$,

$$
\begin{aligned}
& wz_i a_i w' \vdash_1^T w'' \\
& \Leftrightarrow [\textit{wg. 2.}] \\
& wz_i a_i w' \vdash_1^{T_{\langle z_1,a_1,D_1\rangle,\ldots,\langle z_{i-1},a_{i-1},D_{i-1}\rangle}} w'' \\
& \Leftrightarrow [\textit{Satz 2}] \\
& wz_i a_i w' \vdash_{D_i+1}^{T_{\langle z_1,a_1,D_1\rangle,\ldots,\langle z_{i-1},a_{i-1},D_{i-1}\rangle \oplus \langle z_i,a_i,D_i\rangle}} w''
\end{aligned}
$$

Induktionsschritt $m \Rightarrow m+1$**:**

$$
\begin{aligned}
& \forall\, w, w', w'' \in \left(Z_{\langle z_1,a_1,D_1\rangle,\ldots,\langle z_{i-1},a_{i-1},D_{i-1}\rangle} \cup \Gamma\right)^+, a \in \left(Z_{\langle z_1,a_1,D_1\rangle,\ldots,\langle z_{i-1},a_{i-1},D_{i-1}\rangle} \cup \Gamma\right), \\
& wz_{m+1} a_{m+1} w' \vdash_1^T w'' \\
& \Leftrightarrow [\textit{wg. 2.}] \\
& wz_{m+1} a_{m+1} w' \vdash_1^{T_{\langle z_1,a_1,D_1\rangle,\ldots,\langle z_m,a_m,D_m\rangle}} w'' \\
& \Leftrightarrow [\textit{Satz 2}] \\
& wz_{m+1} a_{m+1} w' \vdash_{D_{m+1}+1}^{T_{\langle z_1,a_1,D_1\rangle,\ldots,\langle z_m,a_m,D_m\rangle \oplus \langle z_{m+1},a_{m+1},D_{m+1}\rangle}} w''
\end{aligned}
$$

■

Mit diesem Beweis wurde Satz 3 gezeigt. Eine Turingmaschine kann prolongiert werden – es dürfen beliebige „leere“ Schritte (NOPs) eingefügt werden, ohne dass das Ergebnis einer Berechnung verändert wird.

5.6 Zusammenfassung, Beantwortung von Teilfragestellung δ und Ausblick

Interne Validität

Abschnitt 5.1 führt die Begriffe „*Interne Validität*“ (oder *Ceteris paribus-Validität*) bei einem Experiment ein und vergleicht sie mit dem Begriff der Korrektheit. Abschnitt 5.2 definiert eine Prolongation bei einer Turingmaschine als ein Ändern der Überführungsfunktion, so dass die Turingmaschine bei bestimmten, vorher definierten Zuständen, mehr Schritte benötigt um die Eingabe zu akzeptieren.

Prolongation in einer Turingmaschine

NOPs in einer sequentiellen Rechnung

In Abschnitt 5.3 wurde mittels Induktion bewiesen, dass ein zusätzlich eingefügter „leerer Schritt“ in eine beliebige Überführungsfunktion einer Turingmaschine nicht das Ergebnis ihrer Berechnung verändert. Dieser leere Schritt kann bei anderen Maschinenmodellen als No-Operation Befehl (*NOP*) interpretiert werden. Abschnitt 5.4 beweist, dass beliebig viele solcher *NOP*s eingefügt werden können und das Ergebniss einer beliebigen Berechnung beibehalten wird.

Teilfragestellung δ ✔

Dieses Kapitel zeigt insgesamt, dass das Ergebnis einer sequentiellen Berechnung nicht verändert wird, so lange durch das hinzugefügte Element (z. B. ein *NOP* oder ein „leerer“ Schritt bei einer Überführungsfunktion einer Turingmaschine) kein Zustand verändert wird. Damit konnte die nicht offensichtliche Teilfragestellung δ unter dieser Bedingung beantwortet werden.

isoliertes Modell

Die Turingmaschine ist ein sehr abstraktes Maschinenmodell, ohne Beeinflussung von außen. Sie ist durch ihre Abstraktion geeignet für grundlegende Beweise. Jedoch sind die Korrektheit und der Wirkzusammenhang nicht nur im sequentiellen sondern auch bei parallelen Berechnungen für die Praxisrelevanz des Analyseinstrumentariums von Interesse und Bedeutung. Für parallele Berechnungen würde sich eine mehrbändige Turingmaschine, wie sie in Abbildung 5.2 dargestellt ist, eignen.

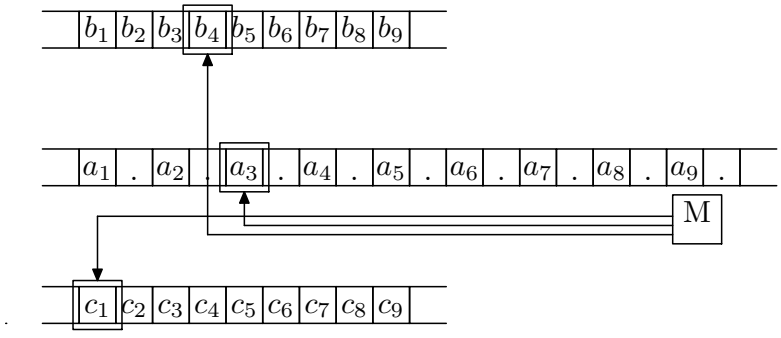


Abbildung 5.2: Mehrbändige Turingmaschine

Modell- und Systembegriff

Da das Ziel dieser Arbeit eine praxisrelevante Performanzanalyse ist und ein System als Summe von ausgeführter Software, Hardware und externer Faktoren definiert wurde (siehe Definition 2 auf Seite 16) eignet sich dieses Modell nicht gut (keine gemeinsame Ressourcennutzung, externe Faktoren sind schlecht integrierbar, eine Uhr bzw. die Zeit ist schlecht darstellbar uvm.). Insbesondere hat dieses Modell keine Synchronisationskonzepte. Im Kontext dieser Arbeit in Erwägung gezogene Möglichkeiten wären beispielsweise *Timed Automata* und erweiterte *Sequenzdiagramme*.

Timed Automata

parallele Registermaschine

Da in dieser Arbeit in Kapitel 7 eine Experimentierumgebung mittels einer virtuellen Maschine realisiert wird, bot sich als praxisnahes, paralleles Modell und zur Darstellung des Wirkzusammenhangs die im nächten Kapitel eingeführte parallele Registermaschine an.

Kapitel 6

Wirkzusammenhang

„Jede Zeit ist eine Sphynx, die sich in den Abgrund stürzt, sobald man ihr Rätsel gelöst hat."

Heinrich Heine

Übersicht des Kapitels

Der Wirkzusammenhang des Analyseinstrumentariums wird in diesem Kapitel aufgezeigt. Dies geschieht mittels eines praxisnahen Maschinenmodells, der parallelen Registermaschine oder kurz PRAM. Abschnitt 6.1 führt die Registermaschine als ihr sequentielles Vorläufermodell ein. In Abschnitt 6.2 wird die parallele Registermaschine (kurz PRAM) als paralleles Maschinenmodell eingeführt. Abschnitt 6.3 definiert die PRAM^{dt}, eine durch eine Akkumulator-Architektur simplifizierte PRAM. Diese ist angelehnt an die gut erforschte, dokumentierte und praktisch umgesetzte *SB-PRAM*. Abschnitt 6.4 diskutiert die Korrektheit bei dem propagiertem Experiment in diesem parallelem Maschinenmodell. Abschnitt 6.5 zeigt den Wirkzusammenhang des Analyseinstrumentariums. Hier wird dargestellt, wieso durch den experimentellen Ansatz Optimierungspotenzial und -kandidaten in einem System gefunden werden können.

6.1 Die Registermaschine

Exkurs 6.1.1 Turing- vs. Registermaschine

praxisnähere Modellierung

Wegen ihrer Einfachheit erlaubt die Turingmaschine elegante Beweisführungen. Deshalb wurde sie in Kapitel 5 zum Beweis der Korrektheit des Analyseinstrumentariums benutzt. 1963 wurde,

inspiriert durch die Architektur der sich etablierten Rechner eine praxisnähere Modellierung vorgeschlagen [KB06]. Shepherdson und Sturgis veröffentlichten 1963 in der Arbeit „Computability of Recursive Functions“ ihr Modell einer Random Access Machine oder RAM [SS63]. Beide Maschinenmodelle, die Turingmaschine und die Random Access Machine, können sich gegenseitig ohne superpolynomiellen Laufzeitverluste simulieren [Rei99, S. 76 ff.]. In dieser Arbeit wird anstelle von Random Access Machine der analoge deutsche Terminus Registermaschine verwendet.

Registermaschine

Eine Registermaschine ist ein Modell eines realen, sequentiellen Computers nach dem Von-Neumann Modell [Neu93]. Der Speicher ist analog dem eines modernen Computers aufgebaut [AHU74].
Cook und Reckhow [CR72] geben als Erweiterung ein reelles, logarithmisches Kostenmaß $l(n)$ zum Speichern von Werten an. Dies ist als Berechnungsmodell sinnvoll, da ansonsten beliebig große Werte in Registern gespeichert werden könnten, was bei realen Maschinen nicht realisiert werden kann.

Eine Registermaschine RAM besteht aus [CR72]:

16 Definition Registermaschine Einem Eingabeband

Das Eingabeband besteht aus einer abzählbaren Folge von Registern $x_1, x_2, \ldots$. Der Inhalt der Register wird analog zu [CR72] mit $X_0, X_1, X_2, \ldots$ bezeichnet.

Dem Speicher der Registermaschine

Der Speicher der Registermaschine besteht aus einer abzählbaren Folge von Registern $R_0, R_1, \ldots$. Diese Register können beliebige natürliche Zahlen speichern. Der Index i ist die Adresse der Speicherzelle. R_0 (auch abgekürzt als AC, z. B. in [CR72]) wird als Akkumulator bezeichnet. Zum Start einer Berechnung sind alle Register mit 0 initialisiert.

Dem Befehlszähler

Der aktuelle Zustand einer Registermaschine ist eindeutig durch den Befehlszähler IC und dem Inhalt des Datenspeichers konfiguriert.

Einem Programm

Eine endliche, fortlaufende, nummerierte Folge $p_1, p_2, \ldots$ von Befehlen aus einer Befehlsmenge.

Einer Befehlsmenge

Die Befehlsmenge bzw. der Befehlssatz $\mathcal{B}$ der Registermaschine. Der (zumeist einheitlich verwendete) Befehlssatz der RAM ist in Tabelle 6.1 angegeben. In der Spalte „Beschreibung/Wirkung“ ist angegeben, wie die Konfiguration der Registermaschine aus dieser Definition bei der Ausführung eines Befehls verändert wird. Die linke Seite, getrennt durch ein $\leftarrow$, gibt den neuen Inhalt des Registers **nach** der Ausführung des Befehls an, die rechte Seite den Inhalt **vor** der Ausführung des Befehls [SS63, S. 219].

Exkurs 6.1.2 Parallele Berechnungsmodelle

Parallele Berechnungsmodelle

Einer der ersten Computer, der ENIAC, der als „erste[r] allgemeine[r], großformatige[r], elektronische[r] Computer der Welt“ [BB81] (zitiert nach [Roj97, S. 26]) bezeichnet wird, war bereits hochgradig parallel und dezentralisiert [HJ88]. Seit dem über Rechenmaschinen nachgedacht wird, war Parallelität eine naheliegende Option um die Berechnungsgeschwindigkeit zu steigern (z. B. *Babbage* mit seiner *difference machine* [HJ88, S. 10]).

theoretische Informatik

Die theoretische Informatik entdeckte das Konzept der Parallelität jedoch erst in den späten siebziger Jahren für sich [Kel92, S. 5]. Die Registermaschinen wurden 1978 in der Arbeit „*Parallelism in random access machines*“ um Parallelität erweitert [FW78].

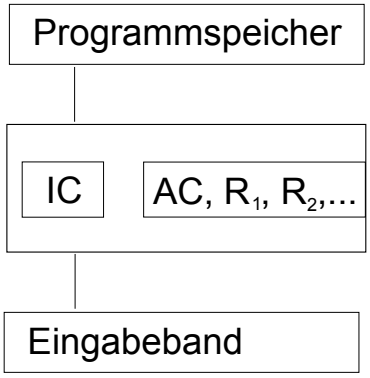


Abbildung 6.1: Ein idealisiertes Abbild einer Registermaschine, bestehend aus:
❑ Dem Programmspeicher
❑ Dem Befehlszähler (*IC*)
❑ Dem (unendlichen) Speicher der Registermaschine ($AC, R_1, R_2, \ldots$)
❑ Einem Eingabeband

6.2 Einführung – Die parallele Registermaschine (PRAM)

In den folgenden Abschnitten wird die parallele Registermaschine (PRAM) eingeführt. Zur Dokumentation des Wirkzusammenhangs wird eine eigene parallele Registermaschine formal dargelegt.

6.2.1 Wahl des Maschinenmodells

Darstellung des Wirkzusammenhangs

Für diese Arbeit zur Darstellung des Wirkzusammenhangs des Analyseinstrumentariums wird ein theoretisches Modell benötigt, die parallele Registermaschine (PRAM) bietet sich wegen ihrer Einfachheit und Nähe zu realen Systemen zur Darstellung an. Insbesondere da gemeinsame Ressourcennutzung und parallele Prozesse integriert werden können, eignet sie sich zur Darstellung des Wirkzusammenhangs des Analyseinstrumentariums.

frühes, abstrahiertes Modell

Das PRAM Modell ist ein historisch sehr frühes Modell, es wurde konzipiert zur Entwicklung von parallelen Algorithmen. Wichtige Erkenntnisse und Konzepte moderner paralleler Programmierungstechniken, wie Kommunikationsverzögerungen und Synchronisationskonzepte, sind nicht integriert oder wurden abstrahiert.

keine einheitliche Definition der PRAM

Im Gegensatz zur Registermaschine wird in der Literatur kein einheitliche Modell der PRAM angegeben. Es werden Modelle mit unterschiedlichen Attributen definiert, passend für das zu erörternde Problem. Uneinheitlichkeit herrscht vor allem hinsichtlich der Befehlsmenge und der Speicherorganisation. Des Weiteren wird dazu tendiert, die Un-

Operation	mnemonic	Beschreibung/Wirkung
load constant	LOD, j	$AC \leftarrow j$; $IC \leftarrow IC + 2$
add	ADD, j	$AC \leftarrow AC + X_j$; $IC \leftarrow IC + 2$
subtract	SUB, j	$AC \leftarrow AC - X_j$; $IC \leftarrow IC + 2$
store	STO, j	$X_j \leftarrow AC$; $IC \leftarrow IC + 2$
branch on positive Accumulator	BPA, j	*if* $AC > 0$ *then* $IC \leftarrow j$; *otherwise* $IC \leftarrow IC + 2$
read	RD, j	$X_j \leftarrow$ *next Input*; $IC \leftarrow IC + 2$
print	PRI, j	*output* X_j; $IC \leftarrow IC + 2$
halt	HLT	*stop*

Tabelle 6.1: Befehlssatz einer typischen Registermaschine (nach [CR72, S. 75]).

tersuchung und Diskussion von einer sehr hohen Abstraktionsebene, wie beispielsweise einer Hochsprache, die parallele Programmierkonzepte unterstützt (z. B. *concurrent Pascal* [Han75b; Han75a] oder *Fork* [Hag+94; KT96; Keß+94]), durchzuführen.

6.2.2 *Aufbau einer PRAM*

Parallelität durch RAMs

Informell ist die Idee hinter einer eine Parallel Random Access Machine (PRAM), dass aus (beliebig vielen) sequentiellen Registermaschinen (siehe Definition 16) mit einer Kommunikationsmöglichkeit durch einen gemeinsamen, globalen Speicher ein paralleles Maschinenmodell erzeugt wird.

6.2.2.1 Minimalkonsens der PRAM

Minimalkonsens PRAM

Als Minimalkonsens werden parallele Registermaschinenen (PRAMs), als eine Erweiterung der Registermaschine, wie folgt angegeben:
Eine PRAM besteht aus:

- **n Prozessoren**
  Einer Menge von Prozessoren $P_0, P_1, \ldots, P_{n-1}$ [FW78].
- **Globaler Speicher**
  Einem globalem Speicher M [FW78].

6.2.2.2 Erweiterter Aufbau der PRAM

Als erweiterter Konsens wird die parallele Registermaschine, nach Fortune und Wyllie [FW78], in der Literatur folgendermaßen angegeben:
Eine Parallel Random Access Machine (PRAM) besteht aus:

- **n Prozessoren**
  Einer unendlichen, abzählbaren Menge von Prozessoren $P_0, P_1, \ldots, P_{n-1}$ [Hag+94]. Jeder dieser Prozessoren hat einen unendlichen, abzählbaren lokalen Speicher [Fic+89].
- **Globaler Speicher**
  Einem unendlichen, abzählbaren globalen Speicher M. Die lokalen sowie die globalen Speicherzellen sind, beginnend mit 0, fortlaufend nummeriert. Der Index der Speicherzelle wird auch Adresse genannt.
- **Programmspeicher**
  Ein spezieller Speicher, in dem das Programm der PRAM gespeichert ist.
- **Globaler Taktgeber**
  Eine globale Uhr steuert alle Prozessoren. Die Uhr fungiert als Taktgeber und jeder Prozessor führt genau einen Berechnungsschritt während eines Takts bzw. Zyklus aus. Dieser Zyklus ist in drei Phasen geteilt (siehe Abschnitt 6.2.3).
- **Zusätzlichen Mikrobefehlen**
  Die PRAM enthält eine echte Obermenge der Befehle einer RAM (vgl. [AHU74]). Der vollständige Befehlssatz der PRAM wird in Abschnitt 6.2.5 angegeben.

Speicher für Eingabewerte

Anstelle eines gemeinsamen globalen Speichers, der initial mit den Eingabewerten belegt ist, wurden, analog zur Registermaschine[1] andere Varianten vorgeschlagen, die einen zusätzlichen, abgesonderten, eigenständigen Speicher für Eingabewerte aufweisen [MNV94; LY89; Fic+89]. Somit sollen auch Algorithmen mit sublinearer Laufzeit behandelt werden können[2].

[1] siehe Definition 16, Seite 91

[2] Siehe auch: Mikrobefehl *READ*, Abschnitt 6.2.5.1, Seite 100.

Eine PRAM kann demnach als eine synchrone ***MIMD*** (*multiple-instruction multiple data*) Architektur mit gemeinsamem Speicher klassifiziert werden [Fly72]. MIMD

6.2.3 Phasen in einer PRAM

Während jedes Schritts einer Berechnung kann ein Prozessor (P_i) neben lokalen Operationen entweder von **einer Zelle** (M_j) des gemeinsamen oder des lokalen Speichers lesen oder in eine Zelle des gemeinsamen Speichers oder des lokalen Speichers schreiben. Hierbei können unterschiedliche Prozessoren (P_i und $P_l, (i \neq l)$) unterschiedliche Operationen durchführen. Phasen der PRAM

Die drei einzelnen Schritte **Lesen, lokale Operation** und **Schreiben** werden zur Vereinfachung als prozessorübergreifende, hintereinander stattfindende separate Phasen angesehen – dies erleichtert die Analyse und verlängert die Laufzeit nur um einen konstanten Faktor [Fic+89]. prozessorübergreifende, separate Phasen

6.2.4 PRAM Modelle zur Lösung von Schreib- und Lesekonflikten

Wenn zwei Prozessoren P_i und $P_l, (i \neq l)$ parallel Operationen auf einer identischen, gemeinsamen Speicherzelle (M_j) durchführen, kann dies zu Inkonsistenzen führen. Beispielsweise können zwei unterschiedliche Prozessoren P_i und $P_l, (i \neq l)$ unterschiedliche Werte in die Speicherzelle M_j schreiben. Hierzu müssen Konventionen getroffen werden, die unterschiedliche PRAM-Modelle definieren [MV84; DR00]. Konflikte

Die trivialste Möglichkeit besteht darin, nur exklusives Schreiben oder Lesen eines Prozessors zuzulassen – was natürlich die Anzahl der Berechnungsschritte des Maschinenmodells erhöht, wenn parallele Schreib- oder Leseoperationen vorteilhaft wären. Eine kurze Diskussion über die Mächtigkeiten unterschiedlicher, in der Literatur definierter Modelle, befindet sich in Exkurs 6.2.1. Konventionen und Mächtigkeit des Modells

In der Literatur werden die Modelle mit "**E**" für exklusives Lesen ("**R**" - für das englische read) oder Schreiben ("**W**" - analog für das englische write) bzw. mit "**C**" für gleichzeitiges ("concurrent") Lesen und Schreiben abgekürzt. In Tabelle 6.2.4 sind die Klassifikation der PRAM Modelle dargestellt, wobei eine 1 exklusive Schreib- oder Leseoperationen indizieren, n für gleichzeitiges Lesen oder Schreiben von beliebig vielen Prozessoren auf eine gemeinsame, identische Speicherzelle steht. E - exclusive read/write
C - concurrent read/write

6.2.4.1 Modelle mit exklusiven Zugriffen

Im verbleibenden Abschnitt werden die Eigenschaften der einzelnen Modelle kurz illustriert und die wichtigsten in der Literatur verwandten Konventionen für die CRCW PRAM vorgestellt.

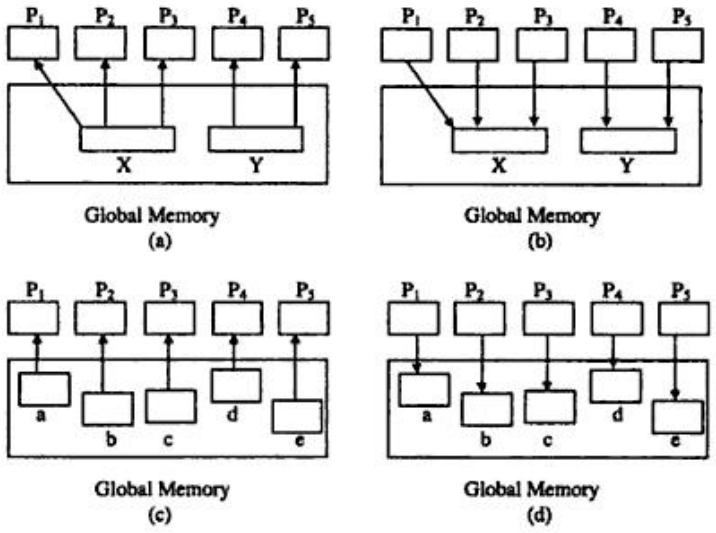
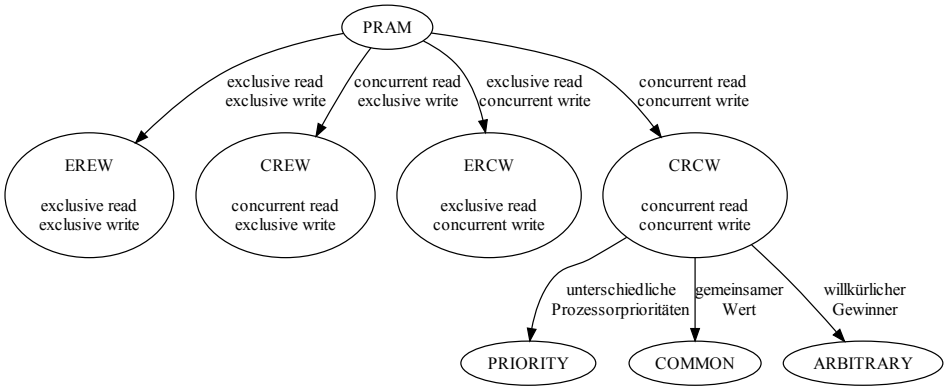


Abbildung 6.2: Berechnungsmodelle zum Zugriff auf gemeinsame, globale Speicherzellen bzgl. Lese- und Schreiboperationen. (a) Concurrent read (b) Concurrent write (c) Exclusive read (d) Exclusive write. Zitiert nach [Iva02, S. 102].



Die in Abschnitt 6.2.4 behandelten PRAM Modelle klassifiziert nach Konventionen für parallelen Zugriff auf identische Speicherzellen.

Abbildung 6.3: Behandelte PRAM Modelle

		read: 1	read: n
write	1	EREW	CREW
	n	CREW	CRCW

Tabelle 6.2: Klassifikation von PRAM Modellen anhand paralleler Zugriffskonventionen auf identische, gemeinsame Speicherzellen des globalen Speichers. Eine 1 indiziert eine exklusive Operation, ein „n“ zeigt gleichzeitig mögliche Operationen auf identische Speicherzellen an.

❑ **exclusive read exclusive write (EREW)**

EREW

Nur ein Prozessor kann in einem Bearbeitungsschritt aus dem gemeinsamen Speicher lesen und in den gemeinsamen Speicher schreiben. Dieses Modell wurde in [LPV81] eingeführt.

❑ **concurrent read exclusive write (CREW)**

CREW

Ursprüngliche Maschine, eingeführt in [FW78]. Beliebig viele Prozessoren können gleichzeitig in einem Bearbeitungsschritt von der gleichen Speicherzelle lesen, jedoch nur ein Prozessor kann in einem Schritt in eine beliebige Speicherzelle schreiben.

CROW EROW

Eine Erweiterung sind Modelle, die Schreibkonflikte vermeiden, in dem jeder Prozessor für seine Schreiboperationen eine eigene Speicherzelle hat, beispielsweise concurrent-read owner-write (**CROW PRAM**) [DR86; DR00] und exclusive-read owner-write (**EROW PRAM**) [FW89; FW90] (zitiert nach [Fic+89]).

6.2.4.2 Modelle mit gleichzeitigen Zugriffen

charakteristisches Modell CRCW PRAM

Im Modell CRCW müssen durch die gleichzeitigen Lese- und Schreibzugriffe besondere Konventionen getroffen werden, da dieses Modell durch die Operationen von vielen Prozessoren auf eine gemeinsame Speicherzelle prädestiniert für Konflikte ist. Dieses Modell ist jedoch besonders interessant, da es die charakteristische Idee der PRAM mit parallelen Operationen einzelner Prozessoren am besten integriert.

❑ **CRCW – concurrent read concurrent write**

Standard CRCW Modelle

Die nach [Fic+89, S. 5] in der Literatur am häufigsten benutzten Modelle für parallele Registermaschinen mit gleichzeitigem Lese- und Schreibzugriff auf identische Speicherzellen sind:

❑ **COMMON**

COMMON

Das COMMON Modell arbeitet nach der Prämisse, dass alle gleichzeitig schreibenden Prozessoren einen gemeinsamen Wert in die selbe Speicherzelle speichern [Kuc82] (zitiert nach [Fic+89, S. 5]).

❑ **ARBITRARY**

ARBITRARY

Im ARBITRARY Modell bleibt von allen geschriebenen Werten ein willkürlicher Wert in der Speicherzelle erhalten. Damit muss jeder für dieses Modell konstruierte Algorithmus unabhängig vom tatsächlich gespeicherten Wert funktional sein. [Vis83](zitiert nach [Fic+89, S. 5]).

❑ **PRIORITY**

PRIORITY

Im PRIORITY-Modell haben Prozessoren festgelegte, unveränderbare Prioritäten, die Schreibkonflikte lösen sollen [Gol78].

Exkurs 6.2.1 Mächtigkeit der Modelle

Mächtigkeitshierarchie

Ein Modell A wird in der theoretischen Informatik mächtiger genannt, wenn es auf Grund einer Hierarchie (z. B. der Hierarchie formaler Grammatiken, die eine formale Sprache erzeugen [Cho56]) eine höhere Hierarchiestufe als Eingabe akzeptiert wie ein Modell B.
In der Literatur wird der Begriff „mächtiger“ im Kontext von PRAMs nicht explizit definiert sondern intuitiv verwandt.

Definition 1 Mächtigkeit
Ein Modell A ist mächtiger ($\geq$) als ein Modell B, wenn jeder Algorithmus der für Modell B geschrieben wurde auf Modell A unverändert ausgeführt werden kann.

PRIORITY $\leq$ ARBITRARY

Damit sind nicht alle Modelle gleichmächtig. Jeder Algorithmus der auf einem ARBITRARY Modell fehlerfrei ausgeführt wird, läuft unverändert auf einer PRIORITY PRAM. Damit ist das PRIORITY Modell mindestens so mächtig wie ARBITRARY.

ARBITRARY $\leq$ COMMON

Ähnlich ist ARBITRARY mindestens so mächtig wie COMMON wie in [Bop89] und [Edm91] gezeigt wird (zitiert nach [Fic+89, S.17]).

COMMON $\leq$ CREW PRAM

Fich, Ragde und Wigderson [FRW84] zeigen, dass eine COMMON PRAM durch eine CREW PRAM simuliert werden kann.

CREW PRAM $\leq$ EREW PRAM

Dass eine Maschine, die gleichzeitig mehrere gemeinsame Speicherzellen lesen kann, mächtiger ist, als ein Modell welches nur exklusives Lesen erlaubt, sollte intuitiv klar sein. Für einen Beweis sei auf [FRW84] verwiesen.
Damit ergeben sich folgende Zusammenhänge:

PRIORITY $\leq$ ARBITRARY $\leq$ COMMON $\leq$ CREW PRAM $\leq$ EREW PRAM

Fich, Li, Ragde und Yesha [Fic+89, S. 9] geben tabellarisch, und damit sehr übersichtlich, die bekannten Zusammenhänge bzw. Transformations- bzw. Simulationsaufwand der hier präsentierten PRAM Modelle an.

Instruktion		Funktion
LOAD	*operand*	Führt die Operation mit Hilfe des Akkumulators aus.
STORE	*operand*	
ADD	*operand*	
SUB	*operand*	
JUMP	*label*	Der Befehlszähler wird auf label gesetzt.
JZERO	*label*	
READ	*operand*	Siehe Text.
FORK	*operand*	
HALT	*operand*	

Tabelle 6.3: Der in Fortune und Wyllie [FW78] angegebene und damit ursprüngliche Befehlssatz einer parallelen Registermaschine.

6.2.5 Befehlssatz der PRAM

uneinheitlicher Befehlssatz

In der Literatur herrscht völlige Uneinheitlichkeit bezüglich des Befehlssatzes einer PRAM. Zu meist wird auch, wie bereits erwähnt, in Hochsprachen argumentiert. So geben Fortune und Wyllie [FW78, S. 115] einen *FORK* Befehl zum Erzeugen eines neuen Prozessors an, während andere Versionen diese nicht haben dafür jedoch einen *CALL* Befehl zum Aufruf eines Unterprogramms (z. B. die LPRAM von Savitch [Sav82]).

6.2.5.1 Ursprünglicher Befehlssatz nach Fortune und Wyllie

ursprünglicher Befehlssatz

Die ursprüngliche parallele Registermaschine wurde 1978 von Fortune und Wyllie [FW78] mit der in Tabelle 6.3 angegebenem Befehlssatz definiert, die untenstehend weiter erläutert werden.

- *LOAD*, *STORE*

  LOAD und *STORE* führen typische Speicheroperationen, Lesen und Schreiben bzw. Speichern, analog einer RAM, im lokalen Speicher durch [FW78, S. 114, 115].

- *ADD*, *SUB*

  Führen Addition und Subtraktion analog zu einer Registermaschine aus [FW78, S. 114, 115].

- ❑ *JUMP*, *JZERO*

  JUMP führt einen unbedingten Sprung aus. Der Befehlszähler (Instruction Counter, *IC*) wird auf die Adresse „*LABEL*“ gesetzt. Mit dem hier gespeicherten, codierten Befehl wird die Verarbeitung fortgesetzt [FW78, S. 114, 115].
  JZERO führt einen bedingten Sprung aus. Der Befehlszähler (Instruction Counter, *IC*) wird auf die Adresse „*LABEL*“ gesetzt, **falls** der Akkumulator den Wert 0 enthält [FW78]. Mit dem in „*LABEL*“ gespeicherten, codierten Befehl wird die Verarbeitung fortgesetzt [FW78, S. 114, 115].
- ❑ *READ*

  Der Wert des Operanden eines *Read* Befehls spezifiziert das Eingaberegister. Der Inhalt dieses Registers wird in den Akkumulator geladen [FW78, S. 114, 115]. Spezielle Register für die Eingabe haben bei diesem Maschinenmodell den Sinn, um auch sublineare Laufzeiten erörtern zu können.
- ❑ *FORK*

  Ein *Fork Label* Befehl, der von Prozessor P_i ausgeführt wird, sucht sich den ersten inaktiven Prozessor P_j, löscht den lokalen Speicher dieses Prozessors und kopiert den Akkumulator von P_i nach P_j. Danach startet P_j mit den Befehlen am Label *Label* [FW78, S. 114, 115].
- ❑ *HALT*

  Ein *Halt* Befehl der von Prozessor P_i ausgeführt wird, lässt diesen stoppen [FW78, S. 114, 115].

CREW

Fortune und Wyllie [FW78] führen ihre Maschine als CREW Modell ein. Schreibt mehr als ein Prozessor gleichzeitig auf eine identische Speicherzelle, bricht die parallele Registermaschine sofort ab und verwirft. Fortune und Wyllie [FW78] erwähnen implizit die in Abschnitt 6.2.3 eingeführten Phasen, in dem sie alle (gleichzeitig möglichen) Leseoperationen vor Schreiboperationen ansiedeln [FW78, S. 115].

Phasen

Akzeptanz, Determinismus

Die von Fortune und Wyllie [FW78] vorgeschlagene Maschine akzeptiert die Eingabe nur, falls der erste Prozessor (P_0) mit einer 1 im Akkumulator stoppt. Ein Programm ist nicht deterministisch, falls ein label für einen Sprungbefehl (*JUMP*, *JZERO*) mehr als ein mal existiert, ansonsten deterministisch [FW78, S. 114].

typischer Befehlssatz

Andere Autoren definieren, wenn überhaupt, den Befehlssatz ihrer PRAM unterschiedlich. Greenlaw, Hoover und Ruzzo [GHR95, S. 22] sprechen deswegen auch von einem „typischem“ Befehlssatz einer PRAM. Greenlaw, Hoover und Ruzzo [GHR95, S. 23] definieren beispielsweise den zusätzlichen *IDENT* Befehl[3], der die Nummer des aktuellen Prozessors in den Akkumulator lädt.

[3] bei anderen Maschinenmodellen, z. B. bei der SB-PRAM, auch als *LOADINDEX* bezeichnet.

6.2.6 Die SB-PRAM

SB-PRAM

Umfassende Forschung wurde von der Universität Saarland zum Thema parallelen Registermaschinen betrieben. Hier wurde ein praktischer Prototyp der PRAM, die SB-PRAM (Saarbrücken Parallel Random Access Machine) realisiert [GRR95; DKP02; Bac+97; Hag+94]. Darauf aufbauend wurden Programmiersprachen implementiert, wie beispielsweise die Programmiersprache *Fork* [KT96; Hag+94; Keß+94; Kes97] sowie *p4* für parallele Registermaschinen [GRR95] [4].

Programmiersprachen

Hagerup u. a. [Hag+94, S. 304] erweiterten den Befehlssatz einer Registermaschine für eine parallele Registermaschine um folgende Befehle:

- *LOADINDEX*
  Lädt den Index des ausführenden Prozessors in eine Speicherzelle des lokalen Speichers.
- *WRITE*
  Schreibt einen Wert aus einer Speicherzelle des lokalen Speichers in eine Speicherzelle des globalen Speichers.
- *READ*
  Schreibt einen Wert aus einer Speicherzelle des globalen Speichers in eine Speicherzelle des lokalen Speichers.

zusätzliche Mikrobefehle

Diese PRAM unterscheidet sich von der ursprünglichen definierten parallelen Registermaschine von Fortune und Wyllie [FW78] um die zusätzlichen Maschinenbefehle *LOADINDEX*, *WRITE* und *READ* (der Befehl *READ* wird in dieser parallelen Registermaschine mit einer anderen Semantik[5] definiert als von Fortune und Wyllie [FW78]) [GRR95; DKP02; Bac+97]. Fortune und Wyllie [FW78] definieren den Maschinenbefehl *Fork* während Hagerup u. a. [Hag+94, S. 305ff] diesen als maßgebliches Konstrukt in die Hochsprache, nach dem diese auch benannt ist, implementieren.

6.2.6.1 Multipräfixoperationen

Fluent Machine

Ranade, Bhatt und Johnsson [RBJ88] haben mit Hilfe einer *Fluent Machine*, bestehend aus über 100.000 Prozessoren verknüpft durch ein *Butterfly*-Netzwerk [RBJ88, S. 1], eine *PRIORITY CRCW PRAM* simuliert [Sch95, S. 8][RBJ88]. Hier erweitern Ranade, Bhatt und Johnsson [RBJ88, S. 3] die Operationen *fetch-and-op* der *NYU Ultracomputer* [GLR81] und *scan Operations* der *Connection machine* [Ble89] und nennen diese *Multiprefix Operationen* (oder auch *Multipräfix Operationen*). Mehrere disjunkte Prozessoren(-gruppen) können parallel und synchron Operationen mit einer assoziativen Funktion auf eine gemeinsame Speicherzelle mit Daten ausführen [Sch95, S. 8].

[4] siehe Exkurs 6.2.2, Seite 103

[5] siehe Tabelle 6.3, Seite 99

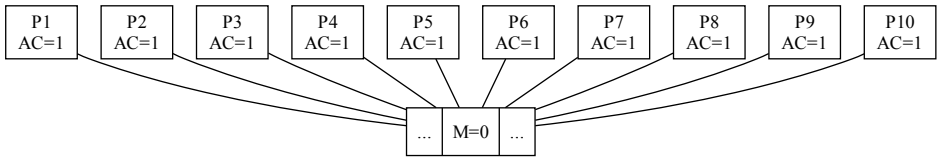
17 Multipräfix
Seien $P_{i_1}, \ldots, P_{i_k}$ Prozessoren, die in einem Schritt auf eine gemeinsame, identische Speicherzelle mit der Adresse A die assoziative Operation $\odot$ mit den Daten $R_{0,i_1} = D_{i_1}, \ldots, R_{0,i_k} = D_{i_k}$ anwenden. Vor der Operation sei der Inhalt der Speicherzelle $M(A) = D$. Nach diesem Schritt gilt für den Inhalt der Speicherzelle $M(A) = D \odot D_{i_1} \odot \ldots \odot D_{i_k}$. Der Akkumulator des Prozessors P_{i_j} erhält den Wert $D \odot D_{i_1} \odot \ldots \odot D_{i_{j-1}}$ zurück (Definition angelehnt an [Sch95, S. 9, Definition 2.3]).

SB-PRAM Auch die SB-PRAM besitzt, aufbauend auf Ranade, Bhatt und Johnsson [RBJ88], diese *Multipräfix Operationen* [She93]. Dies ermöglicht der SB-PRAM so auch mittels Parallelität mehrere Operationen gleichzeitig auf einer identischen, gemeinsamen Speicherzelle auszuführen [GRR95, S. 2][KRR96; For+97]. So können beispielsweise elegant und schnell Sperren (*locks*) und Barrieren realisiert werden[6].

Zyklen Bei der praktischen Realisierung der SB-PRAM benötigen diese Multipräfixoperationen, durch das verbindende Netzwerk [Sch95, S. 13], zwei Runden (clock cycles) [Kes]. Dies wird von Scheerer [Sch95, S. 13, Bezeichnung 2.1] als delayed load bezeichnet. Zur Vereinfachung der Analyse, aufbauend auf Definition 17 und angelehnt an eine „optimale" PRAM benötigt die Multipräfixoperation in dem in dieser Arbeit definierten (hypothetisch optimalen) Maschinenmodell[7] eine Runde/Zyklus.

Beispiel 6.2.1 MPADD

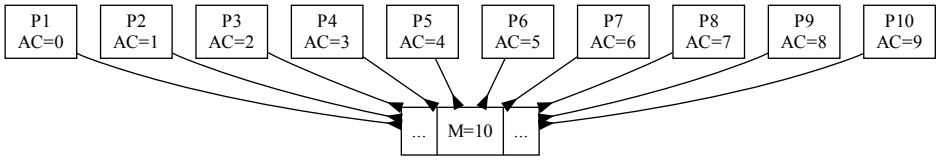
- Eine Gruppe von 10 Prozessoren, in Abbildung 6.2.1 als P1, P2, P3, P4, P5, P6, P7, P8, P9, P10 bezeichnet, führen eine *MPADD*-Operation auf die gemeinsame Speicherzelle M des globalen Speichers aus.
  - Jeder dieser Prozessoren hat den Wert 1 im Akkumulator.
  - Die globale, gemeinsame Speicherzelle M hat den Wert 0 gespeichert.



- Nach der Ausführung ergibt sich folgende Situation:

[6] siehe Abschnitt 6.4.6, Seite 121
[7] siehe Abschnitt 6.3, Seite 103

- Prozessor P1 hat den Inhalt der Speicherzelle M übernommen.
- Prozessor P2 hat den Inhalt der Speicherzelle M plus den Inhalt des Akkumulators von P1 übernommen.
- ...
- Prozessor P10 hat den Inhalt der Speicherzelle M plus den Inhalt aller Akkumulatoren von P1 bis P9 übernommen.
- Die gemeinsame, globale Speicherzelle M erhält als Wert die Summe aller Akkumulatoren von P1 bis P10 und den ursprünglichen Wert von M.



Exkurs 6.2.2 Höhere Programmiersprachen für die PRAM

fehlende Standardprogrammiersprache

Bis dato gibt es keine Standard oder de facto Standard höhere Programmiersprache für die PRAM [Hag+94; KT96]. Hagerup u. a. [Hag+94, S. 1] zählen hier die einige der sich ergebenden naheliegenden Nachteile auf. Zwar gibt es generelle Entwicklungen im Bereich der parallelen Programmierung und Programmiersprachen, jedoch speziell zugeschnitten auf autonome Systeme mit eigenem Taktgeber [Hag+94, S. 1].

autonome Systeme

So unterscheiden Hagerup u. a. [Hag+94, S. 1] vor allem zwei Konzepte der Interprozessorkommunikation, die diverse Programmiersprachen implementieren:

❑ **Message exchange**

Der Nachrichtenaustausch wird von Programmsprachen wie ADA [Taf+02; PZ97], OC-CAM [Cor84; WWMi93] und Concurrent C [GR88b; SLG88; GR89] bzw. Concurrent C++ [GR88a] unterstützt.

❑ **Protected shared variables**

Concurrent Pascal [Han75a; Han75b] hat gemeinsame, geschützte Variablen zum Austausch von Werten.

Problematisch ist auch, dass vor allem das wegen seiner Einfachheit oft zur Erörterung benutzte Concurrent Pascal nicht die Mächtigkeit einer PRAM mit der MIMD Architektur ausnutzt. Zur Darstellung des Wirkzusammenhangs muss jedoch auf Hochsprachen verzichtet werden, sondern es muss direkt auf die Prozessorarchitektur und Befehlsvorrat eingegangen werden.

6.3 Die PRAM^{dt}

Anleihen aus der SB-PRAM für die PRAM^{dt}

Das Konzept der parallele Registermaschine muss um das Analyseinstrumentarium für diese Arbeit erweitert werden. Dieses besondere Modell wird als PRAM^{dt} bezeichnet. Da die *SB-PRAM* eine gut erforschte, praktisch realisierte und dokumentierte Variante der PRAM inklusive lauffähigem Prototype darstellt, wird die hier eingeführte PRAM^{dt} star-

ke Anleihen aus diesen Arbeiten enthalten. Ausschlaggebend für die Wahl zur SB-PRAM im Gegensatz zum ursprünglichen PRAM Modell [FW78] waren insbesondere die Multipräfixoperationen, welche eine Realisation von Sperren und Barrieren ermöglicht, und ein mächtigerer Befehlssatz (z. B. *LOADINDEX*). In Exkurs 6.3.1 werden die Unterschiede der SB-PRAM zu der hier definierten Maschine betrachtet[8].

6.3.1 *Aufbau der PRAM^{dt}*

18 PRAM^{dt}

Die PRAM^{dt} besteht aus:

- ❑ **n Prozessoren**
  Einer unendlichen, abzählbaren Menge von berechnenden Elementen, den Prozessoren $P_0, P_1, \ldots, P_{n-1}$. Jeder Prozessor hat einen unendlichen, lokalen Speicher. Der Aufbau eines Prozessors wird in Definition 19 beschrieben.
- ❑ **Speicher**
  Neben dem unendlichen, lokalen Speicher jedes Prozessors, sind folgende Speicher vorhanden:
  - ❑ **Gemeinsamer, globaler Speicher**
    Ein unendlicher, gemeinsamer Speicher, auch Hauptspeicher genannt [Sch95, vgl. S.5 ff]. Alle Prozessoren können auf diesen mittels eines gemeinsamen Adreßraums lesend und schreibend zugreifen [Sch95, vgl. S. 5, Definition 2.1].
  - ❑ **Programmspeicher**
    Ein spezieller Speicher, in dem das Programm der PRAM gespeichert ist [Kes; GRR95].
- ❑ **Globaler Takt- oder Schrittgeber**
  Eine globale Uhr steuert die PRAM^{dt}. Die Uhr fungiert als Takt- bzw. Schrittgeber. Ein Schritt wird nach Scheerer [Sch95, S. 5, Definition 2.1] auch als Runde oder Zyklus bezeichnet
  - Im Gegensatz zur SB-PRAM (bzw. normalen PRAM), bei der jeder Prozessor in einer Runde entweder eine lokale Operation oder einen Zugriff auf den Hauptspeicher (uniformer Speicherzugriff) ausführt [Sch95, S. 5, Definition 2.1], können diese hier durch das Analyseinstrumentarium mehrere Zyklen umfassen.
  - Analog zu Abschnitt 6.2.3 steht am Anfang eines prolongierten Berechnungsschrittes ein Lesen, am Ende ein Schreiben, was die Analyse vereinfacht.

[8] siehe Exkurs 6.3.1, Seite 106

19 Prozessor der PRAM^{dt}

Der Prozessor einer PRAM^{dt} besteht aus unendlichen Registern. Diese gliedern sich wie folgt:

- **Lokaler Speicher**

  Jeder Prozessor besitzt einen unendlichen, lokalen Speicher. Dieser lokale Speicher wird mit $R_{j,n}$ oder $L_j(Adresse)$ angesprochen, wobei j die eindeutige Nummer des Prozessors, n die Nummer des Registers.

- **Spezialregister**

  Die PRAM^{dt} enthält folgende Spezialregister:

  - IC_i

    Der Befehlszähler oder Instruction Counter (IC_i) gibt den nächsten zu bearbeitenden Befehl des Prozessors an.

  - *Akkumulator*

    Der Akkumulator (Acc_i) ist ein spezielles Register der PRAM^{dt}.

  - *Stackzeiger*

    Der Stackzeiger (SP_i) ist ein spezielles Register, der auf die Spitze des Stacks zeigt. Wenn ein klassisches „*Push*“ oder „*Pop*“ durchgeführt wird, wird der Stackzeiger inkrementiert oder dekrementiert.

    Der Stackzeiger wurde wegen des „*Call*“-Befehls zum Aufruf von Unterprogrammen in den Befehlssatz der PRAM^{dt} integriert.

  - MP_i

    Ein Hilfsregister zum Zwischenspeichern des Akkumulators für Multipräfixoperationen. Dieses Register wird nur intern verwandt und ist nur mittels Multipräfixoperationen implizit ansteuerbar.

  - PSW_i

    Der Statusregister des Prozessor. Die dafür gebräuchliche Abkürzung PSW beruht auf dem “Processor Status Word”, ein normalerweise 16 Bit Register (Word) in dem der Status des Prozessors angegeben wird.

    Das Statusregister wird in Definition 20 detailiert beschrieben.

20 Statusregister des Prozessors (PSW)
Das Statusregister enthält mindestens folgende Informationen:

- *INDEX*
  Die Nummer des Prozessors. Diese kann mit mittels des Maschinenbefehls *LOADINDEX* abgefragt werden.
  Da die Anzahl der Prozessoren in der PRAM^{dt} unbegrenzt sind, muss demnach auch das Statusregister unendliche Zahlen speichern können. Da sich die Abkürzung PSW auf ein 16 Bit Register bezieht, sich aber etabliert hat, wird diese beibehalten.
- *Inaktiv*
  Nach Beendigung der Berechnung dieses Prozessors oder bei der Initialisierung der PRAM^{dt} wird ein Bit (Flag) gesetzt, dass dieser Prozessor inaktiv ist und rekrutiert werden kann.
  Ein *FORK* Befehl[a] kann einen Prozessor aktivieren und setzt dieses Bit zurück.
  Ein *HALT* Befehl beendet eine Berechnung und setzt dieses Bit.
  Anstelle eines Inaktiv-Flag kann natürlich auch ein Aktiv-Flag in das Prozessorstatusregister integriert werden. Die angegebenen Befehle setzen dann das Bit invers.

[a] siehe Tabelle 6.3.2, Seite 107

Der Zugriff auf ein Objekt des Statusregisters wird im Folgenden, analog zu vielen objektorientierten Sprachen, wie Java [Gos+05] oder C++ [Str00], mit einem . gekennzeichnet.
So bezeichnet ein $Acc_i \leftarrow PSW_i.INDEX$ das Kopieren der Prozessornummer aus dem Statusregister in den Akkumulator des i.-Prozessors.

6.3.2 Der Befehlssatz der PRAM^{dt}

Mikrobefehle

Die Mikrobefehle orientieren sich bei der PRAM^{dt} an einer Akkumulator Architektur [Å00; BS76], sie hat einen an die SB-PRAM angelehnten, jedoch reduzierten Befehlssatz [Sch95, S. 20,21].

Exkurs 6.3.1 Unterschiede zum Befehlssatz der SB-PRAM

- Die Befehle *ADC*, *SBC*, *LSL*, *LSR*, *ASL*, *ASR*, *ROL*, *ROR*, *ROCL*, *ROCR*, *MUL*, *RM* der Compute Befehle [Sch95, S. 20,21][Kes] wurden nicht übernommen. Die Funktion dieser Befehle können aber mit mehreren Befehlen der PRAM^{dt} emuliert werden.
- Die PRAM^{dt} enthält im Gegensatz zur SB-PRAM keine Floatingpoint Operationen [Sch95, S. 69ff][Kes].

Instruktion		Funktion
LOAD	operand	Führt die Operation mit
STORE	operand	Hilfe des Akkumulators aus.
LOADK	operand	
ADD	operand	
SUB	operand	
AND	operand	
OR	operand	
NAND	operand	
XOR	operand	
JUMP	label	Der Befehlszähler wird auf label gesetzt.
JZERO	label	
CALL	label	Modulaufruf
RET	label	Rückkehr
HALT		Stoppt den Prozessor
LOADINDEX		Operationen für parallele Berechnung
FORK	label	
WRITE	Adresse	Zugriff auf gemeinsamen Speicher
READ	Adresse	
MPADD	Adresse	Multipräfixoperationen
MPOR	Adresse	
MPAND	Adresse	
MPMAX	Adresse	
NOP		Keine Operation

Tabelle 6.4: Befehlsatz der PRAM^{dt}

- Die Befehle *GETSR*, *GETHI* und *PUTSR* wurden nicht in den Befehlssatz übernommen [Sch95, S. 20].
- Erweitert wurden die Mikrobefehle um den Befehl *FORK*, der bei der SB-PRAM nicht als Mikrobefehl realisiert ist [Sch95, S. 7off].
- Schreib- und Leseoperationen auf den lokalen sowie auf den globalen Speicher haben im Gegensatz zur SB-PRAM unterschiedliche *Mnemonics* (*LOAD*/*STORE* bzw. *READ*/*WRITE* gegenüber *LD* und *ST*), was die Unterscheidung erleichtert.
- Der Mikrobefehl *GETNR* wurde *LOADINDEX* genannt [Sch95, S. 20].

6.3.2.1 Konventionen der Beschreibungssprache

- C: Eine Konstante ($C \in \mathbb{N}$).
- R: Ein Identifikator für ein beliebiges Register, z.B. PSW, Acc, $L_i(lA)$. Der Identifikator R wird für eine prägnante Beschreibung der $PUSH$ und POP Operation benötigt.
- Acc_i: Der Akkumulator des i.-Prozessors.
- IC_i: Der Befehlszähler des i.-Prozessors.
- PSW_i: Das Prozessorstatuswort des i.-Prozessors.
- SP_i: Der Stackpointer des i.-Prozessors.
- $label$: Eine eindeutige Adresse ($label \in \mathbb{N}$) zur Identifikation einer Speicherzelle im Programmspeicher.
- lA: Eine eindeutige Adresse ($lA \in \mathbb{N}$) zur Identifikation einer Speicherzelle im lokalen Speicher des jeweiligen Prozessors.
- gA: Eine eindeutige Adresse ($gA \in \mathbb{N}$) zur Identifikation einer gemeinsamen, globalen Speicherzelle.
- $L_i(lA)$: Der Inhalt der Speicherzelle mit der Adresse lA des lokalen Speichers des i.-Prozessors.
- $M(gA)$: Der Inhalt der Speicherzelle mit der Adresse gA des gemeinsamen Speichers der PRAM^{dt} .
- $S(SP_i)$: Der Inhalt der Speicherzelle auf den der Stackzeiger des i.-Prozessors zeigt. $S(SP_i)$ und $L_i(SP_i)$ sind gleichbedeutend.
- $Acc_i \leftarrow (lA)$: Der Inhalt der lokalen Speicherzelle mit der Adresse lA wird in den Akkumulator des i.-Prozessors kopiert.
- $R \leftarrow R + 1$: Der Inhalt des Registers wird inkrementiert.
- **Befehlsformat:** Hier wird angegeben, wie die Mikrobefehle im Programmspeicher der PRAM^{dt} gespeichert sind. Jeder Mikrobefehl ist durch eine Nummer codiert, lokale Adresse, globale Adresse etc. sind nicht abgekürzt.

Die Definitionen der einzelnen Mikrobefehle werden im Anhang als Nanobefehle angegeben. Für jeden einzelnen Nanobefehl ist angegeben, in welcher der drei Phasen (Lesen, lokale Operation und Schreiben) er vom Prozessor ausgeführt wird.

21 Nanobefehl
Ein Nanobefehl ist eine elementare Hardwareoperation, die durch ein entsprechendes Steuersignal ausgelöst wird (z.B. Spannung an einem Multiplexer etc.) [Gil93, vgl. S. 19, S. 73].

Mnemonik	Operanden	Semantik	Mikrobefehl
AND	Acc_i, lA	$Acc_i \leftarrow Acc_i \wedge L_i(lA)$	Def. 38, S. 293
OR	Acc_i, lA	$Acc_i \leftarrow Acc_i \vee L_i(lA)$	Def. 39, S. 293
NAND	Acc_i, lA	$Acc_i \leftarrow Acc_i \overline{\wedge} L_i(lA)$	Def. 40, S. 293
XOR	Acc_i, lA	$Acc_i \leftarrow Acc_i \oplus L_i(lA)$	Def. 41, S. 293
LOAD	Acc_i, lA	$Acc_i \leftarrow L_i(lA)$	Def. 42, S. 293
STORE	Acc_i, lA	$L_i(lA) \leftarrow Acc_i$	Def. 43, S. 294
LOADK	Acc_i, C	$Acc_i \leftarrow C$	Def. 44, S. 294
ADD	Acc_i, lA	$Acc_i \leftarrow Acc_i + L_i(lA)$	Def. 45, S. 294
SUB	Acc_i, lA	$Acc_i \leftarrow Acc_i - L_i(lA)$	Def. 46, S. 294
NOP	/	/	Def. 47, S. 294
JUMP	*label*	$IC_i \leftarrow label$	Def. 48, S. 294
JZERO	Acc_i, *label*	*if* $Acc_i \leq 0$ *then* $IC \leftarrow label$	Def. 49, S. 294
PUSH	R	$S(SP_i) \leftarrow R; SP_i + 1$	Def. 50, S. 294
POP	R	$R \leftarrow S(SP_i); SP_i - 1$	Def. 51, S. 295
CALL	*label*	$PUSH\ PSW_i; PUSH\ IC_i; IC_i \leftarrow label$	Def. 52, S. 295
RET	/	$POP\ IC_i; POP\ PSW_i$	Def. 53, S. 295
HALT	/	$PSW_i.inaktiv \leftarrow 1$	Def. 54, S. 295
LOADINDEX	Acc_i	$Acc_i \leftarrow PSW_i.index$	Def. 55, S. 295
FORK	*label*	Siehe Definition	Def. 56, S. 296
READ	Acc_i, gA	$Acc_i \leftarrow M(gA)$	Def. 58, S. 296
WRITE	Acc_i, gA	$M(gA) \leftarrow Acc_i$	Def. 57, S. 296
MPADD	$Acc_a - Acc_m, gA$	Siehe Definition	Def. 59, S. 297
MPAND	$Acc_a - Acc_m, gA$	Siehe Definition	Def. 60, S. 297
MPOR	$Acc_a - Acc_m, gA$	Siehe Definition	Def. 61, S. 298
MPMAX	$Acc_a - Acc_m, gA$	Siehe Definition	Def. 62, S. 298

6.3.3 Zeitpunkte der zeitlichen Variation

Realisation

Im folgenden Abschnitt wird diskutiert, wie die Analyseinstrumentarien, also die unterschiedlichen Ausprägungen des Analyseinstrumentariums, realisiert werden können. Das Analyseinstrumentarium verändert zeitliche Abläufe. Das hier eingeführt Maschinenmodell basiert auf diskreten Schritten:

- jeder Zyklus dauert gleich lange.
- jeder aktivierte Prozessor verarbeitet jede Runde einen Maschinenbefehl aus dem Programmspeicher.

Um einen einzelnen Befehl zeitlich zu variieren, muss dieser auf die Zyklen aufgeteilt oder mehrere Befehle müssen in einem Zyklus abgearbeitet werden. Der nächste Abschnitt, Abschnitt 6.3.4 behandelt die Prolongation zwischen den einzelnen Phasen.

- Zurückführen lassen sich das simulierte Optimieren und die Retardation auf die Prolongation.
- Die Prolongation wurde als das Verlängern eines Ablaufs definiert.[9] Der Schritt, der den Zustand des Maschinenmodells bzw. der PRAM^{dt} ändert (der eigentliche Effekt) muss am Schluß sein, wenn der Zustandsübergang in einem diskreten Schritt erfolgt. Es muss also vor dem eigentlich Befehl „Zeit eingefügt“ werden.

22 Zeitpunkt der Prolongation
Zusätzliche Zeit muss **VOR** dem Zustandsübergang (der Änderung eines Zustandes) eingefügt werden.

6.3.4 Realisation zwischen einzelnen Befehlsphasen

Jede Befehlsausführung der PRAM^{dt} besitzt drei, aufeinanderfolgende Phasen: Lesen, lokale Operation und Schreiben (vgl. Abschnitt 6.2.3, Seite 95). Die atomaren Mikrobefehle der PRAM^{dt} sind demnach wieder (zeitlich) quantifiziert. Man könnte, anlehnend an die Kernphysik bzw. Atomphysik, von Befehls-Quanten sprechen.

Befehls-Quanten

Nun wäre es möglich, beispielsweise zwischen Lesephase und der Phase der lokalen Operationen eine gewissen Zeitspanne, z. B. einen diskreten Zyklus, einzufügen. Damit wäre der Mikrobefehl prolongiert[10], er würde um einen diskreten Zyklus länger dauern. Andererseits könnte vor der Lesephase eines Mikrobefehls eine gewissen Zeitspanne eingefügt werden, somit könnte eine Retardation[11] realisiert werden.
Dieser Ansatz hat jedoch Nachteile:

Realisation des Analyseinstrumentariums

[9] siehe Definition 8, Seite 66
[10] siehe Definition 8, Seite 66
[11] siehe Definition 9, Seite 67

- Dieser Ansatz führt bei Multipräfixoperationen zu Inkonsistenzen (siehe Gegenbeispiel 6.3.1).
- Spezielle Hardware muss für diesen Ansatz zur Realisation des Analyseinstrumentariums zur Verfügung stehen. In Kapitel 7 wird mittels Instrumentierung oder eingefügter Zeit in einer Virtualisierungslösung eine Experimentierumgebung geschaffen. Bei beiden Lösungsmöglichkeiten wäre eine Implementierung der Prolongation zwischen den Phasen sehr schwer realisierbar.
- Bei Analysen von Systemen mit höhergranularen Einheiten (z. B. Modulen oder Subsystemen) kann die hier entwickelte Theorie nicht verwandt werden.

Beispiel 6.3.1 Inkonsistente Multipräfixoperationen
Sei $\odot$ eine assoziative Operation. Eine Multipräfixoperationen ist wie folgt definiert[12].

23 Multipräfixoperationen

$$
\begin{array}{l}
\textit{For } k \in \{PSW_a.INDEX, \ldots, P_m.PSW.INDEX\} \\
\qquad \left.\begin{array}{l} MP_k \leftarrow ACC_k \\ ACC_k \leftarrow M(gA) \end{array}\right\} \text{Lesen} \\
\textit{For } k \in \{PSW_a.INDEX, \ldots, P_m.PSW.INDEX\} \\
\qquad \left.\begin{array}{l} IC_k \leftarrow IC_k + 2 \end{array}\right\} \text{lokale Operation} \\
\textit{For } k \in \{PSW_a.INDEX, \ldots, P_m.PSW.INDEX\} \\
\qquad \left.\begin{array}{l} M(gA) \leftarrow M(gA) \odot MP_k \end{array}\right\} \text{Schreiben}
\end{array}
$$

Ein Prozessor, o.B.d.A. P_j, wird analog dem obigen Vorschlag zwischen der Lesephase und der Phase der lokalen Operation o.B.d.A. um $n \in \mathbb{N}$ diskrete Zyklen prolongiert.

24 Multipräfixoperationen

$$
\begin{array}{l}
\textit{For } k \in \{PSW_a.INDEX, \ldots, P_m.PSW.INDEX\} \\
\qquad \left.\begin{array}{l} MP_k \leftarrow ACC_k \\ ACC_k \leftarrow M(gA) \end{array}\right\} \text{Lesen} \\
\ldots
\end{array}
$$

- Die folgenden Befehle werden für Prozessor P_j um $n \in \mathbb{N}$ diskrete Zeitpunkte später ausgeführt.

$$
\begin{array}{l}
\ldots \\
\textit{For } k \in \{PSW_a.INDEX, \ldots, P_m.PSW.INDEX\} \\
\qquad \left.\begin{array}{l} IC_k \leftarrow IC_k + 2 \end{array}\right\} \text{lokale Operation} \\
\textit{For } k \in \{PSW_a.INDEX, \ldots, P_m.PSW.INDEX\} \\
\qquad \left.\begin{array}{l} M(gA) \leftarrow M(gA) \odot MP_k \end{array}\right\} \text{Schreiben}
\end{array}
$$

- Das Ergebnis der Multipräfixoperation wird verfälscht, der Operand Acc_j fließt nicht in die Berechnung ein. Die Multipräfixoperation führt mittels eines veralteten Wert die Operation durch.

[12] siehe Definition 59, Seite 297

- Der Inhalt der Speichzelle $M(gA)$ kann zum Zeitpunkt des Schreibens durch andere globale Operationen[13] inzwischen verändert worden sein.
- Insbesondere können durch die Prolongation zwischen Lesephase und der Phase der lokalen Operation keine Sperren (locks) oder Barrieren (barriers) implementiert werden, die „unempfindlich“ gegenüber dem Analyseinstrumentarium sind, was ein chaotisches Programmverhalten zur Folge hat[14].

Das Gegenbeispiel zeigt, dass die zeitlichen Variationen atomar mindestens auf Befehlsebene durchgeführt werden müssen.

6.3.5 *Prolongation durch eingefügte Zeit*

25 Prolongation in der PRAM^{dt}
Eine Prolongation bezeichnete das Einfügen von mindestens einem diskreten Zyklus, in der kein Zustand der PRAM^{dt} verändert wird, **vor** der Lesephase eines Prozessors und der Ausführung des (prolongierten) Mikrobefehls.

Instrumentierung

Vorteilhaft mit dieser Definition ist, dass eine Prolongation durch zusätzliche Befehle, die den Zustand der PRAM^{dt} nicht verändern, realisiert werden kann. Beispielsweise durch das Einfügen der No Operation Mikrobefehle (NOP). Dieser Ansatz kann insbesondere bei einer Prolongation auf Codeebene[15] mittels Instrumentierung leicht realisiert werden. Für eine Diskussion der Vor- und Nachteile siehe Abschnitt 7.1.8.3 auf Seite 161.

Virtualisierung

Andererseit kann ein beliebiges Δt [16] an zusätzlicher Zeit vor dem Befehl (bei einer Prolongation) eingefügt werden. Dieser Ansatz wird bei einer Prolongation auf Hardwareebene[17] mittels Virtualisierung möglich.

In Kapitel 7 werden beide Methoden für eine praktische Realisation der Experimentierumgebung gezeigt.

Der Wirkzusammenhangs wird durch das Einfügen zusätzlicher NOPs gezeigt. Einerseits müsste die PRAM^{dt} so erweitert werden, dass die Prozessoren analog zu Kapitel 5 einen „leeren“ Schritt durchführen können, bzw. Prozessoren für einen bestimmten Zeitbereich langsamer sind. Dies würde das Modell sehr aufblähen. Ist dieses Maschinenmodell verfügbar, so kann der hier dargestellte Beweis übernommen werden, ein NOP entspricht dann einem solchen „leeren“ Schritt.

[13] siehe Abschnitt 6.4.2, Seite 115
[14] siehe Abschnitt 6.4.6, Seite 121
[15] siehe Abschnitt 7.1, Seite 139
[16] Δt ist eine physikalische Zeiteinheit beliebiger Messgranularität. Hier muss gelten $\Delta t \in \mathbb{Q}$, da $\Delta t \in \mathbb{R}$ nicht realisierbar ist, bzw. in dieser Arbeit nicht betrachtet wird.
[17] siehe Abschnitt 7.3.4, Seite 171

26 Retardation
Unter einer Retardation kann bei einer PRAM das Verzögern (die Prolongation) bestimmter Mikrobefehle verstanden werden.

Beispiel 6.3.2 Retardation eines Moduls bei der PRAM^{dt}
Eine Retardation eines Moduls/Unterprogramm/Routine/Funktion kann beispielsweise durch eine Prolongation des dem *CALL* nachfolgenden Befehls (der Subroutine) realisiert werden.

Beispiel 6.3.3 Retardation eines Hardwarezugriffs bei der PRAM^{dt}
Eine Retardation eines Hardwarezugriffs kann beispielsweise durch eine Prolongation eines entsprechenden *WRITE* oder *READ* Befehls realisiert werden.

27 Simuliertes Optimieren
Ein simuliertes Optimieren eines Prozessors P_i bezeichnet eine Prolongation aller anderen Prozessoren P_j mit $j \in \{0, \ldots, i-1\} \cup \{i+1, \ldots, n-1\}$

Somit können alle drei Operationen des Analyseinstrumentariums (Prolongation, Retardation und simuliertes Optimieren) auf die Prolongation zurückgeführt werden:

- **Retardation – Verschieben der Start- bzw. Endzeitpunkte**
  Dies entspricht einer Prolongation nur bestimmter Mikrobefehle.

- **Simuliertes Optimieren – (relative) Laufzeitverkürzung**
  Entspricht einer Prolongation aller Prozessoren bis auf den zu untersuchenden bzw. simulierenden Prozessor.

Folgendes muss für die PRAM^{dt} noch gezeigt werden:

- **Die Korrektheit im parallelen Fall.**

  Die Korrektheit sequentieller Berechnungen wurde in Kapitel 5 gezeigt. Die PRAM^{dt} kann nun parallel Berechnungen durchführen. Wie beeinflusst die Prolongation das Ergebnis der Berechnung?
  Gezeigt wird dies in Abschnitt 6.4

- **Der Wirkzusammenhang zum Auffinden von Optimierungspotenzial.**

  Nach welchen Prinzipien kann Optimierungspotenzial in der PRAM^{dt} bzw. allgemein in einem System gefunden werden?
  Gezeigt wird dies in Abschnitt 6.5

6.4 Korrektheit bei einer Prolongation

Analog zu Kapitel 5 muss die *Interne Validität* (oder *Ceteris paribus-Validität* [Bou+04, S. 136]) für ein Experiment gezeigt werden. Im Gegensatz zur Turingmaschine aus Kapitel 5 ist die PRAM^{dt} ein paralleles Maschinenmodell, Uhren (insbesondere Uhren mit *wall clock time*) können integriert und realisiert werden.

Korrektheit

6.4.1 Klassifikation der Maschinenbefehle der PRAM^{dt}

Die PRAM^{dt} benutzt den gemeinsamen Speicher M zum Austausch von Informationen, ansonsten laufen die einzelnen parallel Prozessoren völlig autark, d.h. sie beinflussen sich nicht. Bei der PRAM^{dt} werden in dieser Arbeit zur besseren Darstellung zwei Kategorien von Befehlen klassifiziert: lokale Befehle und globale Befehle. Lokale Maschinenbefehle benutzen den lokalen Speicher, in der Schreib- und Lesephase der PRAM^{dt} werden keine Zustände (Register) verändert. Globale Maschinenbefehle greifen auf den globalen Speicher (z. B. mittels *WRITE*, *READ*, *MPADD*, etc.) zu, oder sind Befehle die die PRAM^{dt} zum Rekrutieren eines Prozessors (*FORK*) oder zum Stoppen eines Prozessors benötigt (*HALT*). Der Befehl *LOADINDEX* zum Laden des eindeutigen Identifikators des Prozessors ist ein Hybrid. Er „verhält“ sich wie ein lokaler Befehl (nur in der Lesephase der PRAM^{dt} werden Register verändert) für globale Informationen (die eindeutige Prozessoridentifikationsnummer). *LOADINDEX* wird im Folgenden als globaler Befehl klassifiziert.

lokale und globale Befehle

6.4.2 Korrektheit der lokalen Befehle der PRAM^{dt} bei einer Prolongation

Im Grunde sind diese Maschinenbefehle die Maschinenbefehle der Registermaschine. Insbesondere können beide Maschinenmodelle einander simulieren [LL92; SB84][HS01, S. 39, Theorem 2.5].

- **Der Akkumulator** eines Prozessors (o.B.d.A. i) Acc_i kann durch einen gesonderte Speicherbereich der in Kapitel 5 eingeführten Turingmaschine[18] dargestellt werden.
- **Der Speicher** bzw. die Register können analog durch das Band der Turingmaschine mit speziellen Trennzeichen realisiert/simuliert werden.
- **Die Mikrobefehle** *AND*, *OR*, *NAND*, *XOR*, *LOAD*, *STORE*, *ADD*, *SUB* der PRAM^{dt} sind kanonische Beispiele der berechenbaren Funktionen einer Turingmaschine und können leicht realisiert werden [Dav56, S. 173].

[18] siehe Definition 11, Seite 74

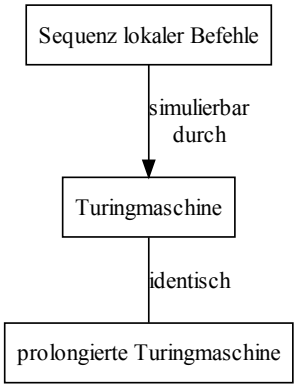


Abbildung 6.4: Simulation einer Sequenz lokaler Befehle der PRAM^{dt} mittels einer prolongierten Turingmaschine.

- **Ein *NOP* Befehl** kann als Einzelprolongation der Turingmaschine gesehen werden. Die Zustände der Turningmaschine oder der PRAM^{dt} werden nicht verändert.
- **Die Sprünge** (*JUMP* und *JZERO*) bzw. die Methodenaufrufe und Rückkehr (*CALL* und *RET*) können mittels spezieller Zustände und einen bestimmten Speicherbereich (für den Stack) der Turingmaschine realisiert bzw. simuliert werden [HS01, S. 40].

Ein Programm bzw. ein Codebereich bestehend aus lokalen Befehlen kann durch eine Turingmaschine realisiert werden. Insbesondere auch durch die in Kapitel 5 eingeführten, prolongierte Turingmaschine.

■

Analog kann dies wie folgt bei der PRAM^{dt} nachvollzogen werden:
Sei μ ein lokaler Befehl aus Tabelle 6.4.2. μ kann nur aus dem lokalen Speicher lesen oder in diesen Schreiben. Der Inhalt des lokalen Speichers L_i kann von keinem anderen Prozessor $P_k \in \{P_0, \ldots, P_{n-1}\} \setminus P_i$ zu keinem Zeitpunkt mittels eines Mikrobefehls verändert werden. Eine Prolongation eines lokalen Maschinenbefehls μ verursacht demnach keine Seiteneffekte bei den anderen Prozessoren der PRAM^{dt}.

■

Maschinenbefehl
AND
OR
NAND
XOR
LOAD
STORE
ADD
SUB
LOADK
JUMP
JZERO
PUSH
POP
CALL
RET
NOP

Tabelle 6.5: Lokale Operationen (lokaler Speicher) der PRAM^{dt}

6.4.3 *Korrektheit der globalen Befehle der PRAM^{dt} bei einer Prolongation*

Maschinenbefehl
LOADINDEX
HALT
FORK
WRITE
READ
MPADD
MPOR
MPAND
MPMAX

Tabelle 6.6: Globale Operationen (gemeinsamer, globaler Speicher) der PRAM^{dt}.

globale Befehle

Die globalen Befehle sind in Tabelle 6.4.3 angegeben. Diese Maschinenbefehle sind spezielle Maschinenbefehle für die Parallelität bzw. greifen auf den globalen Speicher zu. Hier ist es wichtig zu unterstreichen, dass die Prozessoren nur über diesen gemeinsamen Speicher kommunizieren können. Bei einer Prolongation bei gleichzeitigem Zugriff kann es zu Konflikten kommen. Ein angedachtes Schreib- und Lesepaar von Prozessoren kann unter der Prolongation gestört werden. Beispielsweise wird der geschriebene Wert eines Prozessors durch einen anderen Prozessor überschrieben wird bevor der prolongierte (und eigentlich angedachte) Prozessor diesen Wert lesen kann.

Exkurs 6.4.1 diskutiert die in diesem Kontext wichtige Abstraktion der PRAM als Modell gegenüber realer asynchroner Systeme.

Exkurs 6.4.1 Asynchronität und die parallelen Registermaschine

Durch die Abstraktion des PRAM Modells werden wesentliche Eigenschaften wie Synchronisationsprobleme, Speicherthematiken, die Zuverlässigkeit der Maschine und Verzögerungen, die durch Kommunikation entstehen, ausgeblendet. Somit ist es möglich, sich durch diese Abstraktion ganz auf den Entwurf von Algorithmen zu konzentrieren [CZ89].

Cole und Zajicek [CZ89] erweitern in der Arbeit „*The APRAM: Incorporating Asynchrony into the PRAM Model*“ die klassische PRAM um Asynchronität [CZ89, S. 169]. Bei der asynchronen, parallelen Registermaschine (*APRAM - Asynchronus Parallel Random Access Machine*) werden die Kosten für notwendige Synchronisation explizit betrachtet.

Im PRAM Modell hat eine implizite Synchronisation [FW78]. Hier wird angenommen, dass kein Prozess mit der $i + 1$ Anweisung fortfährt, so lange nicht alle anderen Prozesse die Anweisung i verarbeitet haben [CZ89, S. 169]. Dies vereinfacht zwar den Entwurf paralleler Algorithmen, jedoch müssen diese bei einer Implementierung oft wegen asynchroner Operationen umkonstruiert werden [CZ89, S. 169].

Gibbons [Gib89] schlug die „Phase PRAM“ vor, eine Erweiterung bei der die Berechnung in Phasen zerlegt wird [Gib89]. Die einzelnen Prozessoren arbeiten asynchron während einer Phase und synchronisieren sich am Ende von dieser [Gib89]. Andere Vorschläge für asynchrone parallele Registermaschine wurden von Kanellakis und Shvartsman [KS89] [KS89] und Kedem, Palem und Spirakis [KPS90] [KPS90] gemacht.

Wie von Culler u. a. [Cul+93] dargelegt und von anderen Autoren unterstützt wird, muss ein theoretisches Konzept für parallele Algorithmen Asynchronität inkooperieren. Cole und Zajicek [CZ89, S. 170] nennen zwei Gründe für die Asynchronität einer allgemeinen, parallelen Maschine:

❑ **Swapping durch das Betriebssystem**

Ein Betriebssystem, welches auf einer solchen Maschine läuft, sollte die Fähigkeit haben, bestimmte Prozesse auszulagern, um wichtigere bzw. höher priore Aufgaben abzuarbeiten. Damit hätte eine implizite Synchronisation, wie sie im PRAM Modell propagiert wird, ernsthafte „*performance penalities*“, wie Cole und Zajicek [CZ89, S. 170] diesen Effekt nennen[19].

[19] siehe Exkurs 6.4.2, Seite 119

❑ **Ungleichmäßige Lastverteilung (load balancing)**
Durch dynamische Prozesse kann die Last auf den einzelnen Prozessoren nicht gleichmäßig verteilt sein. Bei einer impliziten Synchronisation, wie sie im PRAM Modell propagiert wird, ist auch hier mit *„performance penalities“* zu rechnen.

Cole und Zajicek [CZ89, S. 170] stellen fest, dass das PRAM Modell als Analysemodell für die asynchrone Programmierung ungeeignet ist. Es wird auch erwähnt, dass jeder PRAM Algorithmus durch das Einfügen von entsprechende Barrieren („barrier“) nach jeder Anweisung in einen Algorithmus für die APRAM konvertiert werden kann. Dies würde jedoch die Geschwindigkeit des Algorithmus auf den langsamsten Prozess verlangsamen [CZ89, S. 170].

Exkurs 6.4.2 Performance penalities und das Analyseinstrumentarium

Interessant ist im Kontext dieser Arbeit die von Cole und Zajicek [CZ89, S. 170] erwähnten „performance penalities“. Sie bezeichnen damit ein Warten, insbesondere ein Warten von $n-1$ Prozessoren auf einen Prozessor, das langsamste Glied der Kette, der durch das Swapping oder die ungleichmäßige Lastverteilung noch nicht zur Verfügung steht.

Der Ansatz dieser Arbeit, die Prolongation als Analyseinstrumentarium, konterkariert genau dies. Hier wird bewusst, in einer dedizierten Art und Weise, diese „performance penalities“ als Analyseinstrument genutzt.

Bewusst werden Wartezeiten eingefügt um mit Hilfe der entstehenden „performance penalities“ Synchronisation zu entdecken.

Diese Synchronisationsmechanismen sind dem Programmierer nicht a priori bekannt. So kann gleichzeitiger Ressourcenzugriff, der synchronisiert werden muss, auf Code Ebene oder durch Profiling nur sehr schwer entdeckt werden. Insbesondere ergibt sich das Problem bei der Verwendung bzw. Wiederverwendung von Komponenten, deren Code oder Funktionalität nicht explizit zur Verfügung steht.

Analog zu dem Vorschlag von Cole und Zajicek [CZ89][20] werden die Vorarbeiten von Lamport [Lam78] in dieser Arbeit benutzt, jedoch zu einem größeren Umfang als in [CZ89].

Exkurs 6.4.3 Time, Clocks, and the Ordering of Events in a Distributed System

Die Arbeit *„Time, Clocks, and the Ordering of Events in a Distributed System“* von Lamport [Lam78] ist eine der bedeutensden Arbeiten in der Informatik. Sie ist grundlegend in der Informatik für Prozesssynchronisation. 2000 wurde diese Arbeit mit dem *PODC Influential-Paper Award*, dem heutigen *Dijkstra Prize*, geehrt [Rajoo].

[20] siehe Exkurs 6.4.1, Seite 118

6.4.4 Logische Uhren nach Lamport

logische Uhren

Lamport führt in der Arbeit „Time, Clocks, and the Ordering of Events in a Distributed System“ [Lam78] logische Uhren ein. Demnach ist von einer abstrakten Perspektive eine Uhr nur eine Methode einem Ereignis eine Zahl (im Original als „number“ bezeichnet) zuzuweisen. Diese Zahl repräsentiert die Zeit, in der dieses Ereignis sich ereignet [Lam78, S. 559].

eine Uhr

System der Uhren

Lamport definiert eine Uhr C_i **für jeden Prozess** $Prozess_i$ als eine Funktion, die eine Zahl $C_i\langle a\rangle$ jedem Ereignis in diesem Prozess zuweist [Lam78, S. 559]. Das ganze System der Uhren wird repräsentiert durch die Funktion C, die jedem Ereignis b die Nummer $C\langle b\rangle$ zuweist, wobei $C\langle b\rangle = C_i\langle b\rangle$ wenn b ein Ereignis in Prozess $Prozess_i$ ist.

Counter

In der Arbeit wird ausgeführt, dass bis hier keine Annahme über die Relation der Zahlen zur physikalischen Zeit gemacht wurden. Die Uhren C_i sind demnach eher logische als physikalische Uhren. Diese können durch einen Counter (ohne reale Zeitmessung) realisiert werden.

6.4.5 Zeitmessung in der $PRAM^{dt}$

globale Uhr

Die $PRAM^{dt}$ ist durch **eine globale** Uhr getaktet.[21] Ein Ereignis ist im Anlehnen an Lamport [Lam78] im Kontext der $PRAM^{dt}$ die Ausführung eines Maschinenbefehls eines Prozessors. Ein Prozess ist die Ausführung des Programmcodes durch einen individuellen Prozessor. Nach Cole und Zajicek [CZ89, S. 171] und Brent [Bre74] kann dieses Modell auf Prozesse übertragen werden. Analog zu Lamport [Lam78] wird für die $PRAM^{dt}$ auch eine Funktion C definiert, die einem Ereignis eine Nummer zuweist.

28 Zeitpunkt des aktuellen Befehls $C(PRAM^{dt})$

Die $PRAM^{dt}$ ist durch eine globale Uhr getaktet. Während eines Taktes oder Ticks werden die drei seperaten, prozessorübergreifende Phasen Lesen, lokale Operation und Schreiben ausgeführt. Diese Uhr inkrementiert sich bei jedem Tick o.B.d.A. beginnend von 1. Die Funktion $C(PRAM^{dt})$ liefert den Wert der globalen Uhr bei der Ausführung eines Maschinenbefehls.

Beispiel 6.4.1 Prozessor als Uhr - innere Uhr

Durch entsprechenden Maschinencode kann ein seperater Prozessor als Uhr ausgebildet werden:

```
main:       LOADK Initialwert
            FORK Uhr
...         ...
```

[21] siehe Definition 18, Seite 104

```
UHR:        WRITE C
count:      LOADK 3
            MPADD C
            JMP count
...         ...
```

Die „äussere Uhr“ $C(PRAM^{dt})$ und eine „innere Uhr“, realisiert durch solch einen Code, können bei einer zeitlichen Variation unterschiedliche Werte aufweisen. Durch eine Prolongation muss der j.-Schritt eines Prozessors nicht mehr mit dem j.-Zeittakt der globalen Uhr korrelieren. Die Ausführung der Befehle sind für den prolongierten Prozessor gegenüber den anderen Prozessoren und der globalen Uhr verspätet.
Insbesondere durch diese als Programm realisierten Uhren ist es zur Beweisführung wenig sinnvoll, eine Funktion $C(PRAM^{dt})$ analog zu Lamport [Lam78, S. 559] individuell **für jeden Prozessor** zu definieren, die bei einer Prolongation „langsamer“ läuft, also die Nummer des abgearbeiten Befehls zurückliefert. Erlaubt man den Prozessor noch Zugriff auf seine eigene Uhr so würde es durch eine Prolongation zu einer Zeitdifferenz zwischen der durch das Programm realisierte Uhr und der des Prozessors kommen.

6.4.6 Prolongation von Sperren und Barrieren

Mittels des Multipräfixoperationen[22] können Sperren und Barrieren implementiert werden [KRR96, S. 127].

Beispiel 6.4.2 Sperre mittels MPMAX

- l bezeichnet eine Speicherzelle im globalen Speicher.

```
lock_init(l)        l=0;
lock(l)             while (MPMAX(l,1) ≠ 0);
unlock(l)           l=0;
```

Das Codebeispiel wurde zitiert nach [For+97, S. 5].

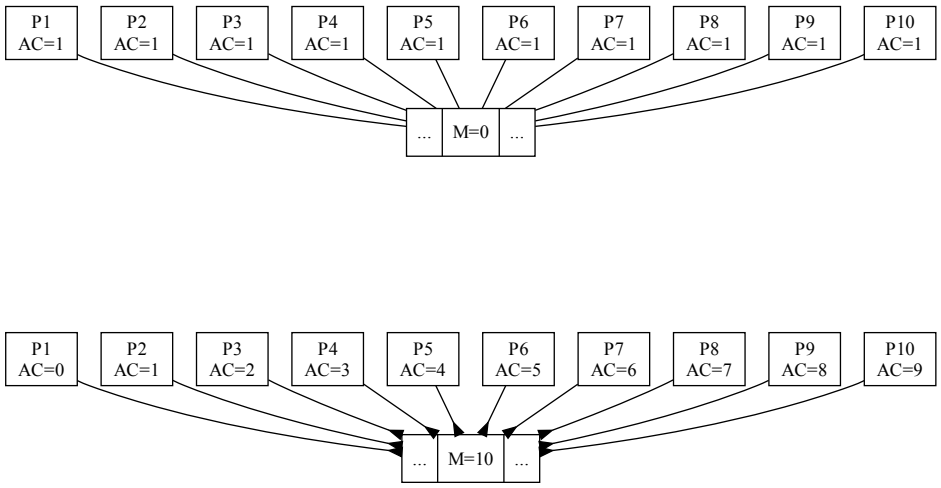
Beispiel 6.4.3 Sperre realisiert durch MPADD

- Eine Gruppe von 10 Prozessoren, in Abbildung 6.2.1 als P1, P2, P3, P4, P5, P6, P7, P8, P9, P10 bezeichnet, führen parallel mittels des *MPADD*-Mikrobefehls auf die gemeinsame Speicherzelle M des globalen Speichers die folgende test-and-set Operation aus.

```
test-and-set:   LOADK 1
                MPADD M
                JZERO lock
                JMP test-and-set
lock:           ...
unlock:         LOADK 0
                WRITE M
```

[22] siehe Definition 17, Seite 102

  - Jeder dieser Prozessoren hat den Wert 1 im Akkumulator.
  - Die globale, gemeinsame Speicherzelle *M* hat den Wert 0 gespeichert. Das Initialisieren der Sperre wird nicht parallel durchgeführt.



- Nach der Ausführung ergibt sich folgende Situation:
  - Prozessor P1 hat den Inhalt der Speicherzelle *M* übernommen, den Wert 0. Diesem Prozessor wird die Sperre gewährt.
  - Die gemeinsame, globale Speicherzelle *M* erhält als Wert die Summe aller Akkumulatoren von P1 bis P10 und den ursprünglichen Wert von *M*.
  - Die Prozessoren P2 bis P10 warten so lange (*JMP* `test-and-set`) bis P1 die Sperre am Label `unlock` verlassen hat, also *M* auf 0 setzt.
- Eine Prolongation des obigen test-and-set Codes bei den Prozessoren P2 bis P10 würde die Prozessoren P2 bis P10 dennoch warten lassen.
- Eine Prolongation von P1 würde P2 die Sperre gewähren. Es geschieht also eine Umordnung.

Erhalt der Semantik

Der Sinn dieser Sperre, nur einen Prozessor einen speziellen Codebereich abarbeiten zu lassen, kann durch eine Prolongation nicht verändert werden. Analog gilt dies für Barrieren [Kes97].
Konsequenterweise wird in Kapitel 10 die Prolongation zum Auffinden von Fehlern in Systemen durch unzureichende Synchronisationskonzepte benutzt.

6.5 Wirkzusammenhang

Zusammenhänge im System

Durch das Analyseinstrumenarium können Zusammenhänge in einem System erkannt werden, die auf gegenseitige Abhängigkeiten hindeuten. Intuitiv ist dieses Phänomen fassbar: gibt es irgendeine zeitliche Abhängigkeit zu einer Komponente, welche zeitlich variiert wurde, wird sich die Laufzeit einer beobachteten Komponente ändern.

Formalisierung durch Synchronisation

Dieser Effekt kann leicht verständlich und anschaulich mit der „Synchronisation“ erklärt werden. Hier darf „Synchronisation“ nicht (nur) im Kontext von nebenläufigen Prozessen wie Threads verstanden werden. Diese Synchronisation ist implizit gegeben, beispielsweise durch einen Unterprogrammaufruf, welches abgearbeitet werden muss, durch Verzweigungen im Kontrollfluß allgemein und durch einen parallelen Zugriff auf eine gemeinsame Ressource und Prozesse, die gegenseitig aufeinander warten müssen.

Abhängigkeit durch Synchronisation

Sind zwei Aktionssequenzen (Prozesse, Module, Hardwarekomponenten etc.) abhängig, müssen sie synchronisiert werden - was durch eine Prolongation erkannt werden kann, da gewartet werden muss. Sind zwei Aktionssequenzen unabhängig, zeigt eine Prolongation der einen Sequenz keine Reaktion in den anderen Aktionen.

Module als Black Box

Programmiersprachen basieren auf der Prämisse, dass jede Anweisung, jeder Ausdruck, jedes Modul eine *„black box“* darstellt [LDH03, S. 40]. Diese einzelnen Module sind verknüpft durch spezielle Schnittstellen [LDH03, S. 40]. Programmierer implementieren das System als eine Reihe dieser sequentiellen Module. Dieser statische Code kann beim dynamischen Ausführen Performanzprobleme zeigen. Der hier propagierte Anwendungsfall des Analyseinstrumentariums ist Optimierungspotenzial im System mittels des Experiments zu identifizieren.

- ❑ Anwendungsfall: Optimierungspotenzial im modularisiertes Softwaresystem finden
  - Das modularisierte Softwaresystem stellt eine „black box“ dar [LDH03, S. 40].
  - ❑ Die einzelnen Module, stellen eine „black box“ dar [LDH03, S. 40].
  - ❑ Die Ausführungszeit eines Software-Moduls M soll optimiert werden.
  - ❑ Andere Module sollen nach Wechselwirkungen untersucht werden.
- ❑ Das Experiment
  - ❑ Die Ausführungszeit anderer Module des Systems werden in ihrer Laufzeit variiert (prolongiert).
  - ❑ Das System wird ausgeführt.
  - ❑ Die Ausführungszeit des Moduls M wird gemessen.
  - ❑ Die Reaktionen dieser Variiationen sollen Rückschlüsse auf Optimierungskandidaten zulassen.

Dieser experimentelle Ansatz ist nicht immer einsetzbar. Gegenbeispiel 6.5.1 zeigt einen solchen Fall. In einem skizzierten *concurrent Pascal* Programm [Han75b; Han75a] misst ein Programm das Modul a aus und reagiert auf eine eventuelle Prolongation.

Beispiel 6.5.1 Gegenbeispiel
Das in Listing abgebildete Programm misst die Zeit eines Moduls a aus. a ist das Performanzproblem des Programmes. Wird a prolongiert (oder auch simuliert optimiert), zeigt das Programm kein Performanzproblem mehr.

```
procedure a;
var ta;
begin
  writln;   "braucht eine Zeiteinheit"
end;

begin
  t=:0;      "starte interne Uhr"
  a;         "führe a aus"
  Δt := t − System.Zeit;

  if Δt != 3 then
    return;          "a wurde verändert − ich beende mich"
  else
    while (1=1);  "a wurde nicht verändert − ich laufe endlos"
end.
```

Listing 6.1: Concurrent Pascal: Nicht messbar abhängig

Identifikation des Performanzproblems

Die Laufzeit des Programms hängt nur von der Prozedur a ab, Modul a ist das Performanzproblem des Programmes. Wird a jedoch prolongiert (oder auch simuliert optimiert) ist es plötzlich kein Performanzproblem mehr. Eine Messung kann nicht a als Ursache für die lange Laufzeit identifizieren.

nicht deterministisch

Analog gilt dies für Programme die stark nicht deterministisch sind, deren Laufzeit beispielsweise von einer Zufallszahl abhängt. In Kapitel 9 werden deshalb viele Experimente durchgeführt um mit dem Gesetz der großen Zahlen nicht deterministische Systeme zu analysieren. Nur so können Aussagen über diese getreffen werden. Es gilt jedoch:

❑ Das Analyseinstrumentarium ist „ehrlich“, es zeigt genau die Zusammenhänge an.
❑ Die Interpretation der Meßwerte kann „täuschen“. Es können immer Komponenten geschrieben werden, die sich unter einer Prolongation anders verhalten.
Zur eingängigen Darstellung des Wirkzusammenhangs wird das gegenüber eines „Referenzlaufes“ demonstriert.

29 Referenzlauf
Unter einem Referenzlauf bezeichnen wir eine Programmausführung mit der absolut gleichen Sequenz von Mikrobefehlen, bis auf die künstlich hinzugefügte Instrumentierung.

Da bei der PRAM^{dt} jeder Zyklus und damit jede Befehlsausführung gleich lange dauert müsste hier nicht die sehr starke Bedingung von der gleichen Sequenz eingefügt werden, sondern die Anzahl der ausgeführten Befehle muss gleich bleiben. Diese Definition eines Referenzlaufes ist zur Darstellung jedoch eingängiger.

6.5.1 Die Happend Before Relation

Die „Happened-Before“ Relation nach Lamport beschreibt Kausalordnungen bei verteilten Systemen.

30 Die Happened Before Relation „$\rightarrow$“ nach Lamport
Die Happened Before Relation $\rightarrow$ aus einer Menge von Ereignissen („*events*“) ist die kleinste Relation, die folgende Punkte erfüllt [Lam78, S. 559]:

1 Wenn a und b Ereignisse im selben Prozess sind und a vor b kommt, dann gilt $a \rightarrow b$.
2 Wenn a das Senden einer Nachricht eines Prozesses ist und b das Empfangen einer Nachricht eines anderen Prozesses ist, dann gilt $a \rightarrow b$.
3 Wenn $a \rightarrow b$ und $b \rightarrow c$ gilt, dann folgt daraus $a \rightarrow c$

Zwei Ereignisse sind gleichzeitig, wenn gilt:
$a \not\rightarrow b$ und $b \not\rightarrow a$.

kausaler Effekt

Lamport erwäht, dass die Definition von $a \rightarrow b$ bedeutet, dass das Ereignis a einen kausalen Effekt auf Ereignis b hat. Zwei Ereignisse sind hier gleichzeitig, falls es keinen kausalen Zusammenhang zwischen beiden gibt.

6.5.2 *Lokaler Wirkzusammenhang*

Lokaler Wirkzusammenhang

Wir betrachten nun den lokalen Wirkzusammenhang. Hierbei wird **Code aus dem Programmspeicher** der PRAM^{dt} auf **einem Prozessor** ausgeführt.

Satz 4 Lokaler Wirkzusammenhang

Mittels einer Prolongation kann, durch die messbare Zeitdifferenz gegenüber einem Referenzlauf, bestimmt werden, ob prolongierter Code ausgeführt wurde.

Punkte 1& 3

Wichtig sind für die Darstellung des lokalen Wirkzusammenhangs der Punkt 1 (im selben Prozess) und der Punkt 3 (die Transitivität) der „Happend-Before“ Relation:

1 Wenn a und b Ereignisse im selben Prozess sind und a vor b kommt, dann gilt $a \rightarrow b$.
3 Wenn $a \rightarrow b$ und $b \rightarrow c$ gilt, dann folgt daraus $a \rightarrow c$

Wirkzusammenhang

❑ **Gegeben ist eine Sequenz von Befehlen** im Programmspeicher (ein Programm).
❑ **Gezeigt werden soll der lokale Wirkzusammenhang**: durch die Prolongation können Zusammenhänge (der Komponenten) bei der Ausführung ausgemessen werden. Hierfür muss ein prolongierter Programmablauf mit dem Referenzlauf (einem nicht prolongierten Programmablauf) verglichen werden. Zur besseren Darstellung wird die Uhr des prolongierten Programmablaufs mit C' gekennzeichnet.

6.5.2.1 Nicht prolongierter Programmablauf

- Seien *start* und *stopp* zwei explizite, vorher definierte Stellen/Befehle/Ereignisse im Programmablauf, mit $C(start) < C(stopp)$.[23][24]
- Sei $Laufzeit = C(stopp) - C(start)$, $Laufzeit \in \mathbb{N}$
- Seien a, b zwei aufeinanderfolgende, lokale Befehle im Programmspeicher, wobei a kein Sprung, *Halt*-Befehl oder Return (*Ret*) ist.[25]

Es ist nicht bekannt, ob $a \rightarrow b$ gilt, d. h. es ist kein a priori Wissen darüber verfügbar, ob die Befehle a und b im Programmspeicher ausgeführt werden. Es ist nicht bekannt, ob dieses Modul die *Laufzeit* beeinflusst.

[23] Die Befehle sind im Programmspeicher, die Stelle im Programm wird implizit mit dieser Definition ausgeführt, durch die globale Uhr ist der ausführende Prozessor unrelevant. Es sind einfach zwei Zeitpunkte wie der Start und das Ende eines Programms.

[24] Damit ist $start \rightarrow stopp$ selbst ein Modul. Es wird als die Auswirkung des prolongierten Moduls $a \rightarrow b$ auf $start \rightarrow stopp$ gemessen.

[25] Ein Sprung kann durch eine einfach Fallunterscheidung behandelt werden.

6.5.2.2 Instrumentierung – Prolongation durch NOPs

Instrumentierung

Es werden nun n_1 bis n_j No-Operation Befehle (NOPs) nach a in das Programm, also in den Programmspeicher der PRAM^{dt}, eingefügt. Alle *label* mit einem Wert kleiner der Adresse von a werden beibelassen, alle anderen *label* um j inkrementiert.[26]

6.5.2.3 Prolongierter Programmablauf

Experiment

Durch den Referenzlauf ist nun die Sequenz der ausgeführten Befehle von *start* bis *stopp*, bis auf die eventuell ausgeführten, hinzugefügten *NOP*s $n_1 \dots n_j$, identisch[27] zum Referenzlauf, dem nicht prolongierten Programmablauf.[28]

- *start* und *stopp* sind die identischen Stellen/Befehle im Programmspeicher, wie beim nicht prolongierten Programmablauf.
- Wird a ausgeführt, gilt $a \to n_1 \to \dots \to n_j \to b$.
- Demnach gilt nach [3] $a \to b$.
- Da die Sequenz der ausgeführten Befehle des prolongierten und nicht prologierten Version (Referenzlauf) bis auf die hinzugefügten Befehle (der Prolongation) identisch ist, gilt: $C(a) = C'(a)$.
- Sei nun $Laufzeit' = C'(stopp) - C'(start)$, $Laufzeit' \in \mathbb{N}$
- Anhand von $Laufzeit'$ kann bestimmt werden ob $a \to b$ für $start \to stopp$ einen Optimierungskandidaten darstellt.

6.5.2.4 Messbarer Wirkzusammenhang

Analyse

Anhand von $Laufzeit'$ kann bestimmt werden, ob a (zwischen $C(start)$ und $C(stopp)$) ausgeführt wird (und damit einen Optimierungskandidaten für $start \to stopp$ darstellt).[29]

❑ **A)** $Laufzeit' = Laufzeit$

Die Laufzeitdauer der prolongierten und nicht prolongierten Version sind identisch.

a bzw. $a \to b$ wird nicht ausgeführt.

---

[26] Die Sprünge werden durch diese Instrumentierung angepasst.

[27] Da bei der PRAM^{dt} alle Befehle gleich lang sind, reicht es hier, dass die Anzahl der ausgeführten Befehle gleich bleibt.

[28] Das Problem hier ist, dass wir evtl. Code haben, der sich selbst ausmißt und auf Veränderungen reagiert bzw. Indeterminismus jeder Art, wie eine Randomfunktion. Siehe das Gegenbeispiel 6.5.1 auf Seite 124.

[29] Da a kein Sprung, *HALT*-Befehl oder Return (*RET*) ist, gilt bei einer Ausführung von a automatisch $a \to b$. Eine Erweiterung ist durch eine Fallunterscheidung möglich.

❑ **B)** $Laufzeit' = Laufzeit + j$
Die hinzugefügten No Operations gehen einfach in die Laufzeit ein.
a bzw. $a \rightarrow b$ wird einfach zwischen $start \rightarrow stopp$ ausgeführt.

❑ **C)** $Laufzeit' = Laufzeit + n \cdot j$
a bzw. $a \rightarrow b$ werden mehrfach (n-mal mit $n \in \mathbb{N}$) zwischen $start \rightarrow stopp$ ausgeführt.

6.5.2.5 Diskussion

Durch das Analyseinstrumentarium und einer gemessenen Zeitdifferenz (zu einem Referenzlauf) kann festgestellt werden, ob zwischen zwei Messpunkten (*start*, *stopp*) der prolongierte Code ausgeführt wird.
Durch die Transitivität (Punkt 3 der Happend-Before Relation) kann der Programmcode zwischen a und b auf mehr Befehle im Programmspeicher erweitert werden, es muss nur als Vorbedingung ein $a \rightarrow b$ überhaupt möglich sein.[30] $a \rightarrow b$ ist so nach Definition 3[31] ein Modul.

❑ **An einer Stelle** zwischen a und b kann prolongiert werden, durch $Laufzeit'$ kann herausgemessen werden, wie oft das Modul $a \rightarrow b$ ausgeführt wird.

❑ **An jeder Stelle** zwischen a und b kann prolongiert werden, hierbei kann jedoch durch Schleifen zwischen a und b anhand von $Laufzeit'$ nicht bestimmt werden, wie oft n_1 bis n_j ausgeführt wird, ausser jeder Befehl wird mit einer anderen Anzahl von NOPs prolongiert (b-adische Entwicklung, Logarithmen von Primzahlen).[32]
Das Optimierungspotenzial des Moduls $start \rightarrow stopp$ ergibt sich wie folgt:

❑ **A)** $Laufzeit' = Laufzeit$
Es ist kein Optimierungspotenzial im Modul $a \rightarrow b$ für $start \rightarrow stopp$ vorhanden. Der Code wird nicht ausgeführt.

❑ **B)**
Es ist wenig Optimierungspotenzial im Modul $a \rightarrow b$ für $start \rightarrow stopp$ vorhanden. Der Code wird einfach ausgeführt.

---

[30] Hierbei muss die Adresse von a nicht notwendigerweise kleiner sein als die Adresse von b.

[31] siehe Definition 3, Seite 19

[32] Vorteilhaft ist die zweitere Option bei Indeterminismus und bedingten Sprüngen, da bei der Ausführung von a hier nicht $a \rightarrow b$ gelten muss.

❑ **C)**

$a \rightarrow b$ wird mehrfach ausgeführt. Durch eine Hot Spot Optimierung bzw. loop-optimization kann *Laufzeit* verkleinert werden (*start* $\rightarrow$ *stopp* wird also optimiert), falls der Code nicht schon optimal ist.

6.5.3 Globaler Wirkzusammenhang

globalen Wirkzusammenhang

Wir betrachten nun den globalen Wirkzusammenhang. Hierbei wird **Code aus dem Programmspeicher** der PRAM^{dt} auf **mehreren Prozessoren** ausgeführt. Wir nehmen zur besseren Darstellung den Satz 5 der Definition 31 — nach ausreichend synchronisiertem Code — vorweg.

Satz 5 Globaler Wirkzusammenhang

Mittels einer Prolongation kann, durch die messbare Zeitdifferenz gegenüber einem Referenzlauf, bestimmt werden, ob, bei ausreichend synchronisiertem Code (siehe Definition 31 auf Seite 132), auf eine gemeinsame Ressource zugegriffen wird.

Punkte 2& 3

Wichtig sind für die Darstellung des globalen Wirkzusammenhangs der Punkt 2 (das Senden eines Prozessors/Prozesses und das Empfangen eines anderen Prozessors/Prozesses) und der Punkt 3 (die Transitivität) der „Happend-Before“ Relation:

2 Wenn a das Senden einer Nachricht eines Prozesses ist und b das Empfangen einer Nachricht eines anderen Prozesses ist, dann gilt $a \rightarrow b$.
3 Wenn $a \rightarrow b$ und $b \rightarrow c$ gilt, dann folgt daraus $a \rightarrow c$

❑ **Gegeben ist eine Sequenz von Befehlen** im Programmspeicher (ein Programm).
❑ **Gezeigt werden soll der globale Wirkzusammenhang**: durch das Analyseinstrumentarium (die Prolongation) können Zusammenhänge bei der Ausführung, durch die gemeinsame Ressourcennutzung, ausgemessen werden. Analog zum lokalen Wirkzusammenhang muss ein prolongierter Programmablauf mit einem Referenzlauf verglichen werden, um die Wechselwirkungen eingängig demonstrieren zu können. Zur besseren Darstellung wird die Uhr des prolongierten Programmablaufs mit C' gekennzeichnet.
Die PRAM^{dt} kann nur durch den gemeinsamen Speicher kommunizieren [CZ89]. Deshalb entspricht ein Senden einer Nachricht einem $WRITE(gA)$ und ein Empfangen einer Nachricht einem $READ(gA)$.[33]

[33] Der eindeutige Identifikator, die Adresse, des gemeinsamen, globalen Speichers wurde mit gA bezeichnet.

6.5.3.1 Nicht prolongierter Programmablauf

- Sei ein $a \rightarrow b$ direkt möglich.
- Seien *start* und *stopp* zwei explizite, vorher definierte Stellen/Befehle im Programmablauf, mit $C(start) < C(stopp)$.[34][35]
- Sei $Laufzeit = C(stopp) - C(start)$, $Laufzeit \in \mathbb{N}$
- Seien a, b zwei aufeinanderfolgende, lokale Befehle im Programmspeicher, wobei b ein *WRITE* ist auf die Speicherzelle $M(gA)$ und a kein Sprung, Return-Befehl (*RET*) oder ein Halt (*Halt*).
- Seien c, d zwei aufeinanderfolgende, lokale Befehle im Programmspeicher, wobei c ein *READ* ist auf die Speicherzelle $M(gA)$.

❑ Wird der Befehl b von einem Prozessor ausgeführt, so heißt der Prozessor $P_{a \rightarrow b}$.

❑ Wird der Befehl c von einem Prozessor ausgeführt, so heißt der Prozessor $P_{c \rightarrow d}$.

Es ist nicht bekannt, ob $a \rightarrow d$ gilt, d.h. es ist kein a priori Wissen darüber verfügbar, ob die zwei verschiedene Prozessoren parallel auf eine Ressource (d. h. eine gemeinsame Speicherzelle) zugreifen und ob der Code ausgeführt wird.

6.5.3.2 Instrumentierung – Prolongation durch NOPs

Instrumentierung

Es werden nun n_1 bis n_j No-Operation Befehle (NOPs) nach a in das Programm, also in den Programmspeicher der PRAM^{dt} , eingefügt. Alle *label* mit einem Wert kleiner der Adresse von a werden beibelassen, alle anderen *label* um j inkrementiert.[36]

6.5.3.3 Prolongierter Programmablauf

Experiment

Sei nun die Sequenz der ausgeführten Befehle von *start* bis *stopp*, bis auf die eventuell ausgeführten, hinzugefügten *NOPs* $n_1 \dots n_j$, von Prozessor $P_{a \rightarrow b}$ identisch zum Referenzlauf, dem nicht prolongierten Programmablauf.

- *start* und *stopp* sind die identischen Stellen/Befehle im Programmspeicher, wie beim nicht prolongierten Programmablauf.
- Wird a ausgeführt, gilt $a \rightarrow n_1 \rightarrow \dots \rightarrow n_j \rightarrow b$
- Demnach gilt nach [3] $a \rightarrow b$
- Da die Sequenz der ausgeführten Befehle des prolongierten und nicht prologierten Version (Referenzlauf) bis auf die hinzugefügten Befehle (der Prolongation) identisch ist, gilt: $C(a) = C'(a)$

[34] Die Befehle im Programmspeicher, die Stelle im Programm werden implizit mit dieser Definition ausgeführt.

[35] Damit ist *start* $\rightarrow$ *stopp* selbst ein Modul. Es wird als die Auswirkung des prolongierten Moduls auf *start* $\rightarrow$ *stopp* gemessen.

[36] Die Sprünge werden durch diese Instrumentierung angepasst.

- Sei nun $Laufzeit' = C'(stopp) - C'(start)$, $Laufzeit' \in \mathbb{N}$
- Prozessor $P_{c \to d}$ kann erst dann lesen, nachdem Prozessor $P_{a \to b}$ geschrieben hat, **wenn** zum Referenzlauf das Schreib- und Lesepaar erhalten bleiben soll (d. h. ausreichend synchronisiert ist).
- Anhand von $Laufzeit'$ kann nun bestimmt werden, ob auf eine gemeinsame Ressource (dem globalen Speicher $M(gA)$) zugegriffen wird, **wenn** der Code ausreichend synchronisiert ist.

Erhalt von Schreib- und Lesepaar

Ein Programmierer muss sicherstellen, dass sein intendiertes Schreib- und Lesepaar erhalten bleibt [CZ89].

31 Ausreichend synchronsierter Code
Unter einem „ausreichend synchronsierten Code“ wird ein Programmcode bezeichnet, der die intendierten Schreib- und Lesepaare auch unter einer Prolongation beibehält.

Ein Programmierer kann auf einem realen System, durch nichtdeterministische Seiteneffekte und unterschiedliche Hardware, keine Annahme zu den Lese- und Schreibzeitpunkt machen.
Er muss eine Sperre implementieren, so dass, $P_{c \to d}$ den Wert von $P_{a \to b}$ liest. Kein anderer Prozessor darf (bzw. sollte) während dieser Zeit schreiben und den Wert $M(gA)$ verändern, $P_{c \to d}$ darf nicht vor dem Schreiben von $P_{a \to b}$ lesen.[37]

- $P_{a \to b}$ hat eine Sperre implementiert, während dieser Zeit darf kein anderer Prozessor schreiben (ein $WRITE\ M(gA)$ ausführen).
- $P_{c \to d}$ hat eine Sperre implementiert, bei der geprüft wird, ob der $READ$ Befehl schon ausgeführt werden darf.
- Sind beide Sperren nicht implementiert, kann dies zu Race Conditions und Heisenbugs führen (siehe Kapitel 10).
  Werden vor dem Befehl b die NOPs $n_1 \to \cdots \to n_j$ eingefügt, verzögert sich das Schreiben um j-Zyklen.
  Fand das Lesen (das Ereignis c) vor dem Zeitpunkt $C(b) + j$ beim Referenzlauf (dem nicht prolongierten Programmablauf) statt, so muss $P_{c \to d}$ beim prolongierten Programmablauf warten, wenn eine Sperre implementiert ist.
  Durch eine Prolongation von $a \to b$ prolongiert sich auch $c \to d$.

❑ Anhand von $Laufzeit'$ kann bestimmt werden ob auf eine gemeinsame Ressource zugegriffen wird ($M(gA)$) wenn der Code ausreichend synchronisiert ist.

❑ Ist der Code nicht ausreichend synchronisiert, kann eventuell das intendierte Schreib- und Lesepaar nicht eingehalten werden. Es können Fehler bedingt durch mangelhafte Synchronisation entstehen (Synchronisationsfehler), die sich auf realen Systemen wiederholen können.

[37] Diese Situation entspricht einem *Race*. Siehe Kapitel 10.

6.5.3.4 Messbarer Wirkzusammenhang

Anhand von $Laufzeit'$ und der richtigen Wahl von j kann bestimmt werden, ob $a \rightarrow d$ gilt.

❑ **A)** $Laufzeit' = Laufzeit$

$a \rightarrow d$ gilt nicht oder j ist zu klein, um die Zusammenhänge zu bemerken. Die Laufzeitdauer der prolongierten und nicht prolongierten Version sind identisch.

Eine eventuelle Sperre kann entfernt werden, wenn sichergestellt ist, dass nie Verzögerungen $> j$ (also $C(b) + j > C(c)$) in der Systemausführung auftreten können.

❑ **B)** $Laufzeit' \approx Laufzeit + j$

a bzw. $a \rightarrow d$ wird einfach ausgeführt. Die hinzugefügten No Operations Befehle NOPs gehen einfach in die Laufzeit ein.

❑ **C)** $Laufzeit' \approx Laufzeit + n \cdot j$

a bzw. $a \rightarrow b$ werden mehrfach (n-mal mit $n \in \mathbb{N}$) ausgeführt. Es wird mehrfach auf $M(gA)$ durch den Code $a \rightarrow b$ geschrieben. Hier befindet sich Optimierungspotenzial.

Gegenüber dem lokalen Wirkzusammenhang muss hier ein $\approx$ verwendet werden, da die spezifische Realisierung der notwendigen Synchronisationskonzepte unterschiedliche Zeitdifferenzen verursachen können.

6.6 Zusammenfassung, Beantwortung der Teilfragestellungen δ, ζ, und ϵ und Diskussion

Registermaschine
PRAM
Aufbau
Phasen
PRAM Modelle
Befehlssatz
SB-PRAM

Abschnitt 6.1 führte das sequentielle Vorläufermodell der parallelen Registermaschine (kurz PRAM), die Registermaschine ein. Abschnitt 6.2 präsentierte die parallele Registermaschine, ein paralleles Maschinenmodell. In Abschnitt 6.2.1 wurde diskutiert, wieso sich die PRAM zur Darstellung des Wirkzusammenhangs eignet. Diese bot sich wegen ihrer Einfachheit und Nähe zu realen Systemen zur Darstellung an. Sie integriert gemeinsame Ressourcennutzung und die Ergebnisse können auf parallele Prozesse übertragen werden [CZ89, S. 171][Bre74]. Anschließend wurde in Abschnitt 6.2.2 der kanonische Aufbau der PRAM definiert. Abschnitt 6.2.3 erklärt die zur Vereinfachung der Analyse eingeführten Phasen der PRAM. Bei parallelen Interaktionen können Konflikte durch gemeinsam benutzte Ressourcen auftreten, Abschnitt 6.2.4 präsentiert deshalb kurz die in der Literatur gängigen Konventionen und unterschiedlichen Modelle. Abschnitt 6.2.5 behandelt den Befehlssatz der PRAM. Eine gut dokumentierte, erforschte und vor allem praktisch realisierte parallele Registermaschine ist die *SB-PRAM* (*Saarbrücken Parallel Random Access Machine*). Abschnitt 6.2.6 führt die SB-PRAM ein, diese wurde insbesondere praktisch umgesetzt. Hier wird der Unterschied zu weiteren, in der Literatur verwendeten, Befehlssätzen der parallelen Registermaschine diskutiert und die in die *SB-PRAM* integrierten *Multipräfixoperationen* erklärt.

PRAM^{dt}
Minimalbefehlssatz

Abschnitt 6.3 definiert die PRAM^{dt}, eine durch eine Akkumulator-Architektur simplifizierte PRAM. Diese ist angelehnt an die gut erforschte, dokumentierte und praktisch umgesetzte *SB-PRAM* und wurde zur Darstellung des Wirkzusammenhangs (mit einer Akkumulatorarchitektur) auf das Wesentliche reduziert. Abschnitt 6.3.1 behandelt den Aufbau der PRAM^{dt}. Abschnitt 6.3.2 definiert aufbauend auf den vorherigen Abschnitten den für diese Arbeit notwendigen Minimalbefehlssatz der PRAM^{dt} (basierend auf einer Akkumulatorarchitektur). In Abschnitt 6.3.3 bis Abschnitt 6.3.5 werden unterschiedliche Ansätze zur Realisation des Analyseinstrumentariums diskutiert.

Wichtige Ergebnisse aus dem Abschnitt 6.3 sind:

- ❑ Die Prolongation muss vor dem Zustandsübergang, der Ausführung eines Maschinenbefehls erfolgen. Es muss vor dem eigentlich Befehl bzw. Mikrobefehl „Zeit eingefügt“ werden.
- ➪ Deshalb lassen sich das *simulierte Optimieren* und die *Retardation* auf die *Prolongation* zurückführen.
- ❑ Ein Gegenbeispiel zeigt deutlich, dass nicht die Phasen der Befehlsausführung einer PRAM^{dt} genutzt werden dürfen um zwischen ihnen die Prolongation zu realisieren.
- ➪ Die Prolongation muss deshalb atomar auf Befehlsebene erfolgen.

Korrektheit

Abschnitt 6.4 diskutiert die Korrektheit bei dem propagiertem Experiment in diesem parallelen Maschinenmodell. Die Maschinenbefehle der PRAM^{dt} werden dazu in Abschnitt 6.4.1 in lokale und globale Operationen klassifiziert. Abschnitt 6.4.2 beweist, analog zu Kapitel 5, die lokale Korrektheit. Abschnitt 6.4.3 diskutiert die globale Korrektheit der PRAM^{dt} unter einer Prolongation. Abschnitt 6.4.4 führt für die Zeitmessung in der PRAM^{dt} in Abschnitt 6.4.5 die logischen Uhren nach *Lamport* ein. Abschnitt 6.4.6 zeigt, dass die Semantik einer, mittels der definierten Multipräfixoperationen implementierten, Sperre oder Barriere unter einer Prolongation erhalten bleibt.

Wirkzusammenhang

Abschnitt 6.5 zeigt den Wirkzusammenhang des Analyseinstrumentariums. Hier wird dargestellt, wieso durch den experimentellen Ansatz mit dem Analyseinstrumentarium Optimierungspotenziale und -kandidaten in einem System gefunden werden.

Referenzlauf

Ein einfaches Gegenbeispiel zeigt, dass die Darstellung des Wirkzusammenhangs am besten anhand eines Referenzlauf demonstriert wird.

- ❑ Ein Programm kann sich selbst ausmessen und unter einer Prolongation ein völlig anderes Verhalten zeigen. Die Prolongation als Operation kann „überlistet“ werden.
- ❑ Nur durch die Laufzeitdifferenz von Programmen, die sich stark nicht-deterministisch bzw. zufällig Verhalten (z. B. durch eine *Random*-Funktion in einer Schleife), kann auf keine Abhängigkeit geschlossen werden bzw. nach dem *Gesetz der großen Zahlen* erst nach einer hohen Anzahl von Versuchen.
- ❑ Beide Funktionen stellen Ausnahmefälle dar. Programme mit Performanzproblemen sind nicht dazu geschrieben, um die Interpretation der Analyse des hier propagierten Experimentes zu verfälschen. Die Prolongation als Analyseinstrumentarium gibt in beiden Fällen das exakte Ergebnis und Verhalten wieder.

Der Referenzlauf und der Programmlauf unterscheiden sich nur durch die zusätzlichen, instrumentierten *NOP*s (No-Operation Befehle).

In Abschnitt 6.5.1 wird die von Lamport geprägte *Happend-Before Relation* eingeführt.

lokaler Wirkzusammenhang

Aufbauend auf den Punkten 1 und 3 dieser Happend-Before Relation wird in Abschnitt 6.5.2 gezeigt, dass durch eine (im Experiment messbare) Laufzeitdifferenz gegenüber einem Referenzlauf bestimmt werden kann, ob eine prolongierte Komponente (ein Modul oder eine Hardwarekomponente d. h. ein Prozessor der PRAM^{dt}) in der zu optimierenden Komponente genutzt wird. Mit dem Analyseinstrumentarium kann somit also festgestellt werden, welche Komponenten Optimierungskandidaten darstellen. Durch die Höhe der Laufzeitdifferenz kann festgestellt werden, welches Optimierungspotenzial die prolongierte Komponente darstellt.

globale Wirkzusammenhang

Aufbauend auf den Punkten 2 und 3 dieser Happend-Before Relation wird in Abschnitt 6.5.3 der globale Wirkzusammenhang demonstriert. Die Interprozessor bzw. Interprozesskommunikation kann in der PRAM^{dt} nur durch den gemeinsamen Speicher erfolgen. Hierzu muss sichergestellt werden, dass das intendierte Schreib- und Lesepaar der Prozessoren wirklich erhalten bleibt. Dies kann nur durch entsprechende Barrieren und Sperren realisiert werden (deren Semantik unter einer Prolongation erhalten bleibt). Mit dem Analyseinstrumentarium kann somit festgestellt werden, welche Komponenten auf eine gemeinsame Ressource zugreifen und damit, durch Sperren, Optimierungskandidaten darstellen. Durch den Faktor der Laufzeitdifferenz kann abgeschätzt werden, welchen Einfluß die prolongierte Komponente auf die Laufzeit hat und damit welches Optimierungspotenzial diese darstellt.

Somit können die Teilfragestellungen δ, ζ und ϵ beantwortet werden.

Teilfragestellung δ ✔

Die Teilfragestellung δ nach der *Validität* bzw. der *Korrektheit* kann bei parallelen Berechnungsmodellen nur mit Synchronisationsmechanismen wie Sperren und Barrieren positiv beantwortet werden. Damit werden die intendierten Schreib- und Lesepaare erhalten, das Ergebnis einer Berechnung wird dann durch das Analyseinstrumentarium nicht verändert. Eventuelle Seiteneffekte und andere Zielplattformen müssen jedoch beachtet werden, so dass dieser (Synchronisations-)Fehler sich wiederholen könnte. Das Analyseinstrumentarium kann folglich als Basis zum Test auf solche Fehler genutzt werden. Eine Testmethodologie (Teilfragestellung κ) wurde in diesem Kapitel noch nicht entwickelt, diese folgt in Kapitel 10.

Teilfragestellung ϵ ✔

Der Wirkzusammenhang des Analyseinstrumentariums wurde auf Basis der PRAM^{dt} (als formales Modell), den *Lamport Uhren* und der *Happend-Before Relation* (als Werkzeuge) gezeigt. Somit konnte die Teilfragestellung ϵ nach dem Wirkzusammenhang beantwortet werden.

Die Teilfragestellung ζ, ob durch das Analyseinstrumentarium Optimierungspotenzial im System entdeckt wird, ergibt sich direkt aus dem Wirkzusammenhang (bei der PRAM^{dt}). Durch das Analyseinstrumentarium kann ausgeführter Code und das gemeinsame Nutzen einer (ausreichend synchronisierten) Ressource erkannt werden. Die Verbesserung der Laufzeit einer Komponente (z. B. eines Moduls, insbesondere auch des Hauptprogramms der PRAM^{dt}) kann nur durch eine Optimierung des aufgerufenen Codes oder dem verbesserten Zugriff auf diese gemeinsame Ressource erfolgen.

Teilfragestellung ζ ✔

Kapitel 7

Experimentierumgebung

„Nur ein Narr macht keine Experimente."

Charles Darwin

Übersicht zum Kapitel

Die Experimentierumgebung, eine Prolongation und das Protokollieren von Laufzeiten, kann durch unterschiedliche Mechanismen und auf unterschiedlichen Ebenen im System realisiert werden.
Abschnitt 7.1 zeigt auf, wie die Experimentierumgebung im Code durch Instrumentierung möglich wird. Demonstriert wird dies mit *AspectJ*.
Eine Erweiterung ist eine Experimentierumgebung mittels einer virtuellen Maschine. Durch die Virtualisierung wird eine größere Kontrolle des zu untersuchenden Systems erreicht. Sie stellt einen ganzheitlichen Ansatz dar, nicht nur Code sondern auch die unterliegende Hardware wird prolongiert. Abschnitt 7.2 führt kurz in die Thematik ein. Abschnitt 7.3 zeigt die Realisation der Experimentierumgebung QEMU^{dt} mit dem Virtualisierer *QEMU*.

7.1 Prolongation auf Modulebene

Experimentierumgebung durch Instrumentierung

Durch eine Instrumentierung lassen sich die vorgeschlagenen Experimente im Code eines Systems realisieren. Aspektorientierte Programmierung stellt hier einen besonders effizienten und bequemen Weg dar [Soko6].

Aufbau von Abschnitt 7.1

In diesem Abschnitt wird nach einer Motivation eine kurze Einführung der aspektorientierten Programmierung gegeben. Darauf aufbauend wird gezeigt, wie sich die Prolongation, das simulierte Optimieren und ein dazugehöriges Tracing damit schnell und einfach realisieren lässt. Abschließend werden die Grenzen der Prolongation mittels aspektorientierter Programmierung diskutiert und zur Prolongation mittels Virtualisierung übergeleitet.

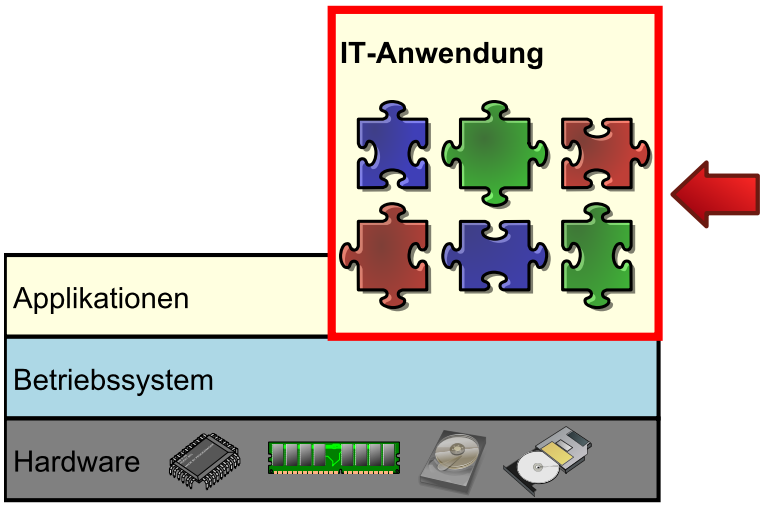


Abbildung 7.1: Die Experimentierumgebung (Prolongation und Logging) auf Modul- oder Codeebene, realisiert durch Instrumentierung.
Die zeitlich beeinflussten und protokollierten Elemente bei der Realisation der Prolongation durch eine Instrumentierung sind die Module (im System durch den roten Pfeil gekennzeichnet).

7.1.1 Experimentierumgebung mittels aspektorientierte Programmierung

Erweiterung Aspektorientierung

Als logische Weiterführung von bestehenden Programmierparadigmen wurde 1997 von Kiczales, Lamping, Mendhekar, Maeda, Lopes, Loingtier und Irwin [Kic+97] die aspektorientierte Programmierung vorgeschlagen auf das XEROX ein Patent besitzt [Kic+02].

Motivation für diese Entwicklung war, dass bestimmte Funktionalitäten über den gesamten Code verteilt sind – beispielsweise das Logging oder Funktionalitäten zur Sicherheit von Systemen – ohne dass diese hinreichend gut modularisiert werden können, da sich diese querschnittlichen Belange oder *Cross-Cutting-Concerns* sich quer verstreut über den gesamten Code verteilen.

Motivation Cross-Cutting-Concerns

Aspektorientierung kann als eine Erweiterung einer Programmierungsprache um eine zweite, programmierbare Dimension verstanden werden. AspectJ ist die gebräuchlichste aspektorienterte Erweiterung für Java und ist frei verfügbar [Theo7]. AspectJ führt zwei neue Konstrukte ein: den *Joinpoint* (bzw. *Pointcuts*) und den *Advice* [Bö5].

programmierbare zweite Dimension

AspectJ Konstrukte

7.1.1.1 Joinpoint

Ein Joinpoint ist eine definierte Stelle im Programmablauf bzw. - fluss [HG06, S. 63]. Dies kann der Aufruf einer Methode oder eine Zuweisung sein. An dieser Position kann Code eingebunden werden. Dies wird als *waeving* bzw. als (ein)-weben bezeichnet [HH04b], es entspricht einer Instrumentierung.[1]

Joinpoint

7.1.1.2 Pointcut

Ein Pointcut ist eine Menge von Joinpoints, welche entsprechend spezifiziert sind [Bö5]. Beispielsweise in der *AspectJ* Syntax durch `call * *.main (..)`. Dies spezifiziert den Aufruf (`call`) aller main-Methoden mit beliebigen Parametern (`(..)`) in allen Klassen in allen Paketen (`*.main`) mit beliebigen Rückgabewerten (`* *.main (..)`) [Mano7, S. 29].
Anstelle der Methodeninvokation (`call`) werden durch passende Schlüsselwörter Zugriffe auf Objekte oder Ausführung von Code spezifiziert [Mano7, S. 30]. Im Anwendungsfall wurde mit execution (Ausführung) und `call`, dem Aufruf von Methoden, gearbeitet.
Auf den Pointcuts, der Menge von Joinpoints, sind Operatoren zur Verknüpfung definiert. Die Schnittmenge bildet `&&` (und), die Vereinigungsmenge `||` (oder) zweier Pointcuts. Der Operator `!` (nicht) ergibt das Komplement bzw. die Komplementmenge eines Pointcuts (bzgl. aller möglichen Pointcuts) [Mano7, S. 30].

Pointcut

Verknüpfungsoperationen

7.1.1.3 Advice

Ein Advice ist Code, der vor (before), nach (after) oder anstelle (around) des JoinPoints bzw. der Pointcuts ausgeführt wird. Hier nimmt return proceed noch eine besondere Stellung im around-advice ein, indem der Kontrollfluss dem eigentlichen „herausgefilterten“ Pointcut weitergegeben wird. Ein around-advice mit einem return proceed umschließt also einen Pointcut [Mano7, S. 30].

Advice als ausgeführter Code

[1] zur Instrumentierung siehe Abschnitt 3.3.2, S. 46

In [Man07] wurde die Variation der Laufzeit durch Instrumentierung von Java-Code mittels AspectJ durchgeführt. Dies ermöglichte eine Performanzanalyse auf Modulebene. Es wurde so mit möglich, das Zusammenwirken der einzelnen Module auf Softwareebene zu verstehen.

7.1.2 *Logging*

Logging, Tracing

Die beiden Begriffe *Logging* und *Tracing* werden in dieser Arbeit synonym benutzt und meinen das Protokollieren der Daten eines Programmlaufs, also insbesondere das Protokollieren der Daten des Experiments.
Im Rahmen dieses Themas wurden unterschiedliche Loggingmechanismen für das Tracing entwickelt. Bewährt hat sich der unten beschriebene Aspekt [Man07; Man06], der für diese Arbeit weiterentwickelt wurde. Dieses Einweben kann via einer Entwicklungsumgebung (z. B. *Eclipse*) oder der Kommandozeile erfolgen [Man07].

7.1.2.1 Anlegen der Protokolldatei

Anlegen des Tracefiles

Die entsprechende Protokolldatei (*Tracefile*) wird vor dem Benutzen durch den static-Block mit createFile angelegt. Zur besseren Unterscheidung erhält der Dateiname das aktuelle Datum mit Uhrzeit. [Mano7]

```
  static {
    Calendar c = Calendar.getInstance();
    String s = "" + c.get(Calendar.YEAR) + c.get(Calendar.MONTH) + "
"
        + c.get(Calendar.HOUR) + "_" +
c.get(Calendar.MINUTE) + "_"
        + c.get(Calendar.SECOND);

    // File generieren mit aktuellem Datum

    try {
      createFile("C:\\Tracing_Demo_" + s + ".txt", "");
    } catch (FileNotFoundException ee) {
      System.out.println("FileNotFoundException");
    }

    tlll = new threadLogLinkedList();
    dt = System.currentTimeMillis();
  }
```

Listing 7.1: AspectJ Code-Snippet: static-Block im Aspekt zum Erstellen einer Logdatei.

Granularität Methodenaufrufe

Die betrachtete und protokollierte Granularitätsstufe sind hier die Methodenaufrufe. Analog können Pakete uvm. protokolliert werden. Bei dieser Version des Loggings werden der Startzeitpunkt und der Endzeitpunkt der Methode protokolliert, die Laufzeitdauer ergibt sich aus beiden Zeitpunkten durch eine Subtraktion. Durch einen Pointcut mit einem around kann die Laufzeitdauer direkt gemessen werden.

Aufrufstack

Bei nur einem gleichzeitig zu protokollierenden Ablauf (*singlethreaded*) kann nun kellerartig, also mit einem *Stack*, der Aufruf einer Methode (call) vor before und danach after mit einem *Pointcut* protokolliert werden.

parallele Systeme

Die Daten sind bei parallelen Systemen (*multithreaded*) gewissermaßen zweidimensional: die Methodenzeitpunkte werden strikt chronologisch protokolliert, können aber in unterschiedlichen Threads stattgefunden haben. Die entsprechenden Threads müssen mitprotokolliert werden, da ansonsten nicht mehr identifiziert werden kann, welche Methode zu welchem Thread gehört. So kann die Laufzeitdauer der Module nicht berechnet werden.

multithread Logging

Für ein multithreadfähiges und -sicheres Logging wurden die beiden Klassen threadLog und threadLogLinkedList implementiert [Mano7].

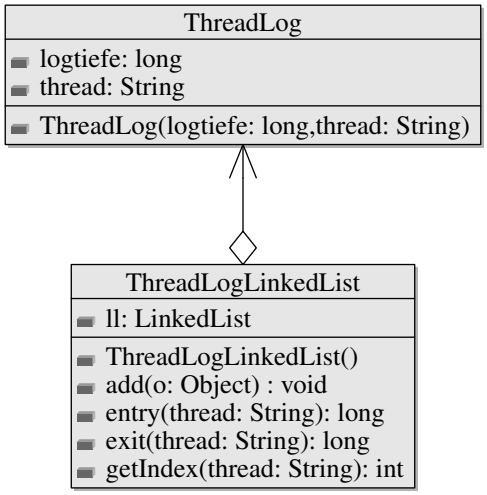


Abbildung 7.2: Klassendiagramm: Hilfsklassen ThreadLog und ThreadLogLinkedList.

Die Klasse threadLog hat Variablen/Objekte für die aktuelle Aufruftiefe des Threads und einen String für den Threadnamen. Die Klasse threadLogLinkedList verknüpft diese Objekte durch eine verkettete Liste.

Filterung Die eigentlichen Pointcuts TAanyMethodBefore() und TAanyMethodAfter() filtern den Eintritt und das Verlassen in eine Methode heraus. Hierbei wurden alle Methoden aus dem Aspekt Trace2 (&& ! within(Trace2)) und den Klassen threadLogLinkedList (&& ! within(threadLogLinkedList)) sowie threadLog && ! within(threadLog) ausgeschlossen um kreisförmige Protokollierung und damit „Endlosschleifen“ bzw. eine endlose Protokollierung durch den Aspekt zu vermeiden.

```
pointcut TAanyMethodBefore():    call (* *.* (..))
&& !   call (* java.*.*(..))
&& ! within(Trace2)
&& ! within(threadLogLinkedList)
&& ! within(threadLog)  ;

pointcut TAanyMethodAfter():     call (* *.* (..))
&& ! call (* java.*.*(..))
&& ! within(Trace2)
&& ! within(threadLogLinkedList)
&& ! within(threadLog)  ;
```

Listing 7.2: AspectJ Code-Snippet: Tracing Pointcuts before and after.

verkettete Liste der Threads

Mittels der Anweisung index = this.getIndex(thread); wird geprüft, ob der aktuelle Thread schon protokolliert wurde. Falls nicht, wird ein neues ThreadLog-Objekt angelegt und in die verkettete Liste eingefügt. Die Entrymethode (entry) wird **vor** der Methodenausführung vom Aspekt ausgeführt und inkrementiert die Logtiefe des Threads. Die Exitmethode (exit) wird **nach** der Methodenausführung von dem Aspekt ausgeführt und dekrementiert die Logtiefe des Threads. [Man07]

```
public class ThreadLog {

  public long logtiefe = 0;
  public String thread = "";

  ThreadLog(long logtiefe, String thread) {
    this.logtiefe = logtiefe;
    this.thread = thread;
  }
}
```

Listing 7.3: Java: Klasse ThreadLog

```
import java.util.LinkedList;
import java.util.ListIterator;

public class ThreadLogLinkedList {

  public LinkedList ll;

  ThreadLogLinkedList() {
    ll = new LinkedList();
  }

  public synchronized void add(Object o) {
    ll.add(o);
  }

  public synchronized long entry(String thread) {

    int index;
    ThreadLog tl_temp;

    index = this.getIndex(thread);

    {
      if (index >= 0) {
        tl_temp = (ThreadLog) ll.get(index);
        tl_temp.logtiefe++;
        return tl_temp.logtiefe;

      } else {
        tl_temp = new ThreadLog(0, thread);
        this.add(tl_temp);
        return 0;
      }
    }
  }

  public synchronized long exit(String thread) {

    int index;
    ThreadLog tl_temp;

    index = getIndex(thread);
    if (index >= 0) {
      tl_temp = (ThreadLog) ll.get(index);
      tl_temp.logtiefe--;
      return tl_temp.logtiefe;

    } else {
      tl_temp = new ThreadLog(0, thread);
      ll.add(tl_temp);
      return 0;
    }
  }

  public synchronized int getIndex(String thread) {
    ThreadLog tl_temp;

    if (ll.size() == 0) {
      return -1;
    }

    ListIterator li = ll.listIterator(0);
    tl_temp = (ThreadLog) ll.get(0);

    if (tl_temp.thread.equals(thread))
      return li.previousIndex() + 1;
    else
      while (li.hasNext()) {

        tl_temp = (ThreadLog) li.next();

        if (tl_temp.thread.equals(thread))
          return li.previousIndex();
      }
    return -1;
  }
}
```

Listing 7.4: Java: Klasse ThreadLogLinkedList

```
import java.util.*;
import java.io.*;
import org.aspectj.lang.*;
import org.aspectj.lang.reflect.*;

public aspect Trace2 {
  protected static int callDepth = 2;
  public static int TRACELEVEL = 2;
  protected static PrintStream stream = System.out;
  protected static long dt;
  public static BufferedWriter bw; // ... zum schreiben
  public static String s;
  protected static ThreadLogLinkedList tlll;

  static {
    Calendar c = Calendar.getInstance();
    String s = "" + c.get(Calendar.YEAR) + c.get(Calendar.MONTH) + "
"
        + c.get(Calendar.HOUR) + "_" +
c.get(Calendar.MINUTE) + "_"
        + c.get(Calendar.SECOND);

    try {   // File generieren mit aktuellem Datum
      createFile("C:\\Tracing_Demo_" + s + ".txt", "");
    } catch (FileNotFoundException ee) {
      System.out.println("FileNotFoundException");
    }
    tlll = new ThreadLogLinkedList();
    dt = System.currentTimeMillis();
  }

  public static void createFile(String fileName, String OUTPUT)
      throws FileNotFoundException {
    File file = new File(fileName);

    if (file.exists()) {
      System.out
          .println("Das File '" + fileName + "'
existiert bereits!");
    } else {

      FileWriter fw = null;
      try {
        fw = new FileWriter(file);
      } catch (IOException e) {
        System.out.println(e);
      }
      bw = new BufferedWriter(fw);
      System.out.println("Das File '" + fileName + "' wurde
erstellt!");

    }
  }

  protected static synchronized void traceEntry(String str, String thread)
{
    if (TRACELEVEL == 0)
      return;
    if (TRACELEVEL == 2)
      callDepth++;
    try {
      bw.write("" + tlll.entry(thread) + "\t" + str +
"\tEnter\t");
      bw.newLine();
      bw.flush();
    } catch (IOException e) {
    }
  }

  protected static synchronized void traceExit(String str, String thread)
{
    if (TRACELEVEL == 0)
      return;
    try {
      bw.write("" + tlll.exit(thread) + "\t" + str +
"\tExit\t");
      bw.newLine();
      bw.flush();
    } catch (IOException e) {
    }
    if (TRACELEVEL == 2)
      callDepth--;

  }

  pointcut TAanyMethodBefore():   call (* *.* (..))
  && !   call (* java.*.*(..))
  && ! within(Trace2)
  && ! within(ThreadLogLinkedList)
```

```
  && ! within(ThreadLog)  ;

  pointcut TAanyMethodAfter():     call (* *.* (..))
  && ! call (* java.*.*(..))
  && ! within(Trace2)
  && ! within(ThreadLogLinkedList)
  && ! within(ThreadLog)  ;

  after(): TAanyMethodAfter ()
  {
    try {

      traceExit("" + Thread.currentThread() + "\t" +
thisJoinPoint + "\t"
          + (System.currentTimeMillis() - dt), ""
          + Thread.currentThread());
    } catch (Exception e) {
    }
  }

  before(): TAanyMethodBefore()
  {
    try {
      traceEntry("" + Thread.currentThread() + "\t" +
thisJoinPoint
          + "\t" + (System.currentTimeMillis() -
dt), ""
          + Thread.currentThread());
    } catch (Exception e) {
    }
  }
}
```

Listing 7.5: AspectJ: Tracing Aspect – Version 2: before and after

Problematisch ist das benötigte `flush`, (siehe die nachfolgende Diskussion in Abschnitt 7.1.8) da ansonsten können Ungenauigkeiten bei der Protokolldatei auftreten.
Der eingewebte und ausgeführte Thread ergibt das in Tabelle 7.1.2.1 abgebildete Protokoll (bzw. Log oder Trace), die als Textdatei gespeichert wurde.

Nanosekunden

Anstelle von `System.currentTimeMillis()` kann auch `System.nanoTime()` verwendet werden. Kritisch ist hier ein eventueller Überlauf (`currentTimeMillis` läuft seit dem 1. Januar 1970 00:00 h) und systeminterne Ungenauigkeiten, da Systemuhren nicht im Nanosekundenbereich messen können. Tabelle 7.1.4 zeigt die entsprechende Tabelle mit *Nanosekunden*.

```
Object around(): TAanyMethodAround()
  {
    long dt2 = System.nanoTime();
    traceEntry("" + Thread.currentThread() + "\t" + thisJoinPoint +
"\t"
        + thisJoinPoint.getThis() + "\t"
        + (System.nanoTime() - dt2), ""
        + Thread.currentThread());

    try {
      return proceed();
    } finally {

      traceExit("" + Thread.currentThread() + "\t" +
thisJoinPoint + "\t"
          + (System.nanoTime() - dt2), "" +
Thread.currentThread());
    }
  }
```

Automatisiert oder semi-automatisiert mit entsprechenden Programmen (z. B. *Excel* oder allgemeinen Editoren) kann diese Tabelle weiterverarbeitet werden. Somit können diese zur Weiterverarbeitung mit Statistikprogrammen wie *R* oder *SPSS* vorverarbeitet werden. Weiterverabeitung

7.1.3 *Prolongation*

Die Idee zur Prolongation ist nun folgende: für jedes Modul, welches verlängert werden soll, wird ein Pointcut spezifiziert. Im zugehörigen Advice wird Code eingefügt, der die Verlängerung übernimmt. Idee

7.1.4 *Prolongation durch Blockieren*

Der naheliegende `sleep()` zum Blockieren eines Threads um eine gewisse Zeit eignet sich nicht sehr gut für die Realisation der Prolongation um einen Zeitfaktor: `sleep()`

❑ **Systemauslastung**

Zwar gilt *Busy Waiting*, also *Polling* oder *Spinning* anstelle einer Blockierung, im Allgemeinen als ein sehr schlechter Programmierstil. Es erhöht die Last im System, andere Ausführungsstränge werden durch die Ressourcenauslastung blockiert [Bloo8, S. 286]. Für die Prolongation als Analyseinstrumentarium kann dies

0	Thread[main,5,main]	call(long Ill...e.fac_plus(long, long))	16	Enter
1	Thread[main,5,main]	call(long Ill...e.fac(long))	16	Enter
0	Thread[main,5,main]	call(long Ill...e.fac(long))	16	Exit
1	Thread[main,5,main]	call(long Ill...e.fac(long))	16	Enter
0	Thread[main,5,main]	call(long Ill...e.fac(long))	16	Exit
-1	Thread[main,5,main]	call(long Ill...e.fac_plus(long, long))	16	Exit
0	Thread[main,5,main]	call(long Ill...e.fac_mal(long, long))	16	Enter
1	Thread[main,5,main]	call(long Ill...e.fac(long))	16	Enter
0	Thread[main,5,main]	call(long Ill...e.fac(long))	16	Exit
1	Thread[main,5,main]	call(long Ill...e.fac(long))	16	Enter
0	Thread[main,5,main]	call(long Ill...e.fac(long))	16	Exit
-1	Thread[main,5,main]	call(long Ill...e.fac_mal(long, long))	16	Exit
0	Thread[main,5,main]	call(void java.io.PrintStream.println(long))	16	Enter
-1	Thread[main,5,main]	call(void java.io.PrintStream.println(long))	16	Exit

Tabelle 7.1: Tracefile von Szenario 1 – Illustrationsszenario Methodenaufrufe gemessen in Millisekunden.

❑ Wegen der Seitenbreite dieses Dokuments wurden bei IllustrationsszenarioMethodenaufrufe.fac_mal und IllustrationsszenarioMethodenaufrufe.fac_plus Punkte eingesetzt, um die Breite zu reduzieren.

❑ Gemessen wurde auf einem *Windows XP* Rechner, *Pentium 4* (3 GHz), *Java 1.6.0_20* mit den Zahlenwerten $a = b = c = d = 2$ für Szenario 1.

0	Thread[main,5,main]	call(long Ill...e.fac_plus(long, long))	3646552	Enter
1	Thread[main,5,main]	call(long Ill...e.fac(long))	7281372	Enter
0	Thread[main,5,main]	call(long Ill...e.fac(long))	7741207	Exit
1	Thread[main,5,main]	call(long Ill...e.fac(long))	8152432	Enter
0	Thread[main,5,main]	call(long Ill...e.fac(long))	8300217	Exit
-1	Thread[main,5,main]	call(long Ill...e.fac_plus(long, long))	8431798	Exit
0	Thread[main,5,main]	call(long Ill...e.fac_mal(long, long))	8701106	Enter
1	Thread[main,5,main]	call(long Ill...e.fac(long))	8906718	Enter
0	Thread[main,5,main]	call(long Ill...e.fac(long))	9031874	Exit
1	Thread[main,5,main]	call(long Ill...e.fac(long))	9173512	Enter
0	Thread[main,5,main]	call(long Ill...e.fac(long))	9293639	Exit
-1	Thread[main,5,main]	call(long Ill...e.fac_mal(long, long))	9410134	Exit
0	Thread[main,5,main]	call(void java.io.PrintStream.println(long))	9579150	Enter
-1	Thread[main,5,main]	call(void java.io.PrintStream.println(long))	9972776	Exit

Tabelle 7.2: Tracefile von Szenario 1 – Illustrationsszenario Methodenaufrufe, gemessen in Nanosekunden.
❑ Wegen der Seitenbreite dieses Dokuments wurden IllustrationsszenarioMethodenaufrufe durch Ill... ersetzt, um die Breite zu reduzieren.
❑ Gemessen wurde auf einem *Windows XP* Rechner, *Pentium 4* (3 GHz), *Java 1.6.0_20* mit den Zahlenwerten $a = b = c = d = 2$ für Szenario 1.

jedoch gewünscht sein. Optimal, zum Zwecke der Analyse, soll ein Modul um einen Faktor X länger erscheinen, inklusive der charakteristischen Ressourcennutzung des Moduls.

❑ **Abhängigkeiten vom Thread Scheduler**
Der *Thread Sleep* ist abhängig vom Scheduler des Betriebssystems [Bloo8, S. 286], welcher jedoch zum Zwecke der Analyse zu ungenau ist. In Experimenten auf einem Windows-System blockierten die Threads um ganzzahlige Vielfache von 16 oder 20 Millisekunden obwohl im Nanosekundenbereich prolongiert wurde [Mano7, S. 42]. Linux und Mac OS X zeigten bei Versuchen ein ähnliches Verhalten.

Prolongation als Werkzeug

Die Prolongation sollte als ein Werkzeug verstanden werden, das je nach Kontext eingesetzt wird. Viele Anwendungsfälle, nicht nur zur Analyse von Systemen, sind möglich. In [Mano6] wurde beispielsweise in einem hochkomplexen Erzeuger-Verbraucher-Problem durch Prolongationen mit einer `sleep()` Operation in der Gesamtlaufzeit verbessert, eine Prolongation durch Blockieren kann je nach Situation durchaus vorteilhaft sein.

Exkurs 7.1.1 Optimierung durch Blockierung.
Konkurrieren mehrere parallele Ausführungsstränge um ein gemeinsames, gesperrtes Objekt, kann durch hinzugefügte Laufzeiten (Prologation mittels einer `sleep()`-Operation) eine bessere Gesamtleistung (z. B. eine schnellere Laufzeit des Programms oder höherer Durchsatz) entstehen. Diese paradoxe Situation hängt von der Umgebung und der Last (*workload*) des Systems ab [DAKoo, S. 153]. Künstlich hinzugefügte Sperren können die Gesamtperformanz erhöhen [Mano6]. Den gleichen Effekt (mit unterschiedlicher Last und Umgebung) können modifizierte Sperren, die erst nach einer gewissen Anzahl von CPU Zyklen blockieren, haben [DAKoo, S. 153]. Dieser Mechanismus mit diametralen Effekt wird als *busywait/busywaiting* oder *spin/spinning* bezeichnet.
Karlin u. a. [Kar+91] vergleichen fünf unterschiedliche Strategien zwischen *always block* bis zum *always spin*. Es wird bemerkt, dass eine optimale Strategie abhängig vom Programm ist und ein adaptiver Algorithmus die Gesamtperformanz beträchtlich erhöhen kann. Insbesondere könnte sich die Prolongation zur Eruierung der optimalen Adaptionsstrategie eignen, was jedoch nicht Thema dieser Arbeit ist.
Mukherjee und Schwan [MS93a] haben hierzu auch empirische Untersuchungen durchgeführt. In einer weiterführenden Arbeit bemerken sie *„When multiple threads on each processor are capable of making progress, the use of blocking is preferred even for fairly small critical sections, since spinning prevents the progress of other threads not currently waiting on a critical section.“* [MS93b].
Veranschaulichen lässt sich dies wie folgt: Ist die Anzahl der Threads signifikant höher als die Anzahl der Prozessoren konkurrieren parallele Ausführungsstränge nun um zwei Ressourcen: das gesperrte Objekt sowie den Prozessoren. Ein Kontextwechsel kostet durch den Overhead nun vor allem Zeit (blocking) jedoch steht der Prozessor anderen Ausführungssträngen zur Verfügung, was sich positiv auf die Performanz auswirken kann.
In [Mano6] wurde, eine Abstraktionsstufe höher, ein spezielles Erzeuger-Verbraucher-Problem behandelt. Hierbei wurden unterschiedliche Objekte erzeugt, die von unterschiedlichen Verbrauchern von optimal bis suboptimal verbraucht werden. Dies kann stellvertretend für ein System von verschiedenen Systemen (Rechnern, etc.) stehen, auf denen Arbeitslast verteilt werden soll. Jedes System verarbeitet jedoch Teile dieser Arbeitslast besser als andere (beispielsweise auf Grund besserer CPUs, Festplatten, Speicher etc.). Durch eine geeignete Permutation kann das System optimiert werden. Diese Permutation kann beispielsweise durch eine Prolongation mittels Blockierung geschehen.
Optimierungsprobleme dieser Art sind sehr interessant und wichtig, werden jedoch nicht in dieser Arbeit gelöst. Die Prolongation stellt hierzu jedoch eventuell ein benötigtes Werkzeug zum Auffinden einer möglichen Bewertungsfunktion dar.

7.1.5 *Prolongation durch Busy Waiting*

Die aspektorientierte Version der Prolongation arbeitet mit Polling oder Spinning, was den Prozessor, als wichtige Komponente, auslastet. Listing 7.6 zeigt dies. Vor Beginn der Schleife wird in der Variablen `dt` die aktuelle Systemzeit (in Nanosekunden) gespeichert, die Schleife wird so lange wiederholt bis der Prolongationsfaktor Δt verstrichen ist. [Mano7] Polling

```
long dt = System.nanoTime();

while (System.nanoTime() - dt < Δt)
  ;
```

Listing 7.6: Pseudocode: Prolongation

7.1.6 *Praktische Realisation der Prolongation am Beispiel von Szenario 1*

In Listing 7.7 ist ein Aspekt zur Realisation der Prolongation für Szenario 1 angegeben. Die entsprechenden Pointcuts können automatisiert bzw. semi-automatisiert aus einem *Trace-File* erzeugt werden. Beide Aspekte, der Trace-Aspekt und der Prolongations-Aspekt, werden in ein Projekt eingewoben um ein Experiment und die Datenerhebung zu haben.

```
public aspect Prolong {
  declare precedence :Trace2,verlaengern;
  static long Prolong_fac = 0;
  static long Prolong_fac_plus = 0;
  static long Prolong_fac_mal = 0;
  pointcut Prolong_fac(): execution (* *.fac (..));
  pointcut Prolong_fac_plus(): execution (* *.fac_plus (..));
  pointcut Prolong_fac_mal(): execution (* *.fac_mal (..));

  Object around(): Prolong_fac(){
    try {
      long dt = System.nanoTime();

      while (System.nanoTime() - dt < Prolong_fac)
        ;
      return proceed();
    } finally {
    }
  }

  Object around(): Prolong_fac_plus(){
    try {
      long dt = System.nanoTime();

      while (System.nanoTime() - dt < Prolong_fac_plus)
        ;
      return proceed();
    } finally {
    }
  }

  Object around(): Prolong_fac_mal(){
    try {
      long dt = System.nanoTime();

      while (System.nanoTime() - dt < Prolong_fac_mal)
        ;
      return proceed();
    }

    finally {
    }
  }

  public static void main(String[] args) {

    final long v1 = 10000000;
    final long v2 = 50000000;
    final long v3 = 100000000;

    Prolong_fac = v1;
    IllustrationsszenarioMethodenaufrufe.main(args);

    Prolong_fac = v2;
    IllustrationsszenarioMethodenaufrufe.main(args);

    Prolong_fac = v3;
    IllustrationsszenarioMethodenaufrufe.main(args);

    Prolong_fac = 0;

    Prolong_fac_plus = v1;
    IllustrationsszenarioMethodenaufrufe.main(args);

    Prolong_fac_plus = v2;
    IllustrationsszenarioMethodenaufrufe.main(args);

    Prolong_fac_plus = v3;
    IllustrationsszenarioMethodenaufrufe.main(args);

    Prolong_fac_plus = 0;

    Prolong_fac_mal = v1;
    IllustrationsszenarioMethodenaufrufe.main(args);

    Prolong_fac_mal = v2;
    IllustrationsszenarioMethodenaufrufe.main(args);

    Prolong_fac_mal = v3;
    IllustrationsszenarioMethodenaufrufe.main(args);

    Prolong_fac_mal = 0;
  }
}
```

Listing 7.7: AspectJ: Prolongation von Szenario 1

7.1.7 Simuliertes Optimieren

Das simulierte Optimieren wurde in Kapitel 6 in [Man07] entwickelt und detailliert beschrieben und wird hier gekürzt wiedergegeben.

grundlegende Idee

Das simulierte Optimieren als eine Ausprägung des Analyseinstrumentariums wurde in Abschnitt 4.3.3 auf Seite 67 vorgestellt. Hier wurde die grundlegende Idee kurz präsentiert: alle Abläufe bis auf einen (oder wenige) werden um einen Faktor verlängert, in Folge erscheint dieser Ablauf relativ verkürzt.

Messung

Die Realisation bei der Instrumentierung ist nun folgende: das entsprechende Modul wird mit Code verwebt bzw. instrumentiert, dass die Laufzeit ausmißt und diese Laufzeit um einen Faktor Y verlängert prolongiert.

Widerspruch

deterministisches Programm

Hier offenbart sich jedoch ein Widerspruch, zu den in Kapitel 5 und Kapitel 6 gewonnenen Erkenntnissen: der Zustandsübergang der atomaren Befehle muss nach der Prolongation stattfinden, d. h. die *NOP*s müssen vor dem eigentlichen Befehl eingefügt werden. Hier kann einerseits als Vorbedingung für das simulierte Optimieren ein deterministisches Programm genommen werden. Unter einem deterministischem Programm wird eine Anwendung verstanden, bei der bei jeder Ausführung der nächste Befehl eindeutig bestimmbar ist. Somit kann in einem Programmlauf die Laufzeitdauer gemessen werden, im nächsten Programmlauf diese zusätzliche Zeit vor den Komponentenausführung hinzugefügt werden. Eine Diskussion, eine Lösung und ein Ausblick befindet sich in Abschnitt 7.4.

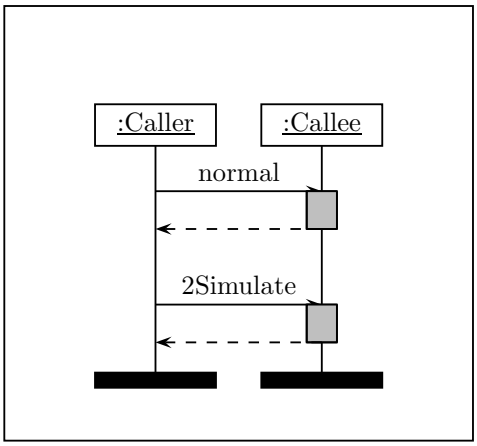


Abbildung 7.3: Didaktisches Beispiel für das simulierte Optimieren.
❑ Das Objekt CALLER ruft zwei mal eine Methode des Objektes CALLEE auf.
❑ Der Aufruf 2SIMULATE soll schneller simuliert werden, im Gegensatz zum Aufruf NORMAL.
Die Grafik wurde zitiert von [Man07, S. 106]

didaktisches Beispiel

Die Implementierung des simulierten Optimierens wird an einem didaktischen Beispiel eingeführt. In Abbildung 7.4 ist ein MSC-Diagramm (sehr ähnlich zu einem Sequenzdiagramm [SG98]) dargestellt, hier gibt es zwei Objekte CALLER und CALLEE, CALLER führt synchrone Aufrufe an CALLEE durch. Der zweite Aufruf mit der Beschriftung „2SIMULATE“ soll nun relativ verkürzt werden. [Man07]

Pointcuts

In der Abbildung 7.4 werden zwei *Joinpoints*, „NORMAL“ und „2SIMULATE“ dargestellt. Durch die umgekehrte Logik dieses Ansatzes darf die Methode „2SIMULATE“ nicht verändert werden. Es müssen *Pointcuts* für alle anderen Methoden der zu analysierenden Anwendung spezifiziert werden, wie dies beispielsweise in Listing 7.8 gezeigt wird. Die *AspectJ*-Direktive `&& ! cflow * *.2Simulate` verhindert, dass die zu untersuchende Methoden, die relativ optimierte Methode, verlängert wird. [Man07]

```
pointcut Verlaengern():  call (* *.* (..))
  && ! cflow * *.2Simulate (..)
  && ! within(Trace2)
  && ! ...weitere nicht simulierte Packages, Klassen etc.
```

Listing 7.8: Simuliertes Optimieren 1

`around` Klammermetapher

Mit diesem spezifizierten Pointcuts werden die Methoden umschlossen. Mittels eines `around`-Advices, den man sich wie eine Klammer vorstellen kann, wird die Startzeit gemessen, mittels eines `return proceed` die Steuerung an die eigentliche, verwebte Me-

thode weitergegeben. Danach wird durch die Zeitdifferenz die Laufzeitdauer der Methode bestimmt, durch eine Prolongation (mittels z. B. Busy-Waiting) wird die Methode verlängert.

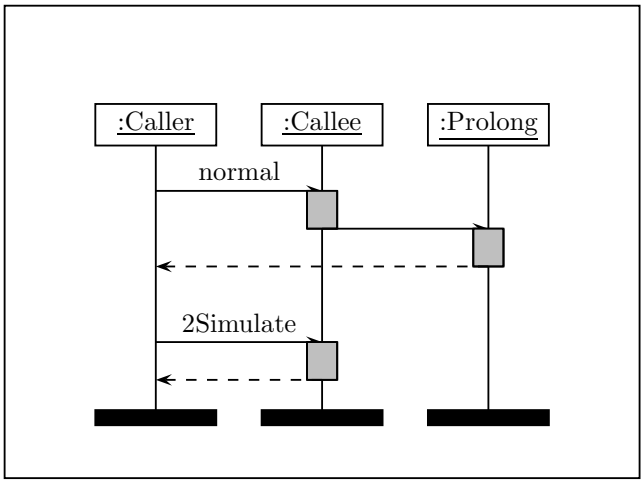


Abbildung 7.4: Didaktisches Beispiel für das simulierte Optimieren.
❑ Es wird ein Objekt Prolong eingeführt, dass die Verlängerung um einen Faktor übernimmt.
❑ Hier in der Abbildung wird einfach verlängert, es wird um die Laufzeit vom Aufruf Callee prolongiert.
❑ Als Resultat erscheint in diesem System der Aufruf von 2Simulate relativ um das doppelte schneller.
Die Grafik wurde zitiert von [Mano7, S. 107].

erneute Verlängerung

Jedoch reicht dieser Ansatz alleine noch nicht. Analog zu der Klammer-Metapher des `around`-Advices werden die Methoden mit der höchsten Stacktiefe als erstes verlängert. Dies wird präferiert, da es eine genaueres Abbild des simulierten Systems ergibt, jedoch summiert sich die Laufzeitdauer in Methoden mit etwas geringerer Stacktiefe auf. So würden bereits prolongierte Methoden die Simulation verfälschen. Illustriert ist dieser Effekt in Abbildung 7.5.

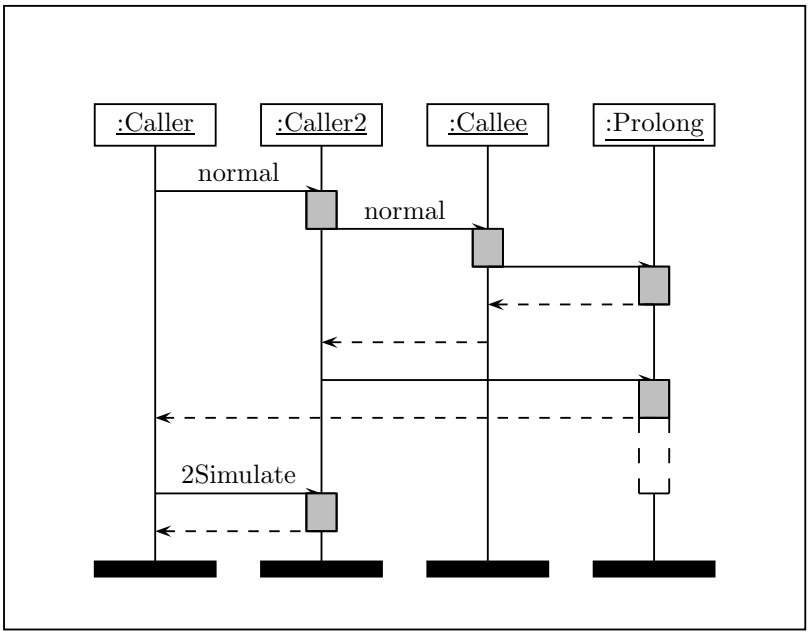


Abbildung 7.5: Didaktisches Beispiel für das simulierte Optimieren.
❑ Würden nur die Laufzeitdauer der zu prolongierenden Methoden gemessen und diese um einen Faktor verlängert, so gehen bisherige Laufzeiten auch ein.
❑ Im Bild ist diese inkorrekte Verlängerung durch den gestrichelten, weißen Balken gekennzeichnet. Die Methode Caller2 würde um die abstrakt dargestellten drei Zeiteinheiten (Callee + Prolong + die eigene Laufzeit) verlängert.
❑ Korrekt wäre nur eine Verlängerung um die Laufzeit von Caller2, Callee wurde bereits verlängert.
❑ Zur Lösung wird die Hilfsklassen `ThreadLogLinkedList` um einen Stackspeicher erweitert.
Die Grafik wurde zitiert von [Man07, S. 109].

Zur Lösung dieses Problems kann und wurde die Hilfsklassen `ThreadLog` um einen Stackspeicher erweitert. Die zusätzliche Funktion `exitTime` wurde in die Klasse `ThreadLogLinkedList` eingefügt. Beides ist in Abbildung 7.6 dargestellt. [Man07, S. 109]

Der komplette Code zum simulierten Optimieren befindet sich im Anhang B auf Seite 299.

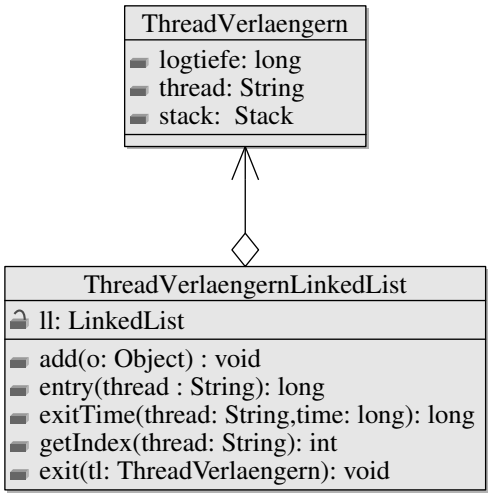


Abbildung 7.6: Klassendiagramm: Die zum simulierten Optimieren erweiterten Hilfsklassen ThreadLog und ThreadLogLinkedList.

❑ Zum korrekten Verlängern muss der Rückgabewert der Methode `exitTime` von der gemessenen Laufzeit der Methode abgezogen werden.

❑ Die Methode `exitTime` kann anstelle der Methode `exit` aufgerufen werden, analog wie diese passt sie den mitprotokollierten Thread und die Logtiefe an (siehe Abschnitt 7.1.2.1).

❑ Das komplette Listing befindet sich im Anhang B.

7.1.8 Nachteile der Instrumentierung für eine Experimentierumgebung

Der Ansatz zur Realisation der Experimentierumgebung durch Instrumentierung ist bequem und schnell, hat jedoch einige inhärente Nachteile.

7.1.8.1 Fehlende Pointcuts für Synchronized Blöcke

Kein Pointcut für synchronized Blöcke

AspectJ stellte zum Zeitpunkt von [Man07] keinen Pointcut für einen *synchronized Block* im Code zur Verfügung. Gerade jedoch der parallele Zugriff auf gemeinsame, gesperrte Objekte stellt einen Vorteil der Prolongation als Instrumentarium zur Performanzanalyse gegenüber dem klassischem Profiling dar[2].

Beispiel 7.1.1 synchronized Blöcke

```
public void aMethod() {
  synchronized(lock1) {
    synchronized(lock2) {
      synchronized(lock3) {
        ...
      }
    }
  }
}
```

Listing 7.9: Java: Verschachtelte Synchronized-Blöcke in Java

Pointcuts für synchronized Blöcke werden für diverse Anwendungsfälle propagiert [XHG08; BH08], sind aber momentan nicht in der aktuellen Version AspectJ 5 integriert.

7.1.8.2 Prolongation nicht feingranular genug

Aspektorientierung nicht feingranular

Die Prolongation geschieht durch die Schleife am Anfang einer Methode. AspectJ bzw. das Joinpoint-Konstrukt ist nicht feingranular genug, einzelne Operationen innerhalb von Methoden zu prolongieren. Zwar können mittels Pointcuts hierarchisch Methoden verwebt werden, dem ist durch die Benutzung von (nichtverwebten) Bibliotheken und Java Primitiven aber ein Ende gesetzt.

[2] siehe Abschnitt 7.1.8.3 auf Seite 161.)

7.1.8.3 Seiteneffekte durch unvollständige Kontrolle über die Hardware und Betriebssystem

unvollständige Kontrolle

Durch die zusätzlichen Schicht der Java Virtual Machine ist durch den Ansatz mit AspectJ eine unvollständige Kontrolle über die Hardware gegeben.

Hardwarekomponenten nicht prolongierbar

Beispielsweise soll eine Operation die auf eine Hardwarekomponente, z. B. die Festplatte, zugreift prolongiert werden. Nur das Ergebnis der Operation wird hinausgezögert, die Hardwarekomponente selbst kann (durch die zusätzliche, nicht instrumentierbare Schicht) nicht prolongiert werden. Greift ein anderer Ausführungsstrang nun auf die Hardwarekomponente zu, steht ihm diese wie gewohnt zur Verfügung.

charakteristische Ressourcennutzung

Um die Mächtigkeit der Prolongation als Analyseinstrumentarium in parallelen Systemen zu nutzen, v.a. um gleichzeitige Ressourcennutzung als Nadelöhr zu erkennen, müsste hier die Hardwarekomponente zusätzlich prolongiert werden. Die charakteristische Ressourcennutzung muss für eine ganzheitliche Prolongation während des Prolongationszeitraums erhalten bleiben.

Insbesondere ergeben sich durch eine prolongierbare Hardware weitere praktisch relevante Anwendungsszenarios wie dies in Kapitel 8 und in Kapitel 10 als Testmethodologie gezeigt wird.

7.1.9 Ausblick zur Prolongation mittels Instrumentierung

Reproduktion der Ressourcenauslastung

Da die Kernidee dieser Arbeit eine Kombination aus der Variation von Komponentenlaufzeiten und empirischen Versuchen ist, muss für diesen Anwendungsfall eine möglichst genaue Ressourcenauslastung reproduziert und die Zeitdauer der Prolongationen akkurat kontrolliert werden können. Der ganzheitlichen Ansatz durch Virtualisierung, bei dem die Kontrolle über die Hardware gegeben ist, wird folgend in diesem Kapitel beschrieben.

elegante Lösung

Dessen ungeachtet stellt die aspektorientiere Realisation der Prolongation eine praktikable Lösung für eine Analyse auf Codeebene dar, und hat trotz etwaiger Seiteneffekte und ungenügender Granularität ihre Existenzberechtigung – vor allem durch die schnelle, elegante und effiziente Realisation der Experimentierumgebung durch eine Instrumentierung des Codes.

7.2 *Virtualisierung – eine kurze Einführung*

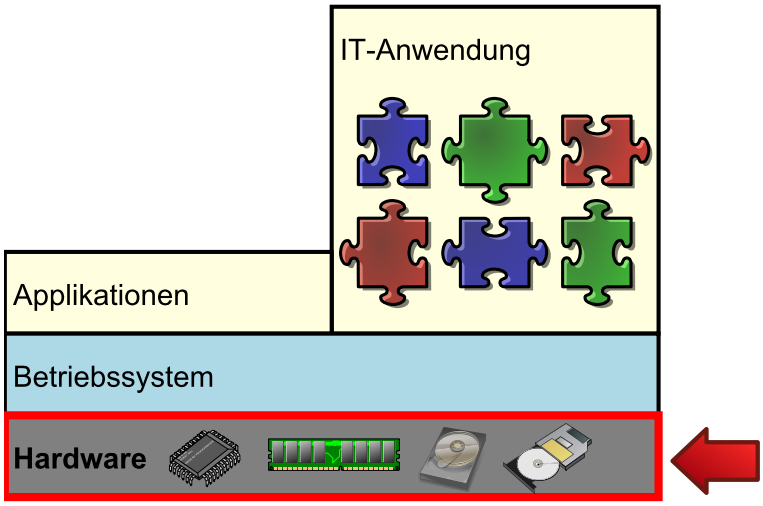


Abbildung 7.7: Die Experimentierumgebung (Prolongation und Logging) auf Hardwareebene, realisiert durch eine Virtualisierungslösung.
❑ Die zeitlich beeinflussten Elemente bei der Realisation der Experimentierumgebung durch eine Virtualisierungslösung sind die Hardwarekomponenten (im System durch den roten Pfeil gekennzeichnet).

Marketingschlagwort Virtualisierung

Virtualisierung hat sich im Marketing der Informationstechnik und IT-Industrie als ein sehr populäres Schlagwort etabliert. Der Begriff kann unterschiedlichste Ansätze umfassen:

- Virtualisierung bezeichnet eine Abstraktion von Ressourcen, z. B. bei der Virtualisierung von Speicher.
- Virtualisierung kann abstrahiert als ein *Isomorphismus* aufgefasst werden, der Funktionen des zu virtualisierenden Systems auf den realen Host abbildet.
- Als Virtualisierung versteht man eine Methode, mit der es möglich wird, die Ressourcen eines Computers in mehrfache Aufsführungsumgebungen (z. B. für Betriebssysteme) zu unterteilen. Hierzu werden unterschiedliche Konzepte und Technologien benutzt, beispielsweise Time-Sharing, partielle oder komplette Simulation von Hardware, Emulation und viele mehr.

In dieser Arbeit wird unter dem Termini Virtualisierung der letzte Punkt verstanden, der insbesondere die anderen Ansätze umfasst: ein Verfahren mit dem via Software mehrere Instanzen einer Betriebsumgebung auf einer Hardwareplattform geschaffen werden.
Im Folgenden wird zur Einordnung und zur Sensibilisierung der breiten Anwendungsmöglichkeiten in der IT ein kurzer historischer Überblick zur Virtualisierung gegeben. Danach folgt eine kurze Vorstellung der benötigten Technologien.

7.2.1 Historische Randnotizen zur Virtualisierung

Hype mit langer Vergangenheit

Virtualisierung entwickelt sich zu einem neuen Trend in Industrie und Wissenschaft, obwohl die Wurzeln relativ weit zurückreichen. Die theoretischen Grundlagen wurden 1959 von Strachey [Str59] mit dem Multitasking sowie 1957 von Bemer [Bem57] mit dem Time-Sharing gelegt.

Exkurs 7.2.1 Time-Sharing versus Multitasking
Hier sollte der konzeptionelle Unterschied zwischen Time-Sharing und Multitasking unterstrichen werden:

- **Multitasking** ist ein gleichzeitiges Bearbeiten von Prozessen. In [Str59] wurde der gedankliche Schritt von physischen zu logischen Betriebsmitteln vollzogen. Ein Prozess entspricht einem logischem Prozessor. Motivation hierfür war, dass durch einen Zugriff auf ein Peripheriegerät der Programmablauf im damaligen Batchbetrieb bei einem Einprozessorsystem blockiert war, was zu einer schlechten Ausnutzung der teueren CPU führte. Mittels eines Kontextwechsels kann ein anderer Prozess (logischer Prozessor) dem realen Prozessor zugewiesen werden, was zu einer höheren Effizienz (Performanz) führt.
- **Time-Sharing** ist die gleichzeitige Bedienung von Benutzern in dem die Ressourcen des Rechners geteilt werden (z. B. mittels Zeitscheiben) [Bem57].

virtueller Hauptspeicher

1962 wurde in Manchester das Rechnersystem *ATLAS* in Betrieb genommen, mit dem inzwischen üblichen Konzept des virtuellen Hauptspeichers, einer virtualisierten Hardwarekomponente [Kim+06, S. 608]. Das System 370 von IBM, vorgestellt im Sommer 1970, erlaubte das Paging [Kri07, S. 7].

Realisierte virtuelle Maschinen

Die Ursprünge der ersten implementierten virtuellen Maschine reichen zurück in das Jahr 1964 auf das dort gegründete IBM Forschungssystem *CP-40* [Var97, S. 10]. Hier waren *Virtual Machines* oder virtuelle Maschinen direkte Kopien der unterliegenden Hardware, eine Komponente namens *Virtual Machine Monitor* (siehe Abschnitt 7.2.2) lief direkt auf der Hardware und ermöglichte das Erzeugen von virtuellen Maschinen. Somit war es zum ersten mal möglich, verschiedene Instanzen eines Betriebssystems gleichzeitig auszuführen [DG08, S. 111].

p-machine zur Portabilität

Nach einem Vorschlag von *Kenneth Bowles* aus dem Jahre 1977 sollte eine Pseudo-Maschine (abgekürzt *„p-machine“*) als virtuelle Hardware die Portabilität (von Pascal) bei den vielfältiger werdenden Mikroarchitekturen sicherzustellen [SN05]. Hierbei wur-

de der Code in eine assemblerähnliche Zwischensprache übersetzt, ein Emulator bildet diesen auf die reale Hardware ab [SN05]. Software wird portiert, in dem ein *p-code Interpreter* für die spezifische Maschine implementiert wird [Wel83, S. 975].

Virtuelle Maschinen zur plattformunabhängigen Ausführung

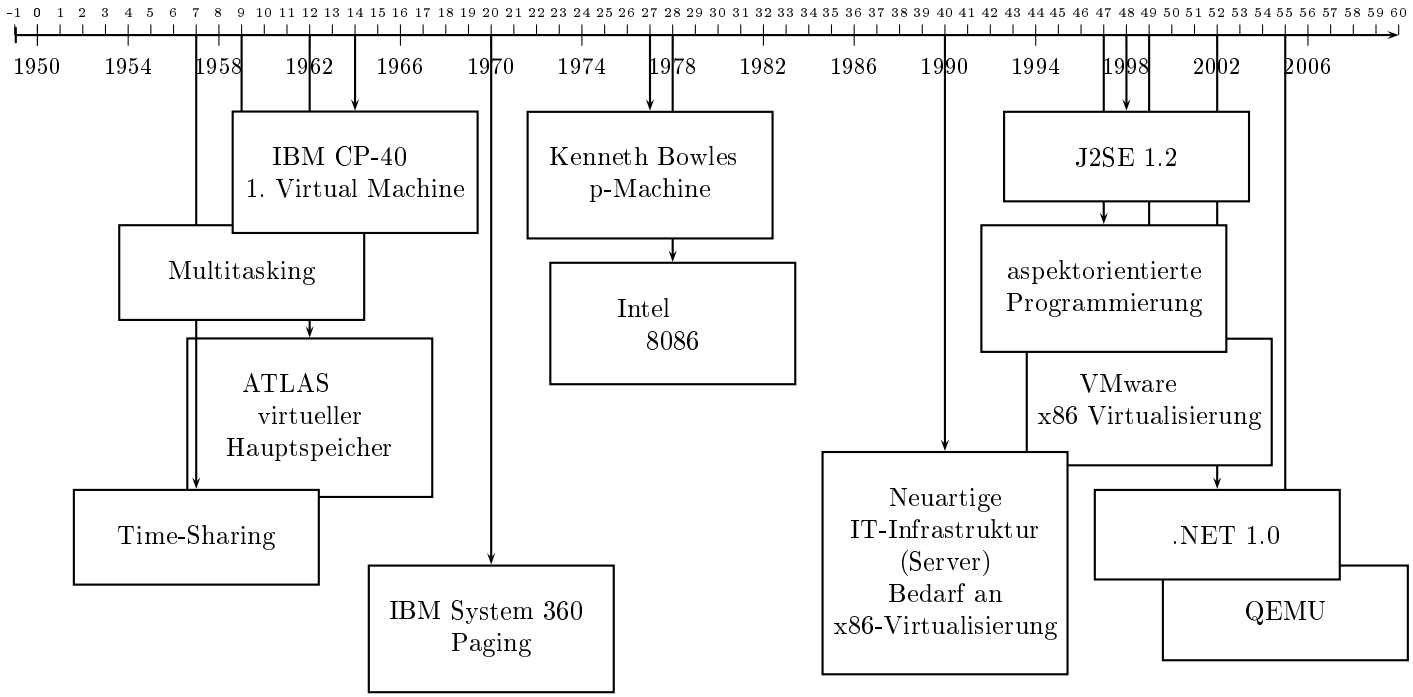
Somit ist die p-Maschine der Vorläufer vieler aktueller virtuellen Maschinen zur plattformunabhängigen Ausführung geworden, wie beispielsweise der *Java Virtual Machine* von *Sun* [LY99] oder *Microsofts Common Language Runtime* (CLR), der Laufzeitumgebung im Rahmen des *.NET-Frameworks* [BP02]. Insbesondere existieren auch Emulatoren für die in Kapitel 6 verwendete PRAM, an der die Prolongation theoretisch betrachtet wurde [Häm92].

Renaissance der Virtualisierung durch Serverdienste

Durch die sinkenden Kosten bei gleichzeitiger Leistungssteigerung der Hardware wurde die Partitionierung der Ressourcen aus Effizienzgründen hinfällig. Durch die sich weitgehend etablierenden Serverdienste wurde das Konzept der Virtualisierung jedoch wieder interessant, z. B. zur Hardwarekonsolidierung, also zur besseren Ausnutzung der Ressourcen einzelner Server, durch den erhöhten Administrationsaufwand der Serverdienste, zur Verbesserung der Sicherheit und vielem mehr.

x86 Virtualisierung

1998 gegründet präsentiert *VMware Inc.* 1999, basierend auf früheren Forschungen der Gründungsmitglieder an der *Stanford Universität* zur Virtualisierung von x86 Architekturen, *VMware Workstation 1.0* für Linux und Windows [Sou08, S. 12]. 1998 wurde ein Patent auf die verwendeten Techniken eingereicht [DBR02]. Die vorgestellte Lösung ist beachtenswert, da sie x86 Systeme auf x86 Systeme mit annehmbarer Performanz virtualisieren, obwohl sie nach den Kritierien von Popek und Goldberg [PG74], durch die spezielle Architektur der x86 Systeme, als nicht, oder nur schwer, virtualisierbar galten.


-1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
1950
1954
1958
1962
1966
1970
1974
1978
1982
1986
1990
1994
1998
2002
2006
IBM CP-40
1. Virtual Machine
Kenneth Bowles
p-Machine
J2SE 1.2
Multitasking
Intel
8086
aspektorientierte
Programmierung
ATLAS
virtueller
Hauptspeicher
VMware
x86 Virtualisierung
Time-Sharing
IBM System 360
Paging
Neuartige
IT-Infrastruktur
(Server)
Bedarf an
x86-Virtualisierung
.NET 1.0
QEMU

7.2.2 *Der Hypervisor oder Virtual Machine Monitor*

Virtual Machine Monitor oder Hypervisor

Ein *Virtual Machine Monitor* (VMM) oder *Hypervisor* ist Software bzw. eine Softwarekomponente, mit der es möglich wird, mehrere virtuelle Maschinen auf einem (realen) physischem Host auszuführen [Hic08]. Der VMM liegt zwischen dem virtualisierten Betriebssystem und der realen Hardware. Insbesondere ist der Hypervisor oder VMM verantwortlich für das Time-Sharing zwischen den einzelnen virtuellen Maschinen.

Exkurs 7.2.2 Hypervisor - Wortursprung

Wortursprung

Die Aufgabe des Virtual Machine Monitors zeigt auch die etymologische Bedeutung der bedeutungsgleichen Bezeichnung Hypervisor. „Hyper“ aus dem Griechischen bedeutet „über“, „Visor“ kommt vom Lateinischen und lässt sich von „videre“ als „sehen“ ableiten. Ein Hypervisor ist demnach ein Aufseher oder ein Überwacher (von Systemen).

7.2.3 *Virtualisierungslösungen*

Eine virtuelle Maschine als eigenständiges System

Die virtuellen Maschinen stellen sich als eigenständige Systeme dar. Hardwarekomponenten sind mittels Software nachgebildet und somit virtuell repliziert. In bestehende Virtualisierungslösungen werden die virtuellen Hardwarekomponenten so nachgebildet, dass die normale oder korrekte Funktionalität des realen Hardwarependants möglichst exakt emuliert wird.

Ansätze zur Realisierung

Es werden zwei generelle Ansätze zur Realisation von Virtualisierungslösungen unterschieden. *Typ-1* läuft direkt ohne eine weitere Softwareschicht direkt auf der Hardware [RS, S. 4]. *Typ-2* nutzt dabei ein Betriebssystem und dessen Gerätetreiber [Pei+04]. Daneben gibt es hybride Ansätze, die beispielsweise wegen Geschwindigkeitsvorteilen direkt auf die CPU zugreifen aber teilweise Treiber für die unterliegende Hardware benutzen. Bei einer *Paravirtualisierung* läuft ein modifiziertes Gastbetriebssystem um eine Virtualisierung zu ermöglichen [Bar+03]. Hierbei wird die Peripherie nicht emuliert, das Gastbetriebssystem „weiß“, dass es virtualisiert ist. Durch diesen Ansatz erhält man eine bessere Leistung.

relevante Virtualisierer

Diese Kategorisierungen werden im industriellen Umfeld, mit dem dazugehörigen Marketing, und der wissenschaftlichen Literatur nicht einheitlich benutzt. Virtualisierungslösungen mit einiger wissenschaftlichen und industriellen Relevanz sind: XEN [Bar+03], QEMU [Bel05; Bar06], VMware [Zim05], KVM [Hab08] und Virtual Box.

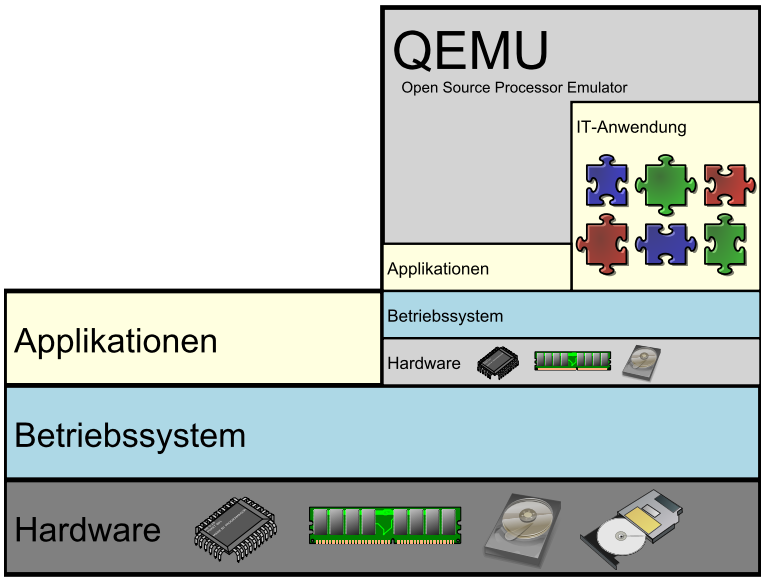


Abbildung 7.8: Virtualisierungslösung.
Hier im Bild dargestellt ist der Emulator QEMU, der als eigenständige Anwendung in der Applikationsebene läuft und Hardwarekomponenten durch eine Emulation virtuell repliziert.

7.3 Experimentierumgebung mittels Virtualisierung

Emulation möglichst exakter Funktionalität

In den existierenden Virtualisierungslösungen liegt der Schwerpunkt bei der Entwicklung der virtuellen Hardware, die als Software realisiert ist, darauf, nur die korrekte Funktionalität so exakt wie möglich nachzubilden. Die Sichtweise ist aber alleine auf die Funktionalität für den Normalfall beschränkt.

Qualitätsmerkmal Geschwindigkeitsvergleich

Die Zielsetzung bei Performanz-Eigenschaften von virtueller Hardware ist immer so schnell wie möglich. Der Geschwindigkeitsvergleich zwischen virtueller Hardware und der darunterliegenden realen Hardware wird vielfach als ein Qualitätsmerkmal der Virtualisierungslösung gesehen.

Experimentierumgebung

In dieser Arbeit wird zur Realisation der Experimentierumgebung ein virtueller Hosts, bei dem die Performanz von Hardwarekomponenten künstlich verlangsamt und die Auswirkungen protokolliert werden kann, implementiert. Dies eröffnet völlig neue Möglichkeiten, da der Ressourcenverbauch nicht mehr fest definiert ist, sondern relativ und dynamisch verändert werden kann.

Performanz der Hardwarekomponenten einstellbar

Diese Zielsetzung von Virtualisierern wird nun, zu Analysezwecken, konterkariert in dem Sinn, dass nun die stufenlose und selektive Langsamkeit, als Performanz- bzw. Leistungsminderung, der virtuellen Hardware zu einer zentralen Eigenschaft wird und diese Möglichkeiten explizit in den Hypervisor eingebracht werden. Es werden also die virtualisierten Hardwarekomponenten so ergänzt, dass deren Performanz individuell und dynamisch eingestellt werden kann.

7.3.1 Der Maschinen-Emulator QEMU

Gründe für QEMU

QEMU ist eine freie[3] virtuelle Maschine, Fabrice Bellard[4] beschreibt QEMU als schnellen Maschinen-Emulator [Bel05, S. 41]. Bei einer Emulation werden nicht die internen Zustände detailgenau nachgebildet, sondern das Ergebnis wird reproduziert. Die Begriffe Emulation und Virtualisierung werden oft synonym genutzt, einige Emulatoren werden als Virtualisierungslösungen bezeichnet.

Emulation und Virtualisierung

Emulator

Ein Emulator bildet jedoch **komplett** eine Architektur, die von der Hostarchitektur abweichen darf, in Software ab. Das heisst, die emulierten Hardwarekomponenten sind Datenstrukturen die vom Emulator verändert werden. Der Maschinencode wird geladen, vom Emulator decodiert und die Datenstrukturen modifiziert [Kri07, S. 27]. Die Decodierung und Modifizierung geschieht auf dem Hostsystem, der Emulator/Virtuali-

[3] QEMU als Ganzes ist als *GNU General Public License* [Gou09], Teile von QEMU sind unter spezifischen Lizenzen, die kompatibel zur GNU General Public License sind, lizensiert [Bel08]. Jede Datei des Quellcodes enthält deshalb spezifische Lizenzierungsinformationen.

[4] Fabrice Bellard ist französischer Mathematiker und Softwareentwickler. Er ist neben QEMU Schöpfer von tinyC [Lin05], Hauptentwickler von FFmpeg, einem Open Source Multimediaprojekt, und hält zwei mal den Rekord bei der Berechnung der Dezimalstellen der Kreiszahl Pi [Del99, S. 161][Spi10].

sierer läuft als Prozess auf dem Hostsystem, so werden die Befehle der unterschiedlichen Hardwarearchitekturen transformiert. Emulatoren sind QEMU [Bel05], Bochs [Law96] und spezielle Emulatoren für ältere Rechnerarchitekturen wie beispielsweise für den Commodore 64 (z. B. [Kro00]).

Systememulatoren

Hierbei gibt es komplette Systememulatoren wie beispielsweise DosBOX [Bli+08] dessen Geschwindigkeit einstellbar ist.[5] Bei anderen Emulatoren (wie Bochs [Law96] oder QEMU [Bel05; Bar06] muss oder kann ein Betriebssystem ausgeführt oder installiert werden.

Gründe für QEMU

Neben dem frei zugänglichen Quellcode, der verändert werden darf, dem frei wählbaren Betriebssystem, emuliert QEMU die Komponenenten komplett, was zur Wahl von QEMU zur Realisation der virtuellen Maschine mit variierbaren Performanzeigenschaften geführt hat.

7.3.2 Nachteile von QEMU

Emulationsoverhead

Durch die dynamische Translation hat QEMU, im Gegensatz zu Typ-1 Virtualisierungslösungen, erhebliche Geschwindigkeitseinbußen (10-20% der nativ kompilierten IT-Anwendung bzw. eine fünf bis zehnfache Verlangsamung (*slowdown*))[BA08, S. 186]. Die Performance kann jedoch verbessert (vgl. dazu z. B. [Hu+09]) oder durch schnellere Hardware kompensiert werden.

Desweiteren leidet das Projekt QEMU unter einer wirklich schlechten oder gar nicht vorhandenen Dokumentation.

7.3.3 QEMU - Aufbau

Exkurs 7.3.1 Linux User Mode Emulator

Linux User Mode Emulator

QEMU integriert auch einen spezifischen Linux User Mode Emulator, eine Teilmenge des Maschinenemulators in dem Linux Prozesse für einen Zielprozessor (target CPU) auf einem anderen Prozessor ablaufen [Bel05, S. 41]. Hier werden momentan folgende Prozessoren unterstützt: ARM, CRIS, m68k (Coldfire), MIPS, SPARC und x86. Weitere Architekturen werden momentan getestet oder sind in der Entwicklung [Bel09]. Motivation diesbezüglich ist es, Kompilate für andere Systeme (Cross Compiling) zu testen [Bel05, S. 41].

Hauptaufgabe von QEMU ist jedoch die Virtualisierung zum Ausführen eines Betriebssystems auf einem anderen Betriebssystem [Bel05, S. 41].

7.3.3.1 Subsysteme von QEMU

Subsysteme von QEMU

QEMU besteht aus folgenden Subsystemen:

[5] Hierbei kann die CPU so verlangsamt werden, dass bestimmte DOS Spiele auf modernen Computerarchitekturen spielbar sind. Siehe `http://www.classicdosgames.com/tutorials/dosbox.html`

❑ **CPU Emulator:**
Es können Prozessoren von folgenden Architekturen emuliert werden: *x86*, *PowerPC*, *ARM* und *SPARC* [Bel05, S. 41], in der aktuellen Fassung *m68k (Coldfire)*, *MIPS*, *MIPS64* und die 64-Bit Version des *x86* [Bel09].

❑ **Emulierte Hardwarekomponenten:**
z. B. *VGA Display*, *digitaler Schnittstellenbaustein 16450*, *PS/2 Maus* und *Tastatur*, *IDE Festplatte* und eine *NE2000 Netzwerkkarte* [Bel05, S. 41].

❑ **Generische Komponenten:**
zeichenorientierte Geräte (*character devices*), blockorientierte Geräte (*block devices*) und Netzwerkgeräte (*network devices*) [Bel05, S. 41].

❑ **Architekturbeschreibungen:**
zum Instantiieren der emulierten Geräte [Bel05, S. 41].

Sowie einen *Debugger* und eine *Benutzerschnittstelle*.

7.3.3.2 Prolongation für x68 Mikroarchitekturen

De facto Standard x86 Architektur

Gemessen am Marktanteil ist Intel der unangefochtene Spitzenreiter bei der Herstellung von Prozessoren für nicht eingebettete Systeme (*embedded Systems*) [Sta+07, S. 45]. 30 Jahre nach der Einführung dominiert hier die x86 Architektur nach wie vor [Sta+07, S. 46]. Aufgrund dieser Dominanz wird in dieser Arbeit die die Prologongation praktisch auf der Mikroarchitekturebene eines x86 (PC-)Systems demonstriert.

Nach Warnke und Ritzau [WR09] wird bei QEMU Version 0.9.1 folgende Hardware für eine *x86* Architektur emuliert:

- Symmetrisches Multiprozessorsystem (SMP), bis zu 255 Prozessoren
- PC-Bus: PCI und ISA-System (i440FX Host PCI Bridge und PIIX3 PCI to ISA Bridge)
- Zwei PCI-ATA-Schnittstellen, Unterstützung für maximal vier (virtuelle) Festplatten
- PC-BIOS von Bochs
- Grafikkarte (Cirrus CLGD 5446 PCI VGA-Karte oder Standard-VGA-Grafikkarte mit Bochs-VESA-BIOS-Extension)
- Netzwerk (NE2000 PCI-Netzwerkadapter)
- CD-/DVD-Laufwerk
- Diskettenlaufwerk
- USB-Controller und virtueller USB-Hub
- Parallel-Port
- Serielle Ports
- PS/2-Tastatur und PS/2 Maus
- Eingebauter PC-Lautsprecher

- Soundkarte (Soundblaster 16, ES1370 PCI)

prolongierbare Komponenten

Alle diese Komponenten können nun prolongiert werden, wobei eine Prolongation einer Komponente wie beispielsweise des eingebaute PC-Lautprechers zur Fehler- oder Performanzanalyse wenig sinnvoll ist.

keine generelle Lösungsmöglichkeit

Die Realisation der Prolongation und der Protokollierung kann nicht so generell und damit nicht so elegant erfolgen wie bei den *Cross Cutting Concerns* mit AspectJ. Deswegen wird hier repräsentativ die Prolongation an der Festplatte demonstriert, die realisierte Experimentierumgebung wird QEMU^{dt} genannt.

Aufwind für QEMU

In letzter Zeit erlebt QEMU wieder einen beträchtlichen Aufwind. So basiert der Android Emulator (*Android* [Sha+10] ist ein Betriebssystem oder Framework von Google speziell für mobile Geräte) des *Android SDK*s beispielsweise auf QEMU. *Google* führt sein Programmierstipendium *Summer of Code* mit QEMU als unterstütztes Projekt durch.

Die interne Struktur von QEMU hat sich seit Beginn dieser Arbeit rapide geändert, weitere starke Änderungen sind zu erwarten. Die Prolongation wird an QEMU Version 0.9.1 demonstriert.

7.3.4 *QEMU^{dt}*

QEMU^{dt}

QEMU^{dt} bezeichnet in dieser Arbeit den erweiterten Prozessoremulator QEMU. Mittels QEMU^{dt} ist man in der Lage, einzelne Hardwarekomponenten zu verlangsamen oder (relativ) zu beschleunigen. Desweiteren kann in QEMU^{dt} ein *Zeitraffer* oder *Time Warp* ausgeführt werden. In der virtualisierten Umgebung vergeht die *wall clock time*, die physikalische Zeit, extrem viel schneller. Durch eine Kombination der in Abschnitt 7.1 vorgestellten Technik können alle Elemente eines Systems beinflußt werden, so dass diese langsamer oder (relativ) schneller sind.

7.3.5 *Entwicklungsumgebung für QEMU^{dt}*

Entwickelt und kompiliert wurde auf einem *Ubuntu 10.04 LTS* und auf einem *Suse Linux 10.3* System. Prinzipiell ist die Entwicklung unter einem Windows System möglich, jedoch muss hier auch noch der *gcc* (*Gnu C Compiler*) installiert werden.

7.3.6 *Abhängigkeiten*

Folgende Abhängigkeiten werden zur Kompilierung noch benötigt:

- SDL
- SDL-devel
- tetex

Suse Linux

`SDL` und `SDL-devel` kann mittels *Yast* installiert werden (im Registerblatt Computer auf dem Menübutton aus der Taskleiste, Administrator Settings (Yast)). Dort kann nach `SDL` gesucht werden und mittels den Kombinationsfeldern und Accept `SDL` und `SDL-devel` installiert werden. Abhängige Pakete wurden gleich mitinstalliert.

Ubuntu Linux

Um *SDL* und *SDL-devel* in *Ubuntu Linux* zu installieren wird in einem Terminal folgendes eingegeben:

```
sudo apt-get install libsdl1.2-dev libsdl1.2debian
```

Listing 7.10: Shell: Installation von SDL und *SDL-devel*

Alternativ wird in einem *Paketmanager* nach den fehlenden Paketen gesucht und diese installiert.

7.3.7 *Download und Übersetzung*

Kompilierung

Ältere QEMU-Versionen konnten nur mit einem *GCC3* (Gnu C Compiler) kompiliert werden. Die aktuellen Versionen *qemu-0.12.5*, *qemu-0.12.4* und *qemu-0.11.1* ließen sich fehlerfrei mit dem *GCC4* (*gcc-4.4*) kompilieren.
Die aktuelle Version (*qemu-0.12.5*) kann wie folgt geladen, entpackt und installiert werden.

```
# download qemu
wget http://download.savannah.gnu.org/releases/qemu/qemu-0.12.5.tar.gz
# unzip it
tar zxvf qemu*tar.gz
# configure it
cd qemu*
./configure --target-list=i386-softmmu
#alternativ:
#./configure --target-list=x86_64-softmmu
#./configure
# build qemu
make
# install qemu
sudo make install
```

Listing 7.11: Shell: Download und Build von QEMU

7.3.8 Prolongation bei QEMU Version 0.9.1

Die Datei Monitor.c wurde zur Steuerung der virtuellen Maschine um folgenden C-Code erweitert:

```
static term_cmd_t term_cmds[] = {
 ...
 {"do_prolong_HD_write" , "i" ,do_prolong_HD_write ,
  "sleepvalue" , "set␣IDE␣HD␣write␣␣prolong␣value␣in␣ms
␣␣(non-negative␣integer),␣negative␣value␣prints␣actual␣setting" } ,
 ...
};
...
static void do_prolong_HD_write ( int sleepvalue )
{
  if ( sleepvalue < 0)
    term_printf ( "Current␣CPU␣ADD␣␣sleep␣value␣in␣␣ms␣\%d\n"
,prolong_HD_write);
  else
    prolong_HD_write = sleepvalue;
 }
 ...
```

Listing 7.12: C: Erweiterung in Monitor.c zur Prolongation des schreibenden Festplattenzugriffs

Über die gemeinsame Variable (`prolong_HD_write`) kann nun die Prolongation einer atomaren Hardwareoperation der virtuellen Maschine verlängert werden. Beispielsweise wurde zum Verlängern der Festplattenoperationen ein Aufruf der unten definierten Funktion in die entsprechenden Operationen (`ide_ioport_write` oder `ide_data_readw`) in der jeweiligen C-Datei (`ide.c` bei QEMU Version 0.9.1 oder `hw/ide/core.c` bei QEMU Version 0.12.5) eingefügt: Prolongation

```
void sleepwait_read (void)
 if ( prolong_HD_read >0){
 sigset_t sis, osis;
 sigfillset (&sis);
 sigprocmask (SIG_BLOCK, &sis, &osis );
 nanosleep (prolong_HD_read, 0) ;
 sigprocmask (SIG_SETMASK, &osis, NULL) ;
 }
```

Listing 7.13: C: Funktion sleepwait_read zur Prolongation eines lesenden Festplattenzugriffs

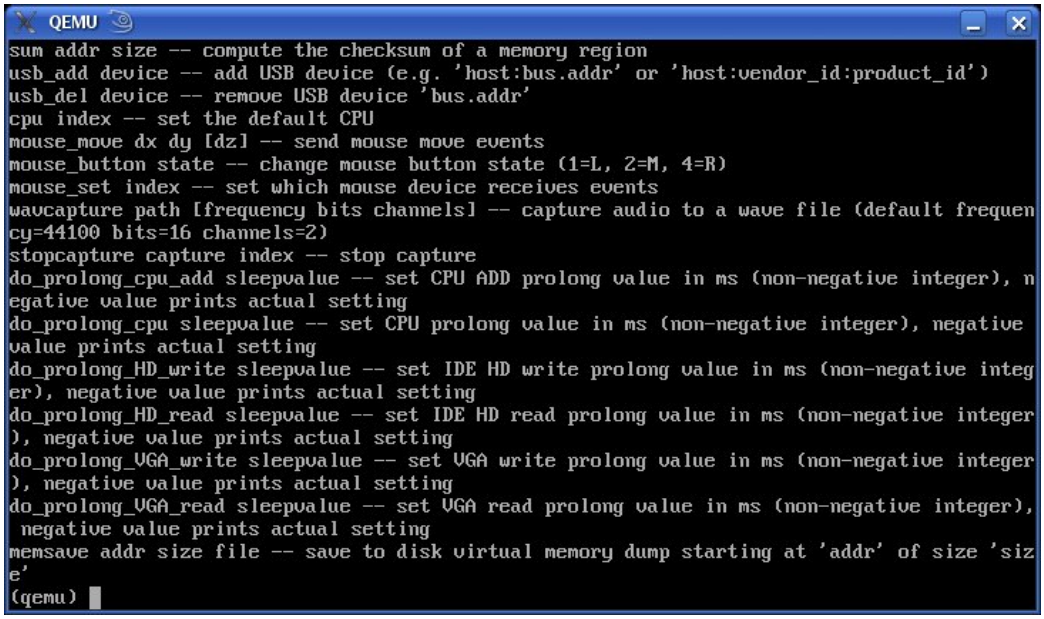


Abbildung 7.9: Der erweiterte QEMU^{dt} Monitor.

❑ Der QEMU bzw. QEMU^{dt} Monitor kann durch gleichzeitiges Drücken von Ctrl + Alt + 2 (`STRG + ALT + 2`) erreicht werden.

❑ Hier im Screenshot sind zusätzlich eingefügte Schnittstellen zur Prolongation sichtbar. Durch ein `do_prolong_VGA_write` wird beispielsweise die Schreibgeschwindigkeit auf die VGA Karte reduziert.

❑ Diese Version von QEMU^{dt} hat mit `do_prolong_cpu_add` die Möglichkeit die Addition des Prozessors zu verlangsamen.

Analog wird eine gemeinsame Variable (`prolong_HD_write`), eine Ansteuerung in der Benutzer- bzw. Betriebssystemschnittstelle `Monitor.c` sowie die folgende Funktion definiert (siehe Listing 7.14), wenn eine Prolongation des schreibenden Zugriffs stattfinden soll. So die virtuelle Maschine weiter verändert werden und weitere Hardwarekomponenten prolongiert werden. Prolongation II

```
void sleepwait_write (void)
{
  if ( prolong_HD_write >0){
  sigset_t sis, osis;
  sigfillset (&sis);
  sigprocmask (SIG_BLOCK, &sis, &osis );
  nanosleep (prolong_HD_write, 0) ;
  sigprocmask (SIG_SETMASK, &osis, NULL) ;
  }
}
```

Listing 7.14: C: Funktion sleepwait_write zur Prolongation eines schreibenden Festplattenzugriffes

Um den Umfang nicht zu sprengen wurde die Prolongation exemplarisch am schreibenden Festplattenzugriff präsentiert. Weitere emulierte Hardware kann analog prolongiert werden, hierzu müssen „nur“ die entsprechenden Funktionen identifiziert werden, was durch die schlechte Dokumentation des Projektes teilweise ein langwieriger Prozess ist.

7.3.9 Logging

Das Logging auf Modulebene kann, wie oben beschrieben, durch Instrumentierung erfolgen. Durch ein Mount einer Transferpartition (siehe Abschnitt C.7) können Daten vom Gastbetriebssystem zum Hostbetriebssystem ausgetauscht werden. Nach dem Schließen der Instanz stehen die Daten, analog wie in Abschnitt 7.1 beschrieben, im Hostbetriebssystem zur Verfügung. Instrumentierung Filetransfer

Mit der Startoption `-d` kann ein QEMU Logging auf Hardwareebene in das File `/tmp/qemu.log` aktiviert werden [WR09]. Mögliche Optionen können mit dem Schalter `-d ?` angezeigt werden. Das Logging auf Hardwareebene wurde in dieser Arbeit nicht betrachtet, da die Analyse der Systemperformanz in Abhängigkeit von den Leistungskenngrößen der Hardware in Kapitel 8 gezeigt werden. Es wird hierfür ein Simulations-

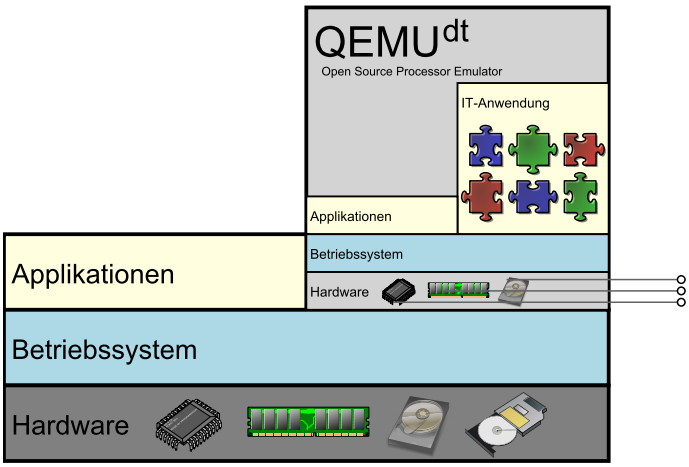


Abbildung 7.10: QEMU^{dt} als schematische Darstellung.
❑ Der Emulator ist eine eigene Anwendung auf dem Hostbetriebssystem.
❑ QEMU^{dt} emuliert Hardware, ein Betriebssystem (Gast) in der IT-Anwendungen gestartet werden können.
❑ Mit den hier beschriebenen Änderungen wurde QEMU auf QEMU^{dt} so erweitert, dass individuell die Performanzeigenschaften von Hardwarekomponenten beinflußbar sind. Auf das entsprechende Interface kann vom Hostbetriebssystem oder vom QEMU^{dt} Monitor zugegriffen werden.

ansatz[6] genutzt, die Interaktionen der Hardwarekomponenten sind nicht so komplex und wurden nicht mittels strukturentdeckenden[7] oder modellbildenden Analysemethoden[8] untersucht.

7.3.10 Der Zeitraffer bzw. Time Warp

Zeitraffer

Ein *Zeitraffer* (*Time Warp*), analog zum simulierten Optimieren, kann mit QEMU sehr einfach realisiert werden. In der Datei `qemu-timer.h` ist eine Konstante gespeichert, die die Anzahl der Ticks pro Sekunde wiedergibt.

```
static inline int64_t get_ticks_per_sec(void)
{
    return 1000000000LL;
}
```

Listing 7.15: C: Time Warp bzw. Zeitraffer

Time Warp

Inkrementiert man diese Konstante läuft die Zeit innerhalb der virtuellen Maschine langsamer. Dekrementiert man diese, hat man einen Zeitraffer oder Time-Warp realisiert. Analog zur Prolongation wurde ein Monitorbefehl mit dem Namen `Time Warp` realisiert werden. Eine Durchführung des Time Warps / Zeitraffers an Szenario 8, angelehnt an den Aging related fault bei der Patriot Missile, befindet sich in Abschnitt 10.4.1 auf Seite 273.
Durch ein Dekrementieren der Konstante kann der gleiche Effekt wie er in der Arbeit „*To Infinity and Beyond: Time-Warped Network Emulation*“ Gupta u. a. [Gup+05] beschrieben ist, realisiert werden. Gupta u. a. [Gup+05] nutzen diesen *Time Warp* für ihre empirische Netzwerktests.

7.3.11 Ausblick und Erweiterungen von QEMU^{dt}

kein Isomorphismus

Man könnte mit dem vorgeschlagenen Ansatz mit der Virtualisierungslösung argumentieren, dass das Experiment eine Simulation oder Emulation des realen Systems ist. Die virtuell, durch Software replizierte, Hardware stellt nicht wirklich einen *Isomorphismus* des Systems dar. Beispielsweise können *Viren* oder *Malware* erkennen, dass sie in einer virtualisierten Umgebung ausgeführt werden [Cra+06].

Trend

Die Virtualisierung entwickelt sich zu einem neuen Trend in Industrie und Wissenschaft [Mce02]. Dem Autor ist keine Anwendung bekannt, die sich nicht virtualisiert ausführen ließ.

[6] siehe Abschnitt 9.2.2, Seite 206
[7] siehe Abschnitt 9.3, Seite 208
[8] siehe Abschnitt 9.4, Seite 215

In diesem Kapitel wurde die Basis für eine Experimentierumgebung durch eine Virtualisierung gelegt, insbesondere wurde gezeigt, dass sich die Experimentierumgebung praktisch realisieren lässt.

Folgende Erweiterungen würden die Experimentierumgebung noch weiter aufwerten.

- Die Integration beider Ansätze:
  - Auf höchster Ebene im System wurde die Experimentierumgebung durch eine Instrumentierung realisiert. Performanzabhängigkeiten, Optimierungskandidaten und -potenzial der Module können so identifiziert werden.
  - In der untersten Ebene des Systems wurde die Experimentierumgebung durch virtualisierte Hardwarekomponenten realisiert.
  - ⇨ Erstrebenswert wäre eine Kombination beider Ansätze. Somit kann die Instrumentierung der Module umgangen werden und beliebige Systeme können – ohne dass der Code instrumentierbar vorliegen muss – untersucht werden.
  - ⇨ Dies könnte beispielsweise mit dem *Tracing Framework DTrace* [CSL04; PDL06; BW08a] realisiert werden oder durch eine Identifikation von *Prolog* und *Epilog* von Funktionen geschehen [CP04, S. 60]. Vigna [Vig07, S. 5] stellt diese Methode bei *obfuscated Code* (verschleiertem Programmcode) vor.
  - ⇨ Eine vielversprechende Erweiterung wäre beispielsweise eine Synthese aus aspektorientierter Programmierung und Virtualisierung. Mit dieser aspektorientierten virtuellen Maschine wären weitere interessante Anwendungsfälle möglich, beispielsweise „green“ Software, Rekonfiguration der Hardware durch die Software, Hardware die sich bei Bedarf austauschen kann und viele mehr.

Mit diesem Framework wäre es möglich, in zielgerichteter Art und Weise die Performanzeigenschaften einzelner Module (mit Hilfe der virtuellen Maschine) zu variieren.

weitere Vorteile

Insbesondere wäre mit diesem Framework das simulierte Optimieren sehr schön zu realisieren. Da hier wirklich auf Mikrobefehlsebene prolongiert werden kann, wäre dieser Ansatz „genauer“ und die Prolongation kann so vor dem Zustandsübergang erfolgen. Mit diesem Ansatz müsste keine Analyse (siehe Kapitel 9) erfolgen, da Auswirkungen einer Verbesserung ohne Seiteneffekte (relativ) ausgemessen werden könnten.

Testframework

- Aufbauend auf diesem Framework kann ein Testframework zum Testen des Systems auf Synchronisationsfehler (Heisenbugs, Races, etc. – siehe Kapitel 10) implementiert werden:
  - Das Testframework dient zur Ansteuerung der virtuellen Maschine zum Test des Systems.
  - Hier wird der Begriff Testframework mit folgender Bedeutung verwandt:

  - Das Testframework ist eine Infrastruktur, die das Ansteuern der virtuellen Maschine prinzipiell ermöglicht und die Korrektheit des Systems validieren oder falsifizieren kann.
  - Eine Automatisierung der Tests des Systems ist mittels des Testframeworks möglich.
  - Der Testentwickler implementiert die Tests mittels des Testframeworks.
  - Das Testframework ist eine Erweiterung bestehender *xUnit Testframeworks* [Mes07], falls diese für die Sprache vorhanden sind und der Code vorliegt. Ansonsten kann für diese Sprache ein Testframework implementiert werden.

- ❑ Ist der Testcode nicht vorhanden, können durch ein *reverse engineering / Disassemblierung* oder direkt aus der Maschinensprache spezielle Module (sowie evtl. Submodule) identifiziert werden. Dies kann analog durch eine Identifikation von *Prolog* und *Epilog* von Funktionen geschehen [CP04, S. 60].
- ❑ Durch ein weiteres Reverse Engineering können, wenn nötig, die relevanten Daten in Registern/Speicher identifiziert werden auf die geprüft werden müssen. Durch das Variieren der Laufzeiteigenschaften der einzelnen Module kann nun getestet werden, ob sich diese Daten in den einzelnen Läufen unterscheiden und ob ein Fehlerfall vorliegt.

7.4 Zusammenfassung und Beantwortung der Teilfragestellung η

In diesem Kapitel wurde demonstriert, wie eine Experimentierumgebung realisiert werden kann. Diese wurde in den unterschiedlichen Ebenen eines Systems implementiert.

oberste Ebene — Auf der obersten Ebene des Systems, ähnlich der *Anwendungsschicht* (*Application Layer*) des OSI-Referenzmodells [Zim80; Int96] wurde zur Umsetzung der Experimentierumgebung eine Instrumentierung genutzt.

AspectJ — *AspectJ* stellt hier durch die querschnittlichen Belange oder *Cross-Cutting-Concerns* eine moderne und elegante Möglichkeit zur Instrumentierung dar [Sok06].
In Abschnitt 7.1.1 wurde AspectJ als Spracherweiterung für *Java* (durch die neuen Konstrukte *Joinpoint*, *Pointcut* und *Advice*) eingeführt.

Logging — In Abschnitt 7.3.9 wird demonstriert, wie die Ergebnisse eines Experimentes protokolliert werden können. Es wurde gezeigt wie mittels *AspectJ* eine Protokolldatei angelegt werden kann. Darauf aufbauend wurden Spezialaspekte wie das *Logging* bzw. das *Tracing* bei parallelen Anwendungen integriert.

Prolongation — Von Abschnitt 7.1.3 bis zum Abschnitt 7.1.6 wird gezeigt, wie das Analyseinstrumentarium, die Prolongation, mittels *AspectJ* realisiert werden kann. Hierfür werden zwei unterschiedliche Strategieen betrachtet: eine Prolongation durch *Blockierung* oder einem *Busy Waiting*. Die Vor- und Nachteile beider Ansätze wurden diskutiert. Exkurs 7.1.1 behandelt eine mögliche Optimierung von IT-Applikationen durch eine Blockierung.

simuliertes Optimieren — In Abschnitt 7.1.7 wurde die Realisierung des simulierten Optimierens durch Instrumentierung demonstriert. Um ungewünschte vielfache Verlängerungen zu vermeiden, musste eine entsprechende Datenstrukturen (ein Stack) eingefügt werden.

Nachteile — In Abschnitt 7.1.8 wurden die Nachteile des Ansatzes betrachtet, insbesondere die fehlenden Pointcuts (z. B. für *synchronized* Blöcke), die grobe Granularität (Methodenebene) und die unvollständige Kontrolle über die Hardware und die sich daraus ergebenden Seiteneffekte.

Virtualisierung — In Abschnitt 7.2 wird auf die Virtualisierung eingegangen. Motiviert und sensibilisiert wird der momentane Hype mit seiner über 50 jährigen Vergangenheit. Der *Hypervisor* als zentrale Komponente wurde eingeführt und unterschiedliche Virtualisierungslösungen und -produkte betrachtet.

Experimentierumgebung durch Virtualisierung — In Abschnitt 7.3 wird der Ansatz dieser Arbeit, eine Experimentierumgebung realisiert durch eine Virtualisierung, diskutiert. *QEMU* als freie virtuelle Maschine oder Prozessoremulator wird vorgestellt (Abschnitt 7.3.1 – Abschnitt 7.3.3). Auf den Aufbau von *QEMU*, auf die integrierten Subsysteme und auf die Nachteile von *QEMU* durch den Emulationsoverhead wird eingegangen.

QEMU^{dt} — In Abschnitt 7.3.4 wird QEMU^{dt} vorgestellt. QEMU^{dt} ist die Erweiterung von *QEMU* um das Analyseinstrumentarium. In Abschnitt 7.3.5 wird dokumentiert von welchen Quellen *QEMU* geladen und wie *QEMU* kompiliert werden kann.

Die Realisation des Analyseinstrumentariums kann bei *QEMU* nicht so generisch wie mittels *AspectJ* erfolgen. Abschnitt 7.3.8 demonstriert dies exemplarisch am lesenden und schreibenden Festplattenzugriff von *QEMU* Version 0.9.1.
Auf das Logging bei QEMU^{dt} wird kurz in Abschnitt 7.3.9 eingegangen.
Eine interessante Erweiterung, der *Zeitraffer* oder *Time Warp*, wurde in Abschnitt 7.3.10 demonstriert. Somit wird es möglich, die Zeit in der virtuellen Maschine gegenüber der physikalischen Zeit (*wall clock time*) schneller oder langsamer ablaufen zu lassen.

Analyseinstrumentarium

Zeitraffer

Somit kann die Teilfragestellung η, nach der **praktischen Durchführbarkeit**, positiv beantwortet werden. Mittels einer Instrumentierung oder Virtualisierung kann das in dieser Arbeit vorgeschlagene Experiment realisiert werden. ✔

Teilfragestellung η

Kapitel 8

Mindestanforderungen

„The only conceivable way of unveiling a black box, is to play with it."

Rene Thom

Übersicht des Kapitels

Dieses Kapitel beschreibt, wie das Analyseinstrumentarium bzw. die Experimentierumgebung dazu genutzt werden kann, um Mindestanforderungen an die Hardware zu eruieren. Abschnitt 8.1 dient zur Begriffsklärung und zur Problemerläuterung. Hier wird motiviert, wieso ein solches Verfahren wertvoll ist. Abschnitt 8.2 präsentiert den aktuellen Stand der Technik und Wissenschaft zur Eruierung von Hardwarerahmenbedingungen. Abschnitt 8.3 zeigt die Methode zur Ermittlung von Mindestanforderungen mittels der Experimentierumgebung. Mittels des Szenarios 4 und dem Szenario 5 wird der Prozess detailiert exemplifiziert.

8.1 Problembeschreibung

zentrales Thema Kompatibilität

Software auf unterschiedlichen Zielsystemen auszuführen birgt das enorme Vorteile [Bro03]. Kompatibilität bzw. Portabilität von Software und Systemen ist deshalb ein zentrales Thema der Software- und Systementwicklung.

❑ **Besseres Kosten-Nutzen-Verhältnis:**

Wenn Software auf möglichst vielen Zielsystemen lauffähig ist, ergibt sich ein höheres Kosten-Nutzen-Verhältnis bei der Erstellung von Software, u.a. weil Software, die bestimmte Funktionen bereitstellt, wiederverwendet werden kann [Lim94; FT96]. Man verwendet dafür qualitativ hochwertigen Code aus (evtl. von extern zugekauften) Bibliotheken wieder.

❑ **Sicherere und effizientere Entwicklung:**

Der Ansatz, dass Software nicht genau passend für das Zielsystem konstruiert werden muss, kam mit der Etablierung der höheren Programmiersprachen auf. Anwendungen werden erstellt, indem mit einem passenden Übersetzer (Compiler) von der Hochsprache in die Maschinensprache des Zielsystems übersetzt wird [JR78]. Zuvor wurden Anwendungen direkt in der Maschinensprache für das Zielsystem implementiert. Da Maschinensprachen im Gegensatz zu Hochsprachen für den Menschen schwerer zu verstehen sind, war dies durch einen höheren Aufwand und höhere Fehlergefahr gekennzeichnet. Insbesondere konnte Code zumeist nicht wiederverwendet werden.

Wegen einer Kostenreduktion und für eine mögliche Wiederverwendung soll Software nicht genau auf ein Zielsystem mit seinen genauen technischen Details konstruiert werden. Man geht bewusst eine Abstraktion ein, um schneller, effizienter und fehlerfreier ein Softwareprodukt zu entwickeln.

❑ **Erfolg kompatibler Systeme:**

Die Dominanz der IBM-PC kompatiblen Systeme und damit der x86 Familie von Intel (Intel hat einen Marktanteil von 80% bei PC-Mikroprozessoren [Fri07]) ist sogar nur durch die Abwärtskompatibilität zu erklären. So sind die Intel-Prozessoren der x86er Familie in den heutigen PCs abwärtskompatibel mit dem 1978 entwickelten 8086 Prozessor [Maz10]. Für die Kunden war sichergestellt, dass teure oder spezielle Software auch auf einer moderneren Version des Rechners lauffähig ist.

Insbesondere unterscheiden sich die heutigen PC-Architekturen, Software soll aber auf allen Systemen mit der gleichen Qualität ausführbar sein.

Technologiediversität

Einerseits ist also Kompatibilität eine nicht-funktionale Anforderung die gewünscht und notwendig ist, andererseits zeigt die Hardware durch den Technologiefortschritt eine viel höhere Diversität auf. Beispielsweise können unterschiedliche Controller mit spezifischen Leistungsdaten verwendet werden oder es gibt die unterschiedlichsten Speichermedien: von sehr schnellem und sehr teurerem Speicher bis hin zu extrem günstigem, langsamem Speicher.

Standard-Schnittstellen

Die heute verwandten Hardwarekomponenten haben zumeist Standard-Schnittstellen (standardisierte oder de-facto Standards) die funktional vollständig identisch sind [Rot84]. Ein Softwareentwickler weiß, unter anderem durch spezielle Entwurfs- und Architekturmuster wie Schichten oder Proxies [Gam+95] insbesondere des Betriebssystems, zu meist nicht, auf welche genaue Ausprägung einer Hardwarekomponente die Software zugreift. So kann ein Datenspeicher beliebige Leistungsdaten haben, beispielsweise unterscheiden sich eine Hochleistungsfestplatte eines Servers oder eines USB-Sticks enorm in ihren Zugriffsgeschwindigkeiten.

Dies führt zu dem Problem, dass eine vollständige Testabdeckung nur schwer möglich ist.

Performanz Systemkomponenten

Es wird also nicht speziell auf ein bestimmtes Zielsystem entwickelt (wegen einer Wiederverwendung, effiziente Entwicklung, Kostenreduktion, Notwendigkeit kompatibler Systeme, etc.), andererseits ist die Diversität der Hardware enorm gewachsen. Dies impliziert folgendes Problem: eine Zielplattform auf die entwickelt werden soll bzw. auf der das System ausgeführt werden soll, ist zwar technisch kompatibel (gleiche funktionale Schnittstellen) jedoch ist die Performanz der Systemkomponenten unter Umständen nicht ausreichend, was im Extremfall zu einem nicht gebrauchsfähigen Softwareprodukt führen kann.

Beispiel 8.1.1 Mindestvoraussetzungen von Standardsoftware
Mindestvoraussetzungen

Beispielsweise sind beim Kauf von Standardsoftware für den PC Mindestvoraussetzungen an das System angegeben, um die Software ausführen zu können. Diese werden durch Tests der Software auf verschiedenen Systemen ermittelt. Durch die steigende Diversität der Systeme können diese Mindestvoraussetzungen nicht mehr bzw. nur ungenügend empirisch ermittelt werden.

8.1.1 Rahmenbedingung für Hardware

Rahmenbedingungen

Es sind unter anderem keine Aussagen über bestimmte Rahmenbedingungen an die Hardware möglich. Wünschenswert wären Aussagen der Art: Hardwarekomponente x benötigt eine Performanz im Intervall von $x.1$ und $x.2$ Millisekunden damit das Softwareprodukt auf dem Zielsystem lauffähig ist, oder, wenn Hardwarekomponente y eine Performanz im Bereich von $y.1$ und $y.2$ Millisekunden hat, muss die Hardwarekomponente z mindestens eine Performanz kleiner als $z.1$ haben.

Kosteneinsparungen

Damit sind einerseits Aussagen über die benötigte Zielplattform möglich, andererseits können in der Produktion der entsprechenden Hardware (beispielsweise im Embedded-Bereich) Kosten eingespart werden, in dem eine performantere, aber teurere Hardwarekomponente durch ein eventuell günstigere Komponente ausgetauscht wird.

8.2 Status Quo - Eruierung von Hardwarerahmenbedingungen

8.2.1 Modellierungen

Modellierung

Auf Grund von (mathematischen) Modellen (Markov-Ketten[1], gewichtete Petri-Netze[2] und vieler mehr) kann eine Vorhersage über die Hardwareanforderungen eines Softwaresystems getroffen werden, damit die Anforderungen (Usability, Reaktion des Systems, ..) erfüllt werden können.

Nachteile

Diese Modelle können nicht immer erstellt werden und dieser Ansatz hat einige inhärente Nachteile[3], er ist mit großem Aufwand verbunden und deshalb teuer. Er wird jedoch vor allem bei kritischer Software verwandt, da er eine a priori Sicherheit bietet.

8.2.2 Skalierung der Hardwareleistung

Kill it with iron

Besteht ein Produkt aus Hard- und Software wird manchmal für den Anwendungsfall übertrieben leistungsfähige, und damit zumeist teurere, Hardware verwendet, die Hardware wird hochskaliert (vgl. den KIWI-Approach[4]). Dies kann sehr sinnvoll sein bei Einzellösungen, da hiermit im Vergleich zur Gesamtproduktion die teueren Testprozesse reduziert werden können. Aber bei Produkten mit hohen Serienzahlen, beispielsweise in der Automatisierungsbranche, sind auch Minimalbeträge entscheidend, so dass hier genau die zum Preis-/Leistungs-/Anforderungsverhältnis passende Hardware eingesetzt werden muss.

erhöhter Energieverbrauch

Des Weiteren kann unverhältnismäßig leistungsfähige Hardware neben höheren Investitionen zu weiteren Kosten führen, beispielsweise durch erhöhte Energie im Betrieb und eine notwendige Kühlung.

Schätzungen

Der Normalfall ist aber zu meist folgender: Entwickler schätzen mittels ihres Expertenwissens den benötigten Hardwarebedarf ab, wobei als Sicherheit zu meist bessere als benötigte Hardware vorgeschlagen wird. Dennoch ist diese Schätzung alles andere als zuverlässig, insbesondere bei großen, komplexen und dadurch unüberschaubaren Systemen.

---

[1] siehe Abschnitt 3.2.8, Seite 45
[2] siehe Abschnitt 3.2.6, Seite 44
[3] siehe Abschnitt 3.5.1, Seite 51
[4] siehe Abschnitt 3.4.0.1, Seite 50

8.2.3 Testen unterschiedlicher Hardwarekonfigurationen

vollständige Testabdeckung unmöglich

Durch den großen Umfang funktional kompatibler Hardware können herkömmliche Testtechniken nur ausgewählte Konfigurationen und Umgebungen validieren. Ein vollständiger Test aller Konfigurationen für bestehende Hardware ist durch die enorme Vielfalt der möglichen Hardwarekomponenten nahezu unmöglich.

dynamischer Hardwaremarkt

Neue Hardware mit denselben Standard-Schnittstellen aber unterschiedlichen Leistungskenngrößen dringt kontinuierlich in den höchst dynamischen Hardwaremarkt. Dies impliziert, dass neuartige, aber abwärtskompatible Hardware notwendigerweise ungetestet bleibt.

Beispiel 8.2.1 Verdrängung von Magnetplatten durch Flash Speicher
Es wird davon ausgegangen, dass Flash-Speicher die herkömmlichen Magnetplatten bei Festplatten ersetzen werden [GF08].

Flash-Speicher zeigt aber im Gegensatz zu Magnetplatten eine enorme Zeitdifferenz zwischen schreibenden und lesenden Zugriffen [Lee+08], was zu Problemen bei dem Softwareprodukt führen kann aber weitgehend ungetestet bleibt.

hoher Testaufwand für kompatible Hardware

Der technische Aufwand für die Tests mit unterschiedlicher Hardware ist enorm hoch, Tests für zukünftige, abwärtskompatible Hardware sind nicht durchführbar. Deswegen bleiben Tests mit kompatibler Hardware weitgehend ungetestet, was aber zu Problemen und langwieriger Fehlersuche führen kann. So ist es denkbar, dass auf der Entwicklungsplattform das System fehlerfrei getestet wurde, jedoch bedingt durch unterschiedliche Performanz der Hardwarekomponenten des Systems funktionale Fehler auftreten können. Möglich wäre beispielsweise, dass auf irgendeinem Zielsystem der Speicher zu langsam ist und plötzlich unerwartete Synchronisationsfehler auftreten.

Anforderung an die Hardware

Der momentane Stand der Technik ist, dass zwar funktionale Anforderungen des Systems getestet werden, Anforderungen an die Hardware jedoch nicht eruiert werden können. Es wird nicht mit ausreichendem Unfang an Hardwareumgebungen getestet, z. B. nur auf wenigen exemplarischen Zielsystemen. Es gibt keine Testmethodologie, die umfangreiche Tests an den unterschiedlichsten Zielsystemen ermöglicht bzw. die Aussagen über notwendige Leistungsgrößen der Hardware zulässt.

Reduktion auf Leistungsbereiche

Das Problem lässt sich aber reduzieren: eigentlich müsste das System nur für bestimmte Leistungsbereiche der Hardware getestet und Aussagen der Reaktion des Systems gewonnen werden. Dies war aber bislang nicht möglich, denn Hardware hat bestimmte, unveränderliche Leistungskenngrößen.

Beispiel 8.2.2 Bestimmung der Hardwareconstraints bei Szenario 4 und Szenario 5
In der Entwicklung oder im Test kann nur der Test mit diversen Speichern durchgeführt werden. Dies hat aber folgende Nachteile:

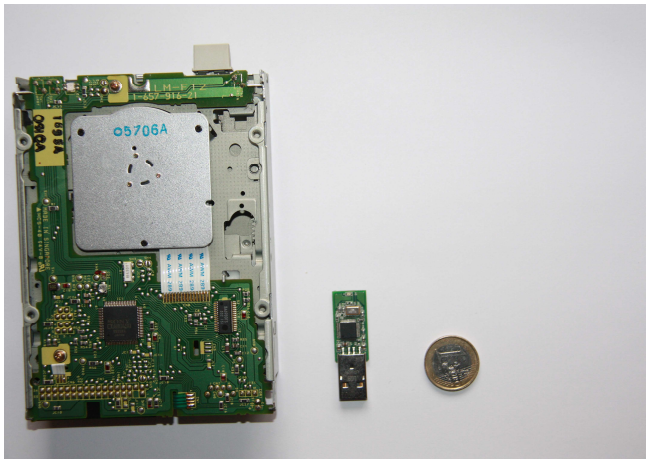


Abbildung 8.1: Magnetfestplatte im Vergleich zu einem USB-Stick mit viel größerer Kapazität.

- **Nicht automatisierbare Tests:**
  Der Test kann nicht automatisiert werden, die Speicher müssen manuell ausgetauscht werden, das Testumfeld muss immer wieder hergestellt werden. Es liegt also ein extrem hoher Aufwand vor.
- **Hoher organisatorischer Aufwand:**
  Die unterschiedlichen Festplatten müssen real vorliegen. Dies stellt einen hohen organisatorischen Aufwand dar und impliziert einen nicht unbedeuteten Kostenfaktor.
- **Keine Aussagen über Leistungskenngrößen:**
  Es kann nur über genau einen Speicher getestet werden. So können keine Aussagen über entsprechende Intervalle und Mindestmerkmale bei den Leistungskenngrößen gemacht werden.

8.3 Analyse von Hardwarerahmenbedingungen

Performanzbeeinflussung

Bei der Analyse der Eigenschaften des Gesamtsystems durch die von der Hardware gegebenen Leistungskenngrößen der Hardwarekomponenten (Performanz, Schreibgeschwindigkeit, Lesegeschwindigkeit, etc.) können die Auswirkungen dieser Gegebenheiten auf die Eigenschaften eines Systems getestet werden. Damit sind Prognosen über die funktionalen und nicht-funktionalen Eigenschaften eines Gesamtsystems in Abhängigkeit von der verwandten Hardware möglich.

Variation der Performanz

Während Ausführung des Systems auf dem virtuellen Host wird die Leistungskenngröße einer (virtuellen) Hardwarekomponente variiert. Dies geschieht auf folgende Art und Weise: Der Testentwickler kann an der entsprechenden Schnittstelle der virtualisier-

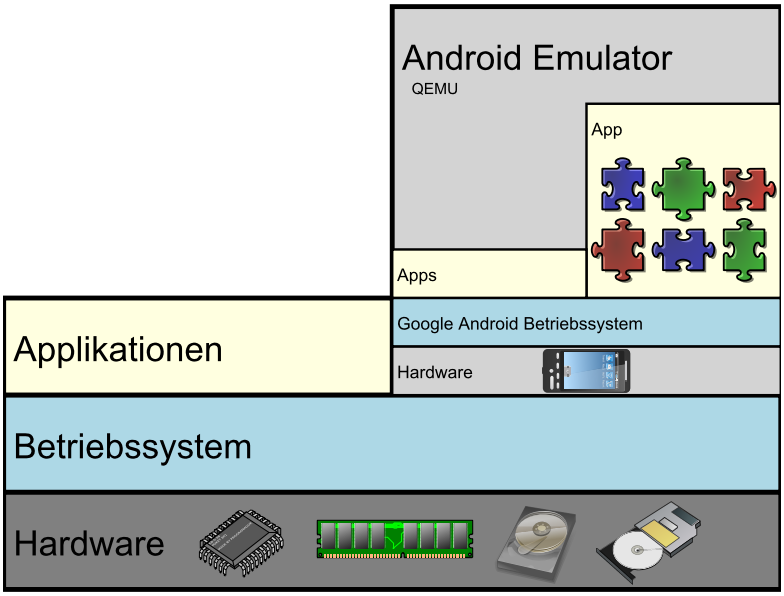


Abbildung 8.2: Der Android Emulator.
❑ Ein auf QEMU basierender Emulator für mobile Geräte mit dem Android Betriebssystem.
❑ Er wird insbesondere gerne von Webseitenentwicklern genutzt, um eigene Entwicklungen für *Android* basierte Systeme zu testen.

ten Komponente einen Algorithmus oder eine zeitliche Beschreibungen anlegen, die die zu simulierende Leistungsverminderung beschreibt. Die virtualisierte Hardwarekomponente verhält sich dann in ihrer Performanz analog zu dieser Beschreibung, d.h. das System wird mit einer Hardwarekomponente mit der beschriebenen Leistung getestet. Die virtuelle Hardwarekomponente muss dazu um die entsprechende Eigenschaften und Leistungsminderung erweitert werden.

Beispiel 8.3.1 Schreib- und Lesegeschwindigkeit bei virtuellen Speicher
Beispielsweise sollten die in den Szenario 4 und Szenario 5 beschriebenen (virtuell replizierten) Speicher um eine Möglichkeit zur Änderung der Schreib- und der Lesegeschwindigkeit erweitert werden. Dies wurde in Abschnitt 7.3.4 mit QEMU^{dt} gezeigt.

Datenerhebung

Die einzelnen Aktionen des Systems werden gemessen und die gemessenen Daten stehen nach der Ausführung zu einer Analyse bereit. Auf Basis dieser Werte können Rückschlüsse auf die funktionalen und nicht-funktionalen Eigenschaften eines Systems unter diversen Hardwarekonstellationen mit ihren individuellen Leistungsbereichen gezogen werden.

Tests auf Hardwarerahmenbedingungen

Mit diesem Ansatz sind Aussagen über notwendige Bedingungen realisierbar, wie zum Beispiel: benötigt der Zugriff auf den Datenspeicher um dieses Zeitintervall länger, kann die Sicherheit des Gesamtsystems nicht gewährleistet werden, oder, verwenden wir eine Netzwerkkarte eines bestimmten Typs muss ein Austausch bei der Festplatte vorgenommen werden. So gibt diese Analyse von unterschiedlichen Leistungskenngrößen Hinweise auf diverse mögliche Einsatzszenarios. Im Allgemeinen ist, wie bereits beschrieben, durch die hohe Anzahl von möglichen Hardwarekomponenten ein solcher Test, durch die hohe Bandbreite nicht-funktionalen Eigenschaften, nicht durchführbar bzw. sehr kosten- und zeitintensiv.

8.3.1 Vorteile des Ansatzes

Folgende Vorteile ergeben sich durch diesen Ansatz:

- **Tests über Leistungskenngrößen:**
  Die Leistung einzelner Hardwarekomponenten kann zielgerichtet verändert werden, ohne die Komponente selbst zu ändern (z. B. durch Austauschen). Damit bleiben die funktionalen Eigenschaften des Systems im Normalfall (ob dies so ist, soll ja gerade getestet werden) erhalten, Tests werden über Leistungskenngrößen der Komponenten möglich.

- **Dynamisches Laufzeitverhalten:**
  Des Weiteren werden die Daten aus dem dynamischen Laufzeitverhalten des Systems gewonnen, was eine getreuere Analyse als andere Analysemethoden zulässt

(statische Analyse, Abschätzen des Hardwarebedarfs von Entwicklern etc.) die bestimmte Eigenschaften des Systems (beispielsweise parallele Verwendung einer Hardwarekomponente) nicht in die Untersuchung integrieren können.

Beispiel 8.3.2 Ermitteln der Rahmenbedingungen an Szenario 4 und Szenario 5
Beide Szenarios lassen sich mit den beschriebenen bisherigen Lösungsmöglichkeiten nicht oder nur mit extrem hohen Aufwand validieren.

- **Szenario 4**
  Der Speicher kann in normalen Testumgebungen nicht explizit langsamer oder schneller gemacht werden, diese Werte sind durch die Hardware bestimmt und aktuell ausgeführte Prozesse beeinflussen diese. Die Speicher müssten also explizit manuell ausgetauscht werden.
- **Szenario 5**
  Die Zugriffsgeschwindigkeit beim Speicher ist in einer nicht virtualisierten Umgebung durch die physikalische Realisation des Speichers bestimmt. Aussagen über bestimmte Grenzwerte wären jedoch essentiell, beispielsweise ob die unterschiedlichen Lese- und Schreibzugriffszeiten bei der geplanten Flash-Disk ein Problem darstellen könnten.

8.3.2 Testausführung von Szenario 4 – Mindestanforderungen an den Speicher

Mindestanforderung an Hardwarekomponenten

Ein System, bestehend aus Software und Hardware, ist daraufhin zu testen, welche Mindestanforderung die Hardwarekomponenten, hier die Festplatte, erfüllen muss (siehe Szenario 5, Seite 28). Zwar könnte ein Testingenieur Modifikationen mittels Software durchführen, z. B. zusätzliche Prozesse eingefügen, die gleichzeitig auf die Hardwarekomponente zugreifen und so künstlich die Leistungskenngrößen verändern. Dies ist aber mit hohem Aufwand verbunden und die Nutzung kann nicht feingranular angepasst werden. So ist es nicht möglich, bestimmte Grenzen für die benötigten Hardwarekomponenten zu eruieren. Es ist keine isolierte Beeinflussung in dedizierter Art und Weise möglich. Dieser Analyseprozess zum Auffinden der Mindestanforderungen kann nicht in einer systematischen Art und Weise erfolgen.

Testcase

Mit der vorgeschlagenen Lösung wird dies jedoch möglich. Ein Testfall (Testcase) ist nun ein Softwaretest der mittels Software implementiert wurde. Am entsprechenden Interface, also der virtuellen Implementierung der Hardwarekomponenten, können nun während der Testausführung bestimmte Leistungskenngrößen verändert werden.

Usability und funktionale Softwaretests

Usability Tests [Holo5; DR93] und (funktionale) Software Tests [YPo5] werden entsprechend durchgeführt, wobei hier das Ergebnis abhängig von den (emulierten) Leistungskenngrößen sein kann und wird.[5]

[5] siehe Exkurs 2.1.5, Seite 20

virtuell variierbar Bei realer Hardware, also ohne eingefügte Virtualisierungslösung, kann dies nicht getestet werden. Die reale Hardware hat bestimmte, durch physikalische Gegebenheiten bestimmte Leistungsmerkmale, die aber in einer virtualisierten Lösung variiert werden können.

mathematische Analyse Zwar wäre es möglich, mittels mathematischen Modellen die Auswirkung nicht-funktionaler Eigenschaften zu analysieren und Grenzen zu deduzieren, jedoch ist dies mit großen Kosten und Aufwand verbunden, so dass dieses in der Praxis kaum eingesetzt wird.[6]

Android Die Szenarios werden am Andorid Betriebssystem dargestellt, es ist auf Seite 27 definiert worden. Gerade dieses aktuelle freie Betriebssystem, welches auf diversen Hardwareplattformen und Konfigurationen ausgeführt werden soll, ist prädestiniert als Darstellungsbeispiel. Der Source Code des *Android Emulator*s des Android Projektes [And10b] kann mit folgend Schritten geladen werden:

8.3.2.1 Installation von Repo:

Repo *Repo* ist ein Tool welches auf dem Open-Source Versionskontrollsystem *Git* aufsetzt. Es kann wie folgt installiert werden [And10c]:

```
curl http://android.git.kernel.org/repo > ~/bin/repo
chmod a+x ~/bin/repo
mkdir working-directory-name
cd working-directory-name
repo init-u git://android.git.kernel.org/platform/manifest.git
```

Listing 8.1: Shell: Installation des Tools Repo

Mittels eines repo sync werden nun alle Files in das momentane Arbeitsverzeichnis geladen.

8.3.2.2 Realisation der Experimentierumgebung

Experimentierumgebung Analog zu Abschnitt 7.3.4 werden die gleichen Änderungen an QEMU durchgeführt. Vom Android Emulator kann nicht auf QEMU^{dt} zugegriffen werden, mittels des Schalters `-qemu` können jedoch Argumente an QEMU^{dt} weitergegeben werden.

8.3.2.3 Erstellen und Starten des Android Emulators

make Ein make erstellt den Android Emulator, was je nach System ziemlich lange dauern kann. Eventuell müssen dafür benötigte Pakete nachgeladen werden, ansonsten endet make mit einem Fehler.

[6] siehe Abschnitt 3.5.1, Seite 51

```
sudo apt-get install git-core gnupg sun-java5-jdk flex bison gperf
    libsdl-dev libesdo-dev libwxgtk2.6-dev build-essential zip curl
    libncurses5-dev zlib1g-dev
```

Listing 8.2: Shell: Installation benötigter Pakete

Start

Mit folgenden Befehlen kann der Android Emulator mit einer Standardkonfiguration gestartet werden.

```
out/host/linux-x86/bin/emulator -system out/target/product/generic/ -
    kernel prebuilt/android-arm/kernel/kernel-qemu
```

Listing 8.3: Shell: Start des Android Emulators

Abbildung 8.3.2.3 zeigt das sich Fenster mit dem Android Emulator.

Abbildung 8.3: Android Emulator.

❑ Mittels Ctrl + F11 (STRG+F11) kann die Orientierung des Fensters geändert werden.
❑ Mittels F2 (F2) gelangt man in das (Android-)Menü.
❑ Weitere Informationen zur Bedienung des umfangreichen Emulators sind auf [And10a] verfügbar.

8.3.2.4 Installation einer Benchmarkapp

Installation der Benchmarkapp

Von Entwicklern des *softweg-studio*s wurde 2009 das Tool *Benchmark v1.03 Application for Android* entwickelt. Dieses kann beispielsweise von `http://www.androidapk.net/` bezogen werden. Mittels des Befehls abd kann dieses Benchmarktool in dem virtualisierten Android installiert werden.

```
cd .../build/tools/
wget http://www.androidapk.net/softweg.hw.performance.apk
cd ../..
out/host/linux-x86/bin/adb install /home/flo/mydroid/flo/build/
    tools/softweg.hw.performance.apk
```

Abbildung 8.4: Android Emulator mit der installierten Benchmark Applikation.

Nach der Installation ist die Benchmarkapplikation in Android ausführbar (siehe Abbildung 8.3.2.4).
In der Abbildung wird die Benchmark Applikation ausgeführt und ermittelt die Leistungskenngrößen in der emulierten Umgebung.

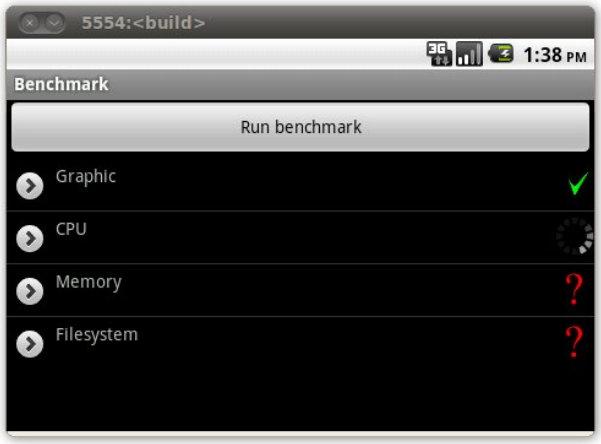


Abbildung 8.5: Android Emulator mit Benchmark Applikation.
In dem Screenshot wird die Benchmark Applikation ausgeführt und ermittelt die Leistungskenngrößen in der emulierten Umgebung.

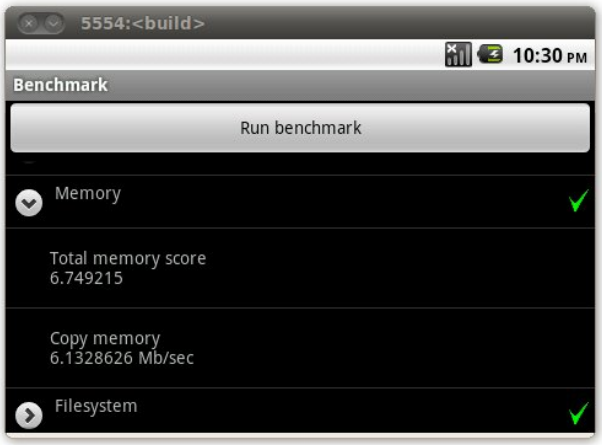


Abbildung 8.6: Android Emulator mit Benchmark Applikation.
Im Screenshot wird der Emulator ohne zeitliche Variation der Hardwarekomponente Speicher ausgeführt.

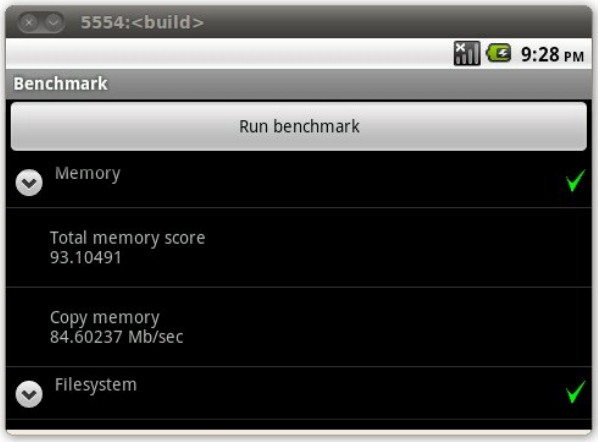


Abbildung 8.7: Android Emulator mit Benchmark Applikation.
Der Emulator wird im Screenshot mit einer zeitlichen Variation ausgeführt.

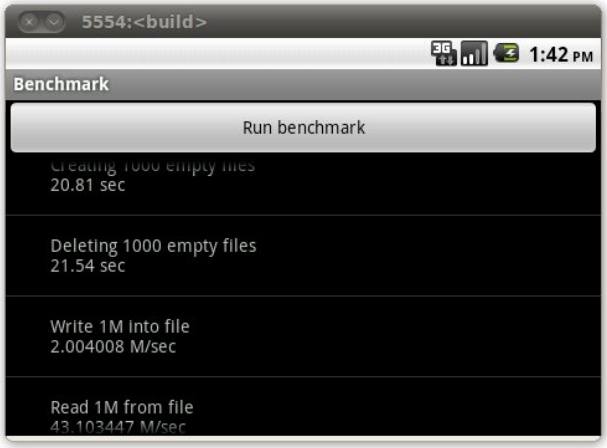


Abbildung 8.8: Testausführung der Benchmark Applikation auf einem Ubuntu 10.04 LTS **ohne** Prolongation einzelner Hardwarekomponenten.

8.3.3 Testdurchführung von Szenario 5 – Unterschiedliche Schreib- und Lesegeschwindigkeit

Speicherzugriffsgeschwindigkeit

In ähnlicher Weise wie bei Szenario 4 kann die Speicherzugriffsgeschwindigkeit variiert werden. Hier wird jedoch nur die Schreibgeschwindigkeit sukzessive reduziert und die Reaktionen des Systems gemessen. Auf diese Weise kann eruiert werden, ob eine Flash-Disk grundsätzlich einsetzbar ist und welche Mindestanforderungen bei den Schreibgeschwindigkeiten bei der Flash-Disk benötigt werden.

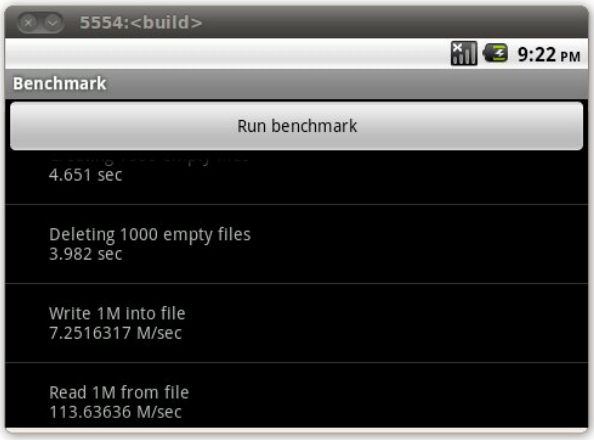


Abbildung 8.9: Android Emulator mit Benchmark Applikation.

❑ Vor diesem Lauf der Benchmark Applikation wurde eine zeitliche Variation durchgeführt.

❑ Spezifische Anwendungen, die viel auf dem Lesen von Dateien basieren, werden ein schlechtes Performanzverhalten in der Emulation zeigen.

Hardwareconstraints

Dadurch können beispielsweise Rahmenbedingungen (Constraints) der Hardware bestimmt werden, auf welchen Systemen die Anwendung ausgeführt werden kann, wo die Grenzen der Hardware liegen.

8.4 Zusammenfassung und Beantwortung von Teilfragestellung θ

Dieses Kapitel erweitert den Test von Systemen um Hardware mit variablen Performanz-Eigenschaften, was nur durch die Virtualisierung dediziert, effizient und strukturiert ermöglicht wird.

Abhängigkeiten bzgl. der Hardwarekenngrößen

Somit können gesamte Systeme (Softwareprodukte inkl. darunterliegender Betriebssysteme) systematisch auf Abhängigkeiten von Performanz-Eigenschaften der Hardware getestet werden. Einerseits muss weder das Betriebssystem noch das zu testende Softwareprodukt geändert werden, andererseits muss physikalisch die Testhardware nicht verändert werden.

Testreihen für Randbedingungen

Es sind Testreihen möglich, mit derer auf einfache Art Randbedingungen der verwendeten Hardware hinsichtlich ihrer Leistungsfähigkeit ausgetestet werden können. Fragestellungen wie: genügt CPU X oder muss das Modell Y verwendet werden können nun effektiv und schnell beantwortet werden. Insgesamt ergeben sich erhebliche Zeit- und Kostenersparnisse sowie höhere Zuverlässigkeit in der Systementwicklung.

methodischer Prozess

So kann in einem methodischen Prozess die Performanz einer Hardwarekomponente, beispielsweise eines nicht-flüchtigen Speichers, sukzessive verändert werden. Die Reaktion der Software kann gemessen werden, somit sind die gewünschten und erforderlichen Rückschlüsse möglich, beispielsweise welches Intervall an Leistungskenngrößen die Hardwarekomponenten vorweisen müssen, damit die Software die gewünschte Reaktion zeigt bzw. die Anforderungen erfüllt. Das Gesamtsystem kann also in Abhängigkeit von den durch die Hardware gegebenen Leistungskenngrößen gezielt untersucht werden.

Szenarios

Mit zwei Szenarios des sich etablierenden Android Betriebssystems bzw. der Software-Plattform für mobile Geräte wurde dieser Prozess in diesem Kapitel illustriert.

- ❑ **Szenario 4 – Speicher mit unterschiedlichen Leistungskenngrößen**
  Die Lese- und Schreibgeschwindigkeit des nichtflüchtigen Speichers wurden in der Emulation variiert. Durch ein Benchmarking und Test (Usability Test und funktionaler Softwaretest) kann auf die Mindestleistungskenngrößen des Speichers, je nach IT-Anwendung, geschlossen werden.
- ❑ **Szenario5 – asymmetrische Leistungskenngrößen bei einer Flashdisk**
  Flashspeicher wird herkömmlichen Speicher ersetzen, zeigt aber asymmetrische Lese- und Schreibgeschwindigkeit. Viele bestehende IT-Anwendungen sind auf dieses Verhalten ungetestet. In Szenario 5 wurden unterschiedliche Leistungskenngrößen in einem emulierten Android Softwareplattform getestet.

Dass ein solcher Testprozess nicht trivial ist und dringend benötigt wird, sieht man an den aktuellen und akuten Performanzproblemen von *iOS 4* auf dem *iPhone 3G*. Im Gegensatz zu *Apple* baut jedoch *Google* mit dem in den Szenarios verwendeten *Android* auf einer hohen Hardwareheterogenität auf. Hier kann der Einfluss der Hardware auf die Systemperformance noch weniger abgeschätzt werden.

iOS 4 / iPhone 3G

Somit kann **Teilfragestellung** θ positiv beantwortet werden:
Die hier entwickelte Technik eignet sich in einer Kombination mit der Virtualisierungstechnologie zur Bestimmung der Mindestanforderungen der Hardwareleistungskenngrößen eines Systems.

Teilfragestellung θ ✔

Kapitel 9

Analyse

„Wieder und wieder bitte ich: Non multa sed multum - weniger Zahlen, aber gescheitere.“

Lenin (Wladímir Iljitsch Uljánow)

Übersicht des Kapitels

Abschnitt 9.1 beschreibt den Stand der Technik und Wissenschaft zum Data Mining und statistischen Analysen in der Softwareentwicklung.

Abschnitt 9.2 kategorisiert kurz die Analysemethoden in strukturentdeckende und modellbildende Analysemethoden sowie Simulationsansätze.

In Abschnitt 9.3 werden die strukturentdeckenden Analysemethoden eingeführt, hier wird die nötige Basis für statistische, multivariate Analysemethode gelegt und Algorithmen aus dem *Data Mining* kurz präsentiert.

Abschnitt 9.4 führt die im Rahmen der Performanzanalyse mit dem Analyseinstrumentarium entwickelten modellbildenden Methoden an.

Abschließend wird in Abschnitt 9.5 bis Abschnitt 9.12 die Performanzanalyse und an den in Abschnitt 2.2 eingeführten Szenarios demonstriert.

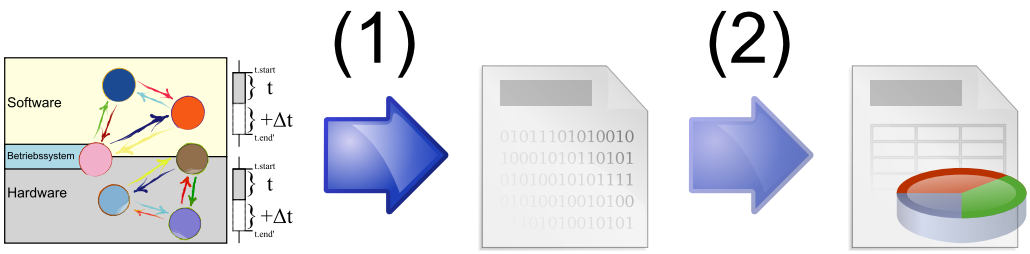


Abbildung 9.1: Auswertung des Experimentes.
❑ Dieses Kapitel beschreibt den Prozess, der mit dem letzten blauen Pfeil symbolisiert und mit (2) beschriftet ist.
❑ In diesem Kapitel wird gezeigt, wie die gewonnen Daten (siehe Kapitel 7) von Punkt (1) systematisch analysiert werden können, um Informationen über Zusammenhänge und Optimierungspotenzial im System zu gewinnen.

9.1 *Data Mining und statistische Analysen in der Softwareentwicklung*

Praxisrelevanz

In dieser Arbeit wird das Analyseinstrumentarium (die Prolongation, die Retardation sowie das simulierte Optimieren) für eine Performanzanalyse mit nachfolgender Performanzoptimierung eingeführt. Dieses Kapitel schließt nun die Lücke zwischen der notwendigen theoretischen Betrachtung und den praxisrelevanten Anforderungen dieser Arbeit.

Auswertung der Daten

Es wird gezeigt, wie die mit der Analyseinstrumentarium gefundenen Daten ausgewertet werden können, um Optimierungspotenzial bei IT-Systemen zu analysieren. Hierzu werden Anleihen aus dem *Knowledge Discovery in Databases* bzw. dem *Data Mining*, mit ihren elaborierten Algorithmen und Vorgehensweisen, gemacht.
Fayyad, Piatetsky-Shapiro und Smyth [FPSS96] definieren den Begriff des Knowledge Discovery in Databases als:

32 Knowledge Discovery in Databases

Definition KDD

Knowledge Discovery in Databases is the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data.

Die verwendeten Methoden werden in Abschnitt 9.2 klassifiziert und kurz präsentiert.

Datenbanktechnologie

Data Mining bzw. Knowledge Discovery in Databases wird, wie es der Name es schon sagt, in der Datenbanktechnologie verwendet um aus großen Datenbeständen wertvolle Informationen zu gewinnen. Es gibt jedoch Vorschläge Methoden aus dem Data Mining im Softwareentwicklungsprozess einzusetzen.

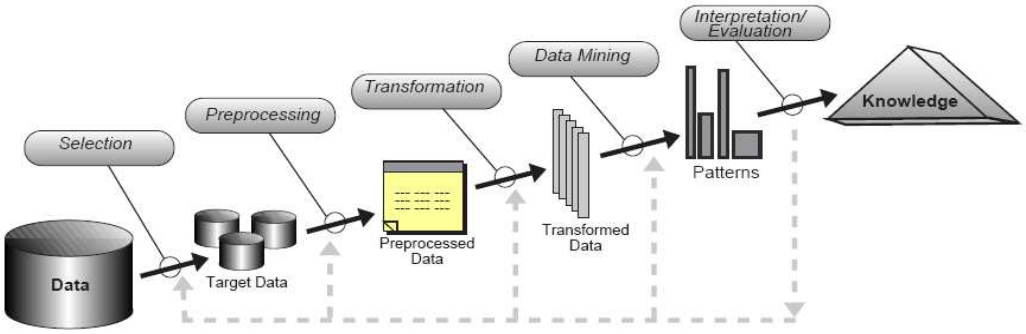


Abbildung 9.2: Prozessmodell im *KDD* nach Fayyad, Piatetsky-Shapiro und Smyth [FPSS96]. Zitiert von Fayyad, Piatetsky-Shapiro und Smyth [FPSS96]. Dieses Prozessmodell steht hinter dem letzten blauen Pfeil in Abbildung 9.1.

schlecht geschriebener Code

Fokaefs u. a. [Fok+09] benutzen ein anhäufendes Clusteringverfahren (*agglomerative clustering*), basierend auf dem *Jaccard-Koeffizienten*, um automatisch schlecht entworfenen Code zu refaktorisieren.

Softwareplagiate

Liu u. a. [Liu+06] entwickelten ein Tool namens *GPlag* welches Softwareplagiate entdecken soll. Dazu erstellen sie einen *program dependence graph* (PDGs), einen Graphen basierend auf Daten- und Kontrollflußabhängigkeiten. Dieser Graph ist weitgehend invariant gegenüber klassischen Methoden zur Verschleierung von Plagiaten.

multivariate Performanzanalyse

Santiago, Rover und Rodríguez [SRR02] nutzten eine multivariate Analysemethode zur Performanzanalyse. Diese variieren jedoch die Eingabewerte und nutzten nicht das hier in dieser Arbeit eingeführte Analyseinstrumentarium mit einer Variiation zeitlicher Eigenschaften.

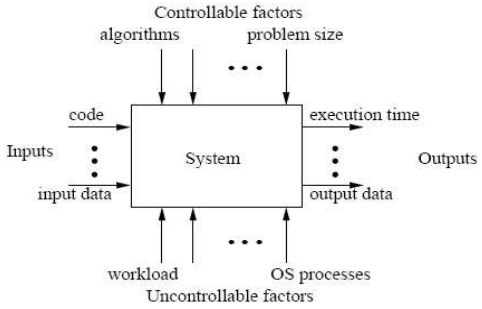


Abbildung 9.3: Der Ansatz zum Experiment nach Santiago, Rover und Rodríguez [SRR02]. Eine Multivariate Performanzanalyse ohne das Analyseinstrumentarium und (vollständige) Kontrolle über das System (z. B. *workload*, *OS-processes*). Variiert werden die Eingabewerte (*Inputs*) und die kontrollierbaren Faktoren (*algorithms*, *problem size*). Quelle: [SRR02].

9.2 *Klassifizierung der Analysemethoden*

Analyseparadigmen

Im Rahmen dieser Arbeit wurde das Analyseinstrumentarium mit unterschiedlichen Perspektiven bzw. Paradigmen zur Analyse von IT-Systemen verwandt.

strukturentdeckend

❑ **Strukturentdeckende Analysemethoden** nutzen den Wirkzusammenhang des Analyseinstrumentariums um Zusammenhänge im System zu entdecken.
Ein Überblick gibt Abschnitt 9.2.1.

Simulationsansätze

❑ **Simulationsansätze** variieren die Performanz einzelner Komponenten (Hard- und Software) und simulieren so den Einfluß einer performanteren/langsameren Komponente auf das System.
Ein Überblick gibt Abschnitt 9.2.2.

modellbildend

❑ **Modellbildende Analysemethoden** nutzen den Wirkzusammenhang des Analyseinstrumentariums um ein Modell des Systems zu konstruieren.
Ein Überblick gibt Abschnitt 9.2.3.

9.2.1 *Strukturentdeckende Analysemethoden*

Zwischenschritt Analyseinstrumentarium

Das System als Ganzes wird untersucht, das Analyseinstrumentarium dient als Zwischenschritt zur Analyse. In einem Experiment werden **unterschiedliche** Komponenten des Systems prolongiert und nachfolgend die resultierende Performanz der betrachteten Komponenten im variierten System gemessen. Wegen des Wirkzusammenhangs können mit nachfolgenden mathematischen Methodensammlungen aus den Meßdaten auf Zusammenhänge und Strukturen des Systems geschlossen werden.
Für die weitere Erörterung müssen zwei Begriffe eingeführt werden:

- **Explorative Statistik**
  Die explorative (erkundende) Statistik wird angwandt um bisher unbekannte Strukturen und Zusammenhänge in den Daten zu ermitteln [Saco4; Pol88].
- **Multivariate Datenanalysen**
  Als multivariate Datenanalyse werden Methoden kategorisiert, die mehrdimensionale (multivariat) Variablen untersuchen [Bolo4; Bac+08].

Paradigma

❑ **Das Paradigma** dieser Analysemethoden von Systemen ist, dass künstlich eingefügte Varianz in den Performanzeigenschaften von Komponenten abhängige Komponenten (durch den Wirkzusammenhang) beinflussen. Diese Beeinflussungen (Varianzen) werden gemessen, die Zusammenhänge der Komponenten im System in den hochkomplexen Meßdaten können durch statistische Methoden entdeckt werden. Es werden durch geeignete multivariate Analysemethoden Zusammenhänge im System aufgedeckt, somit kann die Struktur des Systems erkannt werden.

Kernfrage

❑ **Die Kernfrage** dieser Analysemethoden ist, welche Strukturen, Gruppierungen, Klassen die Komponenten bezüglich ihrer Performanz aufweisen.

Vorhergehensweise

❑ **Die Vorgehensweise** ist ein dediziertes Prolongieren von Komponenten. Wird das System ausgeführt, weisen abhängige Komponenten durch den Wirkzusammenhang ebenfalls Änderungen in ihrer Performanz auf. Durch statistische, multivariate Methoden wird die künstliche Varianz in den Performanzeigenschaften des Systems ausgewertet. Somit können Zusammenhänge im System entdeckt werden.

9.2.2 *Simulationsansätze*

Auswirkung

Hier werden die Auswirkung einer Komponente auf das System untersucht. Es sollen keine Strukturen entdeckt werden, sondern die Abhängigkeit der Performanz des Systems von bestimmten Komponenten soll bestimmt werden.

Paradigma

❑ **Das Paradigma** dieses Ansatzes ist es, eine Komponente durch eine andere, performantere oder langsamere, Komponente auszutauschen. Bei einer Ausführung des Systems können die Auswirkungen direkt gemessen werden.

Kernfrage

❑ **Die Kernfrage** dieser Analysemethode ist ein „Was wäre wenn?“. Wie reagiert das System, wenn eine Komponente langsamer oder performanter wäre bzw. Aufwand in ein Optimieren einer (Software-)Komponente gesteckt wird?

Vorhergehensweise

❑ **Die Vorgehensweise** ist ein Austausch der Komponenten, beziehunsgweise ein simulieren (oder virtualisieren) von Komponenten (Hardwarekomponenten, Module, Softwareeinheiten, etc.) mit unterschiedlichen, insbesondere besseren oder optimierten, Performanzeigenschaften.

Leitidee: simulierte Komponenten

Zentral ist hier nicht der Gedanke von latenten Abhängigkeiten und gegenseitiger Beeinflussung von Performanz, die mittels dem Analyseinstrumentariums in einem System entdeckt werden, sondern direkt, was passieren würde, wenn eine Komponenten um einen Faktor X verlangsamt oder optimiert wird, so dass Auswirkungen eines evtl. Nachbesserns sichtbar werden.

Austausch von Komponenten

Dies kann durch die Prolongation und die Virtualisierungstechnik bzw. durch die Instrumentierung erfolgreich simuliert werden, ohne dass die Komponente selbst verändert werden muss. Die zu untersuchende Komponenten wird mit den gewünschten Performanzeigenschaften simuliert, das System ausgeführt und durch Messungen und Tests können direkt Rückschlüsse gezogen werden, wie eine solche Änderung das System beeinflusst.

Simulationsansätze wurden in Kapitel 8 genutzt. Hier wurde zielgerichtet die Hardware langsamer gemacht. Abschnitt 9.7 zeigt den umgekehrten Ansatz, einen Simulationsansatz, in dem Module schneller simuliert werden. Die Technik ist detailliert in Abschnitt 4.3.3 beschrieben. In Abschnitt 10.4 wird ein Simulationsansatz genutzt, um innerhalb reduzierter *wall clock time* einen *Aging related fault* von Szenario 8 – angeleht an einen Fehler der *Patriot Missile* – zielgerichtet zu reproduzieren.

9.2.3 *Modellbildende Methoden:*

Hier können durch geeignete Analysemethoden einfache Modelle des ausgeführten Systems erstellt werden.

❑ **Das Paradigma** dieser Analysemethoden ist, den Wirkzusammenhang zu nutzen um in einem Experiment ein einfaches Modell des Systems abzuleiten. Dieses Modell beschreibt quantitativ und qualitativ wie die Komponenten eines Systems zusammenhängen. Paradigma

❑ **Die Kernfrage** dieser Analysemethoden ist, nicht nur wie die Komponenten gruppiert werden können sondern auch wie die Performanzabhängigkeiten funktional zusammenhängen. Kernfrage

❑ **Die Vorgehensweise** ist ein dediziertes Prolongieren von Komponenten. Wird das System ausgeführt, weisen abhängige Komponenten durch den Wirkzusammenhang ebenfalls zeitliche Änderungen auf. Durch geeignete mathematisch Methoden wird die Zeitdifferenz in den Performanzeigenschaften des Systems ausgewertet. Somit kann ein einfaches Modell des Systems abgeleitet werden. Vorhergehensweise
Abschnitt 9.4 illustriert die modellbildende Methode an einer von Mangold [Man07] entwickelten Instanz, dem *Resource Dependence Graph*. Modelle

9.3 Strukturentdeckende Methoden

Optimierungskandidaten schwer identifizierbar

Bei unzureichender Performanz ist es bei den zu analysierenden Systemen nur schwer möglich, Optimierungskandidaten des Systems auf die Gesamtperformanz zu ermitteln. Dies hat unter anderem folgende Gründe:

- **Größe des Systems:**
  Die Performanz einzelner Module und Komponenten können zwar gemessen werden, durch die Anzahl (je nach analysierter Granularitätsstufe) ist es unmöglich zu eruieren, welche Elemente zusammenwirken.

- **Wiederverwendung von Softwareelementen:**
  Softwareelemente werden wiederverwendet, oft ist der Code durch Kapselung nicht für eine Analyse zugänglich.

- **Abhängigkeit der Softwareelemente von Hardwarekomponenten:**
  Software dient zur Steuerung von Hardwarekomponenten. Diese können gleichzeitig angesteuert werden, was zu einer Performanzminderung durch die gleichzeitig benutzte Ressource führen kann. Dies ist durch Profiling nicht zu erkennen, beispielsweise kann durch einen Call-Tree bei einem Profiling keine Performanzminderung durch ein gemeinsam benutztes gesperrtes Objekt erkannt werden.

automatisierbare Analyse

Eine Methode die die Prolongation als **Analyseinstrumentarium** nutzt, um Strukturen und Abhängigkeiten bezüglich der Performanz in einem System zu eruieren und diese zu gruppieren wird in diesem Abschnitt vorgestellt. Neben der Lösung der oben genannten Nachteile lässt sich diese Methode automatisieren und stellt deshalb eine enorme Erleichterung bzw. eine effiziente Methode zur Analyse von Systemen dar.

Statistik

Die **Analyse** (Gruppierung, Strukturierung) erfolgt mittels multivariater Verfahren aus der Statistik und Algorithmen aus dem *Data Mining* bzw. *Knowledge Discovery for Databases* (*KDD*). Somit wird es möglich, das Zusammenwirken einzelner Komponenten (Software und Hardware) in einem System zu verstehen.

9.3.1 Multivariate Datenanalyse

Die Analyse von Systemen mit der Prolongation als Analyseinstrumentarium und der Faktorenanalyse als multivariate statistische Methode wurde in [Mano7] eingeführt.
Die deskriptive oder beschreibende Statistik ist ein Zweig der Mathematik bzw. der Statistik, die folgende Ziele hat:

- **Übersichtliche Darstellung** von empirischen Daten durch Tabellen und Grafiken.

❑ **Unterliegende Kenngrößen** werden ermittelt.

Große Mengen von Daten können mit der deskreptiven Statistik aufbereitet werden [Bol04; BCK04; EKT08; Ass03].
In der (deskriptiven) Statistik wird zwischen univarate (seltener univariante), bivariate (seltener bivariante) und multivariate Verfahren unterschieden. Beispiel 9.3.1 verdeutlicht dies.

univariat, bivariat, multivariat

Exkurs 9.3.1 Univariate, bivariate und multivariate Analyse von IT-Systemen

❑ **Univariat**
Ein (instrumentiertes) System wird ausgeführt, die verschiedene Laufzeiten von Modulen werden protokolliert und aufbereitet (tabellarisiert). Insbesondere kann hier auch nur die Laufzeiten eines Softwaremoduls betrachtet werden. Um Informationen über die Durchschnittslaufzeit, Minimal- und Maximallaufzeit, Verteilungen etc. zu bekommen, werden die Daten nach einer Variablen, der protokollierten Laufzeit, analysiert.

❑ **Bivariat**
Untersuchungsgegenstand ist nun, ob die Laufzeit von zwei Komponenten, beispielsweise von Modul A und Modul B bzw. Modul C und die Festplattengeschwindigkeit, korrelieren, also zusammenhängen. Die protokollierten Daten des Systems müssen nun nach zwei Variablen analysiert werden, der Laufzeit von Modul A und Modul B bzw. die Laufzeit von Modul C in Abhängigkeit von der Festplattengeschwindigkeit. Da die Laufzeiten bei jedem Lauf gleich sein können, obwohl eine Abhängigkeit zwischen den zu analysierenden Variablen besteht, wird hier das Analyseinstrumentarium Prolongation verwendet um Abhängigkeiten in den Meßdaten sichtbar zu machen.

❑ **Multivariat**
Interessiert nun das Zusammenwirken der Komponenten in dem System, müssen mehr Variablenbeziehungen untersucht werden. Somit lassen sich Abhängigkeiten und das Zusammenwirken in den Laufzeiten der Komponenten ermitteln, was zu einem besseren Verständnis des Systems führt. Da das Zusammenwirken mehrerer Variablen untersucht wird, folgt unmittelbar, dass eine multivariaten Auswertung nicht aus mehrfach durchgeführten univariaten Analysen entstehen kann [Wer95, S. 149].

9.3.2 *Statistische Deskription*

Die zugrundeligende Idee der strukturentdeckenden und modellbildenden Methoden ist folgende: mittels des Analyseinstrumentariums wird Varianz im System erzeugt. Aufgrund des Wirkzusammenhangs zeigen abhängige Komponenten ebenfalls eine Varianz. Diese Korrelation wird mittels multivariater Methoden ausgewertet. Zur Präsentation müssen kurz die relevanten, statistische Begriffe gegeben werden.

Idee

In einer Stichprobe bzw. Datenerhebung mit n Untersuchungseinheiten seien die Werte $x_1, \ldots, x_n$ eines Merkmals X [ES00, S. 30]. Zufallsvariablen werden zu meist mit Großbuchstaben bezeichnet, entsprechende Realisationen mit Kleinbuchstaben und Indizes.

9.3.2.1 arithmetisches Mittel

arithmetische Mittel

Das arithmetische Mittel einer Stichprobe (der Durchschnitt) ist wie folgt definiert:

33 arithmetisches Mittel
Das arithmetische Mittel, der Durchschnitt, berechnet sich wie folgt:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

9.3.2.2 Kovarianz

Kovarianz

Die Kovarianz ist ein Maß für den Zusammenhang zweier Variablen oder Meßwerte. Die Kovarianz ist positiv bei positiven Zusammenhang, negativ bei reziprokem Zusammenhang.

34 Kovarianz
Die Kovarianz ist eine Maßzahl für den (linearen) Zusammenhang zweier Variablen. Es gilt:

$$cov(X, Y) = \frac{\sum_{i=1}^{n}(x_i - \bar{X})(y_i - \bar{Y})}{n}$$

In „*R*“ kann die Kovarianz mittels des Befehls cov berechnet werden.

9.3.2.3 Korrelation

Korrelation

Die Korrelation wird wie folgt definiert:

35 Korrelation
Die Korrelation ist ein Maß für den linearen Zusammenhang zweier Variablen. Es gilt:

$$Kor(X, Y) = r_{X,Y} = \frac{cov(X, Y)}{\sqrt{varX}\sqrt{varX}}$$

Besteht ein kausaler Zusammenhang, repräsentiert sich dieser in der Korrelation.

Scheinkorrelation

Eine Scheinkorrelation ist eine Korrelation ohne kausalen Zusammenhang. Ein berühmtes (Lehr-)Beispiel ist die regionale Korrelation zwischen Störchen und der Geburtsrate, der kausale Zusammenhang liegt jedoch in der Ländlichkeit der Region.
In dem Statistikprogram „*R*“ kann der Korrelationskoeffizient mit dem Befehl **cor** berechnet werden.
Eine Korrelationsmatrix zeigt durch hohe numerische Werte Zusammenhänge an und ist trivialerweise symmetrisch.

9.3.3 *Die Faktorenanalyse*

Exkurs 9.3.2 Entwicklung der Faktorenanalyse

Entwicklung der Faktorenanalyse

Auf der Basis von Arbeiten des Statistikers Karl Pearson entwickelte Charles Spearman für die Intelligenzforschung der wissenschaftlichen Psychologie die Faktorenanalyse [Spe04, S. 61]. 1904 spricht er von Faktoren als latente Größe [Spe04, S. 20] hinter kognitiven Fähigkeiten.

Spearman beobachtet Korrelationen in Intelligenztests und vermutet eine latente Größe, den allgemeinen Faktor für die Intelligenz (General Factor of Intelligence "g"), hinter kognitiven Leistungen wie „Auffassungsgabe, Logik, Kreativität, etc." [Spe27].

Reduzierung auf Faktoren

Die Faktorenanalyse findet eine weite Verbreitung in Geistes- und Naturwissenschaften. Sie wird benutzt um inhärente Strukturen in einer Menge von Daten aufzudecken [Wer95, S. 205]. Viele Merkmale (Variablen) werden auf weniger Faktoren, die die Zusammenhänge ausreichend genug beschreiben sollen, reduziert. [Man07] [Man+08e]

Variablenauswahl

Die Qualität der Faktorenanalyse ist von deren Ausgangsdaten und ihrer Erhebung abhängig. Vor allem Geisteswissenschaften erheben ihre Daten mittels Umfragen. Die hier verwendeten Daten sind Meßwerte eines Systems, daher es muss keine Variablenauswahl getroffen werden. Für fundierte Betrachtungen zur Variablenauswahl sei auf [BEP96, S. 269] verwiesen.

Das Fundamentaltheorem der Faktorenanalyse geht von der Annahme aus, dass sich jeder Meß- oder Beobachtungswert einer Variablen (x_j oder standardisiert z_j) sich als Linearkombination mehrerer (hypothetischer) Faktoren beschreiben lässt [BEP96, S. 278][Man07].

interpretative Probleme

Da diese Faktoren inhärente Zusammenhänge in den Merkmalsausprägungen als Zahlenwert beschreiben wird **für interpretative Zwecke** die Datenreduzierung und die damit verbundene Ungenauigkeit bewusst bezweckt. Zusätzlich dazu, dass ein Faktor nur ein abstrakter Zahlenwert ist, kann dies jedoch auch zu interpretativen Problemen führen. [Man07]

Transitive Abhängigkeiten

Durch die enorme Anzahl der gemessenen Daten des experimentellen Ansatzes kann nicht leicht ein Zusammenwirken festgestellt werden. Zudem ist das Zusammenwirken transitiv. Ist eine Komponente X von einer Komponente Y abhängig, welche wiederum von Komponente Z abhängig ist, müssen die Daten entsprechend gruppiert werden, um verständlich zu sein. [Man+08e]

Vorstellung eines Faktors

Informell kann ein Faktor als eine Kombination von Komponenten verstanden werden, die in einem Experiment mit dem Analyseinstrumentarium ein ähnliches Verhalten zeigen. Durch die Faktorenanalyse werden diese Komponenten entsprechend gruppiert. Die Faktorenanalyse nimmt eine Dimensionsreduzierung vor [MHR09]. [Man+08e] Mit möglichst wenig Dimensionen soll das Problem genau genug beschrieben werden. Bildlich könnte man sich eine Rotation der Dimensionen vorstellen, bis mit möglichst wenig Dimensionen die Meßwerte möglichst genau beschrieben werden.

Durchführung

Um eine Faktorenanalyse durchzuführen werden die Meßwerte in einer Matrix M gruppiert. Der Wert m_{ij} ist die Laufzeit von Komponente j beim experimentellen Lauf i. Diese Matrix wird zu einer Matrix Z normalisert/normiert, indem $z_{ij} = \frac{m_{ij} - \bar{m}_i}{\sigma_i}$ gesetzt wird (σ_i steht für die Standardabweichung, m_i für das arithmetische Mittel[1]). [Man+08e]
Aus dieser normalisierten/normierten Matrix der Meßwerte Z wird die Korrelationsmatrix R wie folgt berechnet [Man+08e]:

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

Darstellung mit Faktoren

Gesucht ist nun eine Darstellung von Z mittels der Faktoren P und den Faktorladungen A (die Faktorladungen beschreiben informelle etwa wie ein Maß der Zugehörigkeit, ein hoher Wert beschreibt eine hohe Zugehörigkeit), so dass gilt [Man+08e]:

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

Unter der Annahme der Existenz dieser Zerlegung wird folgende Herleitung genutzt [Man+08e]:

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

Sind die Faktoren (wie es gewünscht ist) unkorreliert, ist P^* eine Einheitsmatrix. Somit ergibt sich das Fundamentaltheorem der Faktorenanalyse [Man+08e]:

$$R = A \cdot P^* \cdot A^{\mathrm{t}}$$

9.3.4 *Beispiel zur Faktorenanalyse*

Beispiel

Ein kleines Beispiel soll das Vorgehen verdeutlichen: In Tabelle 9.1(a) sind Messungen von drei Komponenten a, b und c angegeben. Diese werden normalisiert/normiert und einer Faktorenanalyse unterzogen. Es ergibt sich die Tabelle der Faktorladungen (Tabelle 9.1(b))

[1] siehe Definition 33, Seite 210

(a) Beispielsmessungen M

	a	b	c
1	6	2	1
2	3	4	2
3	3	4	2
$\bar{m}_i$	4	3,33	1,67
σ_i	1,73	1,15	0,58

(b) Faktorladungen

	PC1	*PC2*	*PC3*
1	0.771	-0.637	0
2	-0.570	-0.689	0.447
3	-0.285	-0.344	-0.894

Tabelle 9.1: Beispiel zur Faktorenanalyse

Die Faktorenanalyse kann für diese Arbeit nur kurz eingeführt werden. Für weiterführende Informationen sei auf Backhaus, Erichson und Plinke [BEP96] und Mangold [Man07] verwiesen.
Trotz dieser eleganten Herleitung erfolgt die konkrete Implementierung einer *Faktorenanalyse* meist basierend auf einer Regressionsanalyse [BEP96]. Insbesondere deswegen und wegen der Stabilität müssen mind. 3 Meßwerte je Komponente vorliegen. In *R* kann die Faktorenanalyse mit `factanal`, die *Hauptkomponentenanalyse* (eine Faktorenanalyse wie Backhaus, Erichson und Plinke [BEP96] mehrfach betonen) mit `princomp` und `pca` durchgeführt werden.
Für viele Informatiker ist die Faktorenanalyse schwer zugänglich, insbesondere da in einem Studium der Informatik nicht exzessiv in der Statistik ausgebildet wird. Iterative Clusteringalgorithmen sind eingängiger und beliebter.

schwer zugänglich

9.3.5 Clusteringverfahren

Intention

Das Ziel von Clusteringverfahren ist es, (große) Datenmengen automatisch bzw. semi-automatisch in *Cluster* (Kategorien, Klassen oder Gruppen) aufzuteilen, dabei sollen Objekte mit großer Ähnlichkeit zu gemeinsamen *Clustern* zugeordnet werden [ES00].

DBSCAN
Optics

Forschung in den Gebieten *Data Mining*, der Statistik, *Mustererkennung*, dem maschinellen Lernen und angewandten Wissenschaftsdisziplinen wie der *Bioinformatik* haben zu einer hohen Anzahl von *Clusteringverfahren* geführt [KKZ09]. An der *LMU* wurde und wird hierzu Pionierarbeit geleistet. Vielzitierte Algorithmen wie z. B. *DBSCAN* [Est+96; San+98] oder *Optics* [Ank+06; BKS00] sind hier entstanden.

spezifische Algorithmen

Es gibt keinen generellen Clusteringalgorithmus der universell auf alle Probleme anwendbar ist [KKZ09]. Kriegel, Kröger und Zimek [KKZ09] verweisen in der Arbeit „*Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering*“ darauf, dass die Auswahl des Verfahrens vom Problem und der Intention der Interpretation abhängt. Sie geben einen profunden Überblick und eine sehr schöne tabellarische Zusammenfassung unterschiedlicher Clusteringalgorithmen [KKZ09, S. 42].

Ester und Sander [ES00] unterscheidet in dem Lehrbuch „*Knowledge Discovery in Databases*“ zwischen partitionierenden und hierarchischen Clusteringverfahren.

K-Means

Han, Kamber und Pei [HKP06] sprechen von *K-Means*, neben *k-medoids*, als das bekannteste und meist benutzte partitionierende Verfahren [HKP06, S. 402]. *K-Means* [Mac67] ist ein sehr einfacher Algorithmus, bei dem vor dem Start die Anzahl der Cluster festgelegt wird. Diese Anzahl werden als initiale Schwerpunkte willkürlich verteilt, die Daten werden via einer Distanzfunktion zu dem nächsten Schwerpunkt zugeordnet. Die Schwerpunkte werden erneut berechnet und das wird Verfahren so lange durchgeführt bis sich die Schwerpunkte nicht mehr ändern oder eine Iterationsgrenze erreicht ist. [Mac67]

Die Implementation in *R* von *K-Means* wird in Abschnitt 9.10 auf das Szenario 10 (siehe Seite 35), dem Modelchecker cmc [Ham06][Ham09, Kapitel 6][HW06a][HW06b], angewandt.

Daneben wird das hierachische Clusteringverfahren von R (Befehl: `hclust`) genutzt. Nach der R Documentation [R D] sind die Details zu dem verwendete Algorithmus in [Mur85] zu finden.

9.4 Modellbildende Methoden

Graphen

Wichtige Methoden in der Softwareentwicklung, -analyse und -wartung sind Graphen (z. B. der *Call-Graph* [Ryd79]). Graphen reduzieren auf wesentliche Information und sind dadurch besser zur Visualiserung von Zusammenhängen geeignet. [Mano7, Kapitel 5]

Resource Dependence Graph

Ähnlich einem *Program Dependence Graph* [Krio4; FOW87] wurde zur Performanzanalyse der *Resource Dependence Graph* oder *Ressource Dependence Graph*[2] (kurz *RDG*) in [Mano7, Kapitel 5] entwickelt und als Patent angemeldet [Man+o8c; Man+o8d; Man+07; Man+o8a; Man+o8b].

Vorgehen

Hierbei wird die Laufzeit einzelner Komponenten (z. B. eines Moduls) um eine gewisse Laufzeit prolongiert. Das System wird anschließend ausgeführt. Durch den Wirkzusammenhang weisen abhängige Komponenten eine Performanzänderung auf, die für ein Modell rekonstruiert werden.

$O(n)$

Dies kann iterativ geschehen (dies entspricht einer Laufzeitkomplexität von $O(n)$ für das Experiment), optimal sollte die Anzahl der Experimente möglichst gering gehalten werden. [Mano7, Kapitel 5]

$O(1)$

Um ein Modell aus nur einem Experiment zu gewinnen (dies entspricht einer Laufzeitkomplexität von $O(1)$ für das Experiment) wäre die Idee nun folgende: verschiedene Komponenten werden um bestimmte Zeiteinheiten prolongiert, beispielsweise Komponente K_n um x_n.

Die prolongierte Zeitdifferenz x_n für Komponente K_n muss nun geschickt gewählt werden.

9.4.1 Diophantische Gleichungen

diophantische Gleichung

Bei natürlichen Zahlen ($x_n \in \mathbb{N}$) erhält man eine *diophantische Gleichung* (vgl. dazu Hilberts zehntes Problem [Mat93]). Eine diophantische Gleichung ist ein Gleichungssytem, bei dem nur Koeffizienten aus $\mathbb{N}$ zugelassen sind. Also beispielsweise die Anzahl der Aufrufe und ein Laufzeitdelta für die prolongierte Komponente. Dies kann jedoch nicht eindeutig bestimmt werden. [Mano7]

$$\begin{aligned} a + b &= 2 \\ 2a + 2b &= 4 \end{aligned}$$

Die Zahlenpaare $(0, 2)$, $(2, 0)$ sowie $(1, 1)$ sind die ganzzahligen, positiven Lösungen für diese Gleichung. [Mano7]

[2] Wahrscheinlich bedingt durch die automatische Rechtschreibkorrektur in einer Patentabteilung.

9.4.2 Die b-adische Entwicklungen

b-adische Entwicklungen

Zur Lösung bieten sich *b-adische Entwicklungen* an, wie ein Beispiel mit der Basis $b = 10$ von Mangold [Mano7] zeigt:

$$\sum a_i \cdot b^i = a_1 \cdot b + a_2 \cdot b^2 + a_3 \cdot b^3 + a_4 \cdot b^4$$

$$1230 = a_1 \cdot 10 + a_2 \cdot 100 + a_3 \cdot 1000 + a_4 \cdot 1000$$
$$\Rightarrow a_1 = 3, a_2 = 2, a_3 = 1, a_4 = 0$$

Problematisch hierbei ist jedoch, dass die Basis größer gewählt werden muss, als die Aufrufhäufigkeit aller Module [Mano7, S. 85]. Das kann wiederum zu extrem langen Wartezeiten bei einem Experiment führen (durch die hinzugefügten Laufzeiten).

9.4.3 Logarithmus von Primzahlen

Primfaktorzerlegung

Eine von Mangold [Mano7] vorgeschlagene und umgesetzte Lösung ist die häufig in der Kryptographie genutzte eindeutige *Primfaktorenzerlegung*. Da sich jedoch die Laufzeiten der Module addieren muss oder kann hier folgende Rechenregel genutzt werden [Mano7, S. 86]:

$$log_a(x_1 \cdot x_2 \cdot ... \cdot x_n) = log_a(x_1) + log_a(x_2) + ... + log_a(x_n)$$

Logarithmus

Der Logarithmus eines Produkts ergibt sich als die Summe der Logarithmen der Faktoren. Die bedeutet, wenn jede Komponente um die Laufzeit eines Logarithmus von einer Primzahl prolongiert wird, ist die Primfaktorenzerlegung der potenzierten Laufzeitdifferenzen wieder eindeutig. Damit kann aus den Laufzeitdifferenzen zum Referenzlauf rekonstruiert werden von welchem Modulen ein gemessenes Modul abhängt. [Mano7]

An Szenario 1 und Szenario 9.8 wird der *Resource Dependence Graph* illustriert.

9.4.4 Diskussion

Diskussion

Eine Erweiterung des *Resource Dependence Graphen* auf parallele Systeme ist dann möglich, wenn der Scheduler kontrolliert werden kann oder/und die Prolongation nicht unterbrechbar ist. Der *Resource Dependence Graph* dient als ein Beispiel für modellbildende Analysemethoden, viele weitere sind denkbar. Beispielsweise ist auch ein modellbildender Algorithmus möglich, der die Transitivität der Prolongation ausnutzt.

9.5 Strukturentdeckende Performanzanalyse von Illustrationsszenario 1 – Methodenaufrufe

Das Illustrationsszenario 1 wurde auf Seite 22 als didaktisches Beispiel gegeben. Es eignet sich durch die einfache Struktur gut zum Nachvollziehen. Es ist angelehnt an Szenario 1b aus [Mano7], welches sich zur Präsentation der Technik bewährt hat. Illustrationsszenario 1

9.5.1 Experiment und Datenerhebung

Zum Erheben der Daten für das Experiment bzw. dem *Logging* der Modullaufzeiten wurde der in Abschnitt 7.1 implementierte Aspekt, Version 2 zum *Tracing* sowie ein Aspekt zur *Prolongation* eingewebt. Die sich ergebende *Tracefile* – nach einer Ausführung des Szenarios – ist in Tabelle 9.2 wegen der Anzahl der Meßdaten ausschnittsweise dargestellt. Datenerhebung

0	Thread[main,5,main]	call(long Ill. . . fe.fac_plus(long, long))	3768915	Enter
1	Thread[main,5,main]	call(long Ill. . . fe.fac(long))	5234185	Enter
0	Thread[main,5,main]	call(long Ill. . . fe.fac(long))	5834820	Exit
1	Thread[main,5,main]	call(long Ill. . . fe.fac(long))	6044623	Enter
0	Thread[main,5,main]	call(long Ill. . . fe.fac(long))	6637715	Exit
-1	Thread[main,5,main]	call(long Ill. . . fe.fac_plus(long, long))	6793601	Exit
0	Thread[main,5,main]	call(long Ill. . . fe.fac_mal(long, long))	6987201	Enter
1	Thread[main,5,main]	call(long Ill. . . fe.fac(long))	7419658	Enter
0	Thread[main,5,main]	call(long Ill. . . fe.fac(long))	7672763	Exit
. . .	. . .	. . .	. . .	. . .

Tabelle 9.2: Ausschnitt aus dem erstellten Trace-File. Wegen der Seitenbreite dieses Dokuments wurden bei `IllustrationsszenarioMethodenaufrufe.IllustrationsszenarioMethodenaufrufe` Punkte eingesetzt, um die Breite zu reduzieren. Gemessen wurde auf einem *Windows XP* Rechner, *Pentium 4* (3 GHz), *Java 1.6.0_20* mit den Zahlenwerten $a = b = c = d = 2$ für Szenario 1
.

9.5.2 Anpassen und Weiterverarbeitung der Tabelle

Die Tabelle wurde in einem Tabellenkalkulationsprogramm transponiert und geordnet und ergibt eine Tabelle analog zu Tabelle 9.5.2.[3] Anpassen
Die Tabelle kann, so wie sie oben abgebildet ist, mit `tab <-read.table (PFADZURDATEI ,header=TRUE)` in das Statistikprogramm R eingelesen werden. Einlesen

[3] Die Daten der Tabelle wurden angepasst an [Mano7].

fac1	fac2	fac_plus	fac3	fac4	fac_mal	main
1179759	1167187	4015315	1149588	1748267	3627556	14431164
5083607	5082489	10499379	5139480	5270781	16399012	27848232
10697728	10106033	21072231	10089271	10069995	20363482	41711726
67327	68444	1641829	61181	67048	298641	2246934
121244	65651	5501538	62858	64254	293333	6045181
103086	66768	10924853	62857	68445	301155	11471569
604267	68165	895924	508444	59505	1768661	2907074
55873	59505	269588	79619	59505	5321626	5834261
56990	60902	271543	110628	58946	10975417	12060751

9.5.3 *Datenprüfung*

Nun kann optional geprüft werden, ob die gemessenen Daten adäquat sind und keine zu hohen Ausreißer (beispielsweise durch Scheduler oder ähnliches) beinhalten.

```
x<-c(1:9)

y1<-tab[,1]
y2<-tab[,2]
y3<-tab[,3]
y4<-tab[,4]
y5<-tab[,5]
y6<-tab[,6]
y7<-tab[,7]

plot (x,y7,col="black",pch=15,xlab="Nummer des Laufs", ylab="gemessene
    Laufzeitdauer in Nanosekunden", main="")
lines (y7,col="black")
points (y1,col="blue",pch=16)
lines (y1,col="blue")
points (y2,col="green",pch=17)
lines (y2,col="green")
points (y3,col="purple",pch=17)
lines (y3,col="purple")
points (y4,col="brown",pch=18)
lines (y4,col="brown")
points (y5,col="yellow",pch=19)
lines (y5,col="yellow")
points (y6,col="red",pch=20)
lines (y6,col="red")
```

Listing 9.1: R: Daten überprüfen

Auf Basis dieser Daten wird mit `tab.pca <-prcomp (tab)` eine *Hauptkomponentenanalyse* durchgeführt.

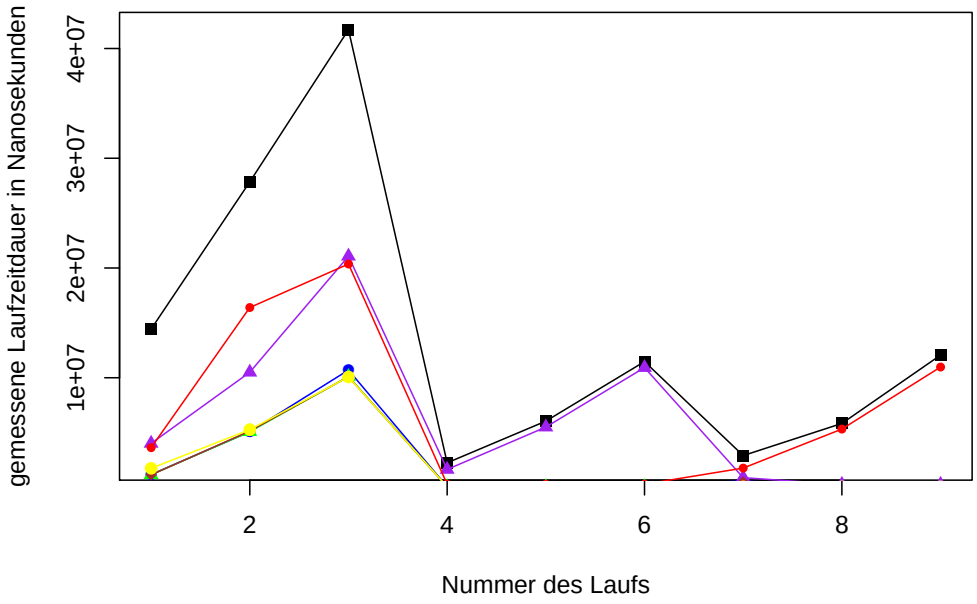


Abbildung 9.4: Messung der Methodenlaufzeiten bei Illustrationsszenario 1.
❑ Die schwarzen Messwerte ist die Laufzeitdauer der `main`-Methode.
❑ Bei den ersten drei Läufen wurde die Methode `fac` prolongiert (grüne, gelbe, braune und blaue Meßwerte).
❑ Bei den Läufen 4 bis 6 wurde die Methode `fac_plus` prolongiert (lila).
❑ Bei den Läufen 4 bis 6 wurde die Methode `fac_mal` prolongiert (rot).
❑ Bei Lauf zwei unterscheiden sich die Meßwerte der Methoden `fac_plus` und `fac_mal`, was offensichtlich eine Meßungenauigkeit darstellt.
Inkorrekterweise wurden zur besseren Darstellung die Meßpunkte durch Linien verbunden.

PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8
fac1	0.20	0.04	0.51	0.06	-0.65	0.42	-0.32
fac2	0.20	0.029	0.43	0.07	0.42	0.49	0.59
fac_plus	0.35	0.72	-0.07	-0.59	0.06	-0.02	-0.05
fac3	0.19	0.02	0.46	0.06	-0.24	-0.74	0.38
fac4	0.20	0.03	0.39	0.22	0.58	-0.19	-0.636
fac_mal	0.40	-0.70	0.01	-0.59	0.05	-0.01	-0.06
main	0.75	-0.01	-0.44	0.48	-0.08	0.02	0.05

Tabelle 9.3: Faktorladungen (gerundet) einer Hauptkomponentenanalyse von Illustrationsszenario 1.

Darstellung

Die Ergebnisse der Hauptkomponentenanalyse werden mittels des Befehls `biplot` multidimensional skaliert angezeigt (siehe Abbildung 9.5.3.1).

```
biplot (tab.pca, ylab=c("Hauptkomponente␣2␣(PCA2)"), xlab=c("Hauptkomponente
    ␣1␣(PCA1)"), xlabs=c("","","","","","","","",""))
```

Listing 9.2: R: Biplot der Hauptkomponentenanalyse

9.5.3.1 Hierarchisches Clustering von Illustrationsszenario 1

Vorverarbeitung für ein Clustering

Aus den gemessenen Daten kann man sich eine Kovarianzmatrix erstellen lassen (Befehl: **cov**). Komponenten mit einem Wirkzusammenhang können anhand der Messwerte an der Kovarianz identifiziert werden. Erstellt man sich die Kovarianzmatrix und und wendet man auf diese eine Distanzfunktion an (Befehl: `dist`) können die Werte geclustert werden. Die Messwerte von Komponenten mit Wirkzusammenhang werden in ein Cluster zusammengefasst. In Abbildung **??** wurde ein hierarchisches Clusteringverfahren auf die so vorverabeiten Daten angewandt.

```
plot (hclust (dist (cov (tab)),method="ward"),xlab="", main="",ylab="")
```

Listing 9.3: R: Plot eines hierarchischen Clusterings

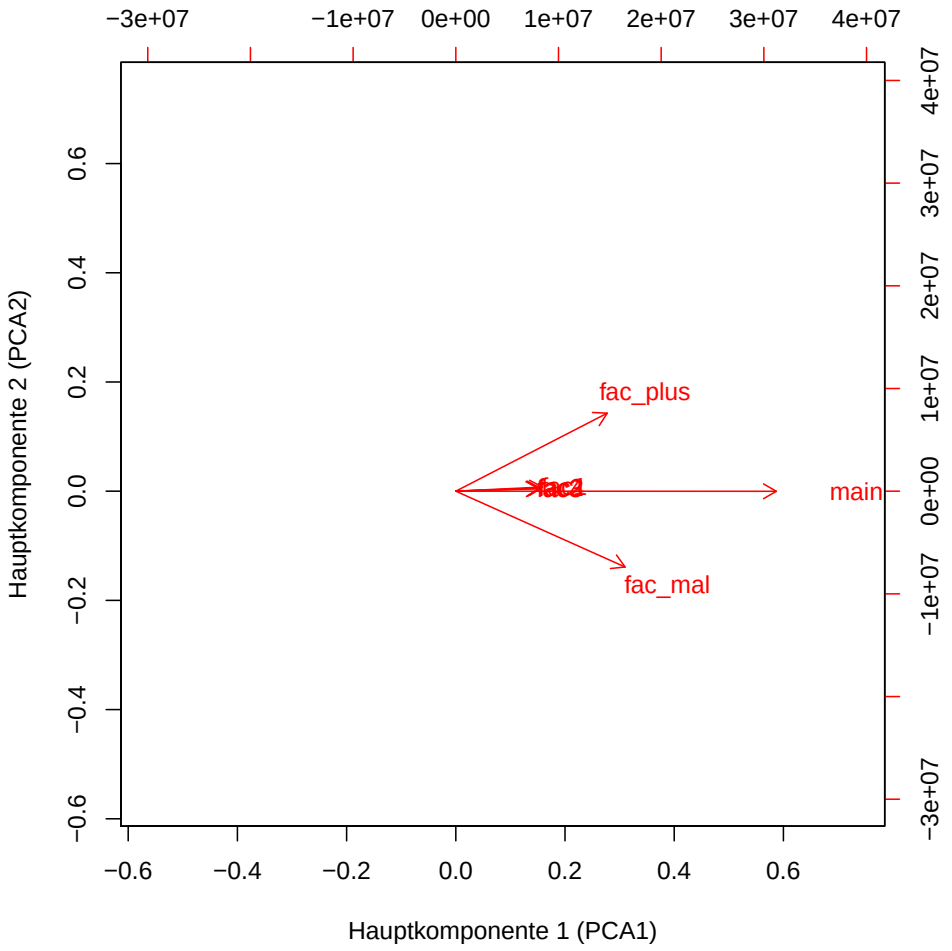


Abbildung 9.5: Biplot von Szenario 1.

❑ Die einzelnen **fac**-Methoden laden direkt auf die Komponente auf der auch **main** liegt auf.

❑ Eine Verbesserung der Laufzeit einer **fac**-Methode wird eine Verbesserung der Laufzeit von **main** nach sich ziehen.

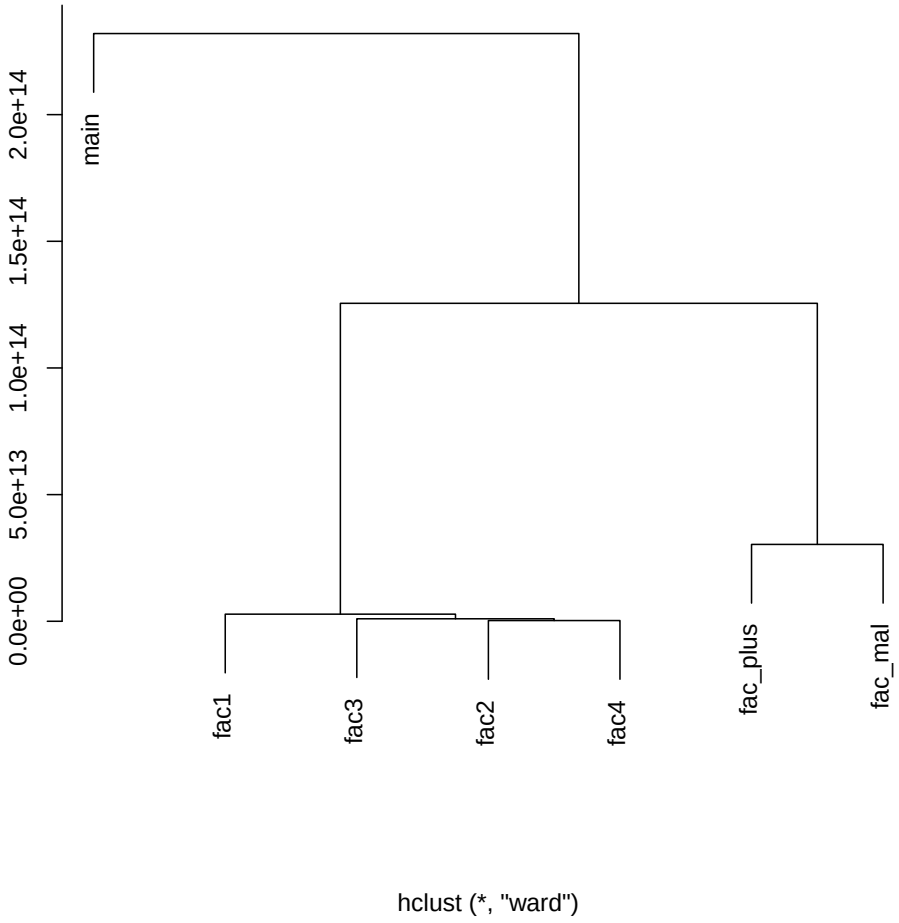


Abbildung 9.6: Hierarchisches Clustering von Szenario 1

❑ Die Kovarianz der gemessenen Methodenlaufzeit wurde mittels `cov(tab))` in *R* berechnet.

❑ Mittels `dist(cov(tab)))` wurde eine *Distanzmatrix* auf diese Kovarianzmatrix angewandt.

❑ Das Ergebnis wurde mittels `plot(hclust())` dargestellt.

9.6 *Modellbildende Performanzanalyse von Illustrationsszenario 1 – Methodenaufrufe*

Hier wird das Szenario 1 zwei mal ausgeführt. Der erste Lauf ist ein um 0 Zeiteinheiten prolongierter Referenzlauf. Beim zweiten Lauf werden alle Methoden um den Logarithmus einer Primzahl prolongiert.

- Die Methode `fac` wird beispielsweise um $log_2 2$ Zeiteinheiten prolongiert,
- die Methode `fac_plus` wird beispielsweise um $log_2 3$ Zeiteinheiten prolongiert,
- die Methode `fac_mal` wird beispielsweise um $log_2 5$ Zeiteinheiten prolongiert,

wobei die Zeiteinheiten gegenüber der Meßungenauigkeit genügend groß gewählt werden müssen. Analog können die Zeiteinheiten mit einem Faktor multipliziert werden, bei der Auswertung muss wieder durch diesen Faktor geteilt werden. [Mano7, S. 89]
Das prolongierte System wurde auf einem Ubuntu System ausgeführt und ergibt die Messwerte aus Tabelle 9.4. [Mano7, S. 89]

Methodenname	$t/\mu s$	$t'/\mu s$	$\Delta t/\mu s$
long fac(long)	24	10026	10002
long fac(long)	31	10028	9997
void fac_plus(long, long)	129	35984	35855
long fac(long)	24	10052	10028
long fac(long)	23	10027	10004
void fac_mal(long, long)	110	43393	43283
void main(String[])	422	79513	79091

Tabelle 9.4: Die Laufzeitwerte zum Resource Dependence Graph von Szenario 1 mit einer Prolongation um 0 Zeiteinheiten und einer Prolongation um den Logarithmus einer Primzahl mal 10000 Nanosekunden (wegen Messungenauigkeiten).

- In der Spalte t' stehen die gemessenen Werte des instrumentierten Systems.
- In der Spalte $\Delta t/\mu s$ sind die Laufzeitdifferenzen zum nicht prolongierten Programm angegeben.

Die Tabelle wurde zitiert von [Mano7, S. 90].

Anschließend muss eine Primfaktorenzerlegung vorgenommen werden (siehe Tabelle 9.5). [Mano7]

Methodenname, ID	$\Delta t/10ms$	$2^{\Delta t}$	Gerundet	Faktoren
fac, 2	1	2	2	2
fac, 2	1	2	2	2
fac_plus, 3	3,59	12	12	2*2*3
fac, 2	1	2	2	2
fac, 2	1	2	2	2
fac_mal, 5	4,33	20,09	20	2*2*5
main, -	7,91	240,37	240	2*2*2*2*3*5

Tabelle 9.5: Die Primfaktorenzerlegung zum Resource Dependence Graph von Szenario 1.
❑ In der Spalte Δt / $10ms$ wurden die Zeitdifferenzen angegeben.
❑ In Spalte $2^{\Delta t}$ wurden die Werte potenziert, diese Werte wurden gerundet.
❑ In der Spalte „Faktoren“ ist die vorgenommene *Primfaktorenzerlegung* angegeben.
Die Tabelle wurde zitiert von [Mano7, S. 90].

Der Graph kann nun leicht aus Tabelle 9.5 (semi-)automatisch konstruiert werden.
❑ Alle Methoden aus der Spalte „*Methodenname, ID*“ bilden einen Knoten.
❑ Alle „*IDs*“ aus der Spalte Faktoren, die nicht der eigenen *MethodenID* entsprechen, gehen als Kante in diesen Knoten (Methode) ein (bzw. gehen von diesem Knoten aus).
❑ Die Anzahl der fremden „*IDs*“ aus der Primfaktorzerlegung ist das Kantengewicht.
Der resultierende Graph ist in Abbildung 9.7 dargestellt. [Mano7]

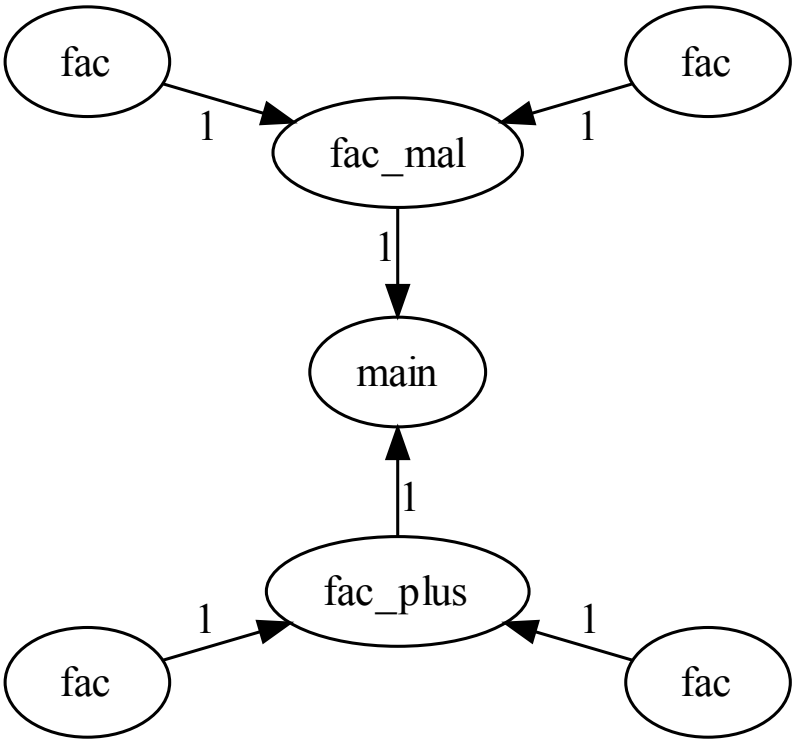


Abbildung 9.7: Der resultierende Resource Dependence Graph (RDG) von Szenario mit einem eigenen Knoten pro Aufruf.

❑ Die Struktur der Anwendung ist leicht zu erkennen.

❑ Die Methode `main` ruft ein mal die Methode `fac_plus` und die Methode `fac_mal` auf, die Performanz von `main` hängt direkt von der Methoden `fac_plus` und der Methode `fac_mal` ab.

❑ Die Methoden `fac_plus` und `fac_mal` rufen jeweils zwei mal die Methode `fac` auf.

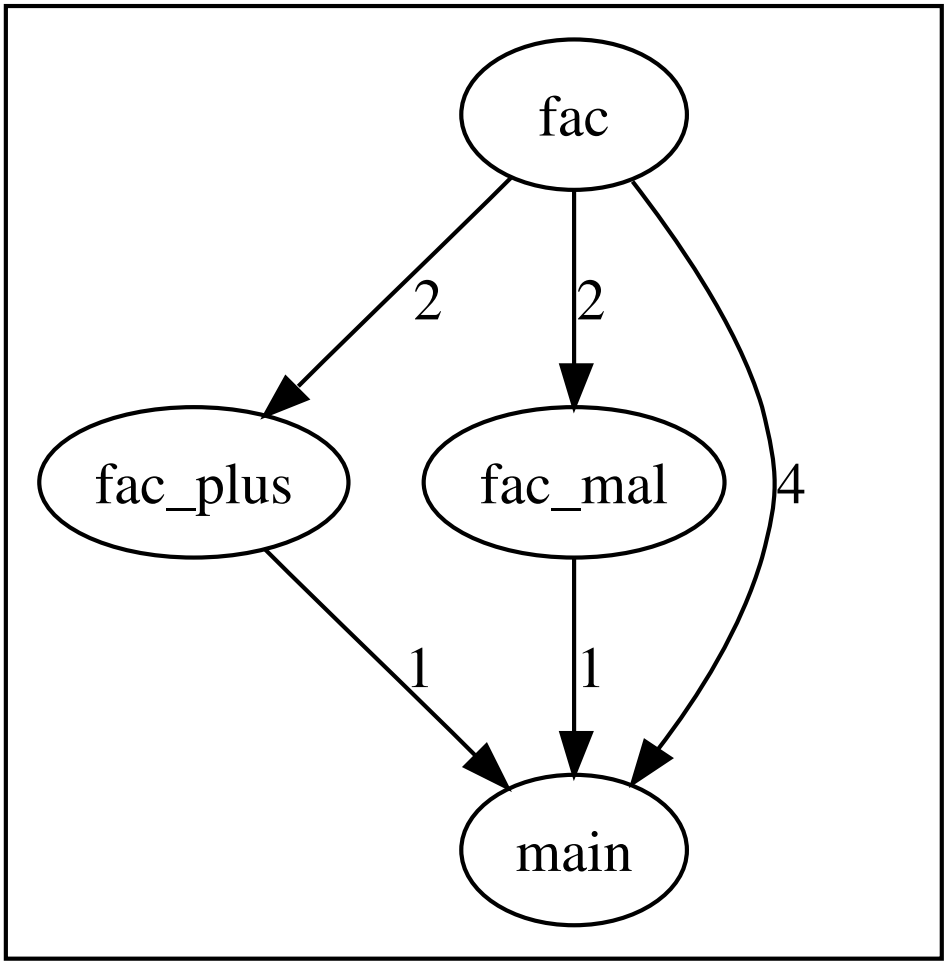


Abbildung 9.8: Der resultierende Resource Dependence Graph (RDG) von Szenario 1 mit Kantengewichten.
❑ Die Struktur der Anwendung ist leicht zu erkennen.
❑ Die Methode `main` ruft ein mal die Methode `fac_plus` und die Methode `fac_mal` auf.
❑ Die Methoden `fac_plus` und `fac_mal` rufen jeweils zwei mal die Methode `fac` auf.
❑ Dadurch sind die transitiven Abhängigkeiten der Methode `main` auf die Methode `fac` erkennbar.
Diese Grafik befindet sich analog in [Mano7, S. 91].

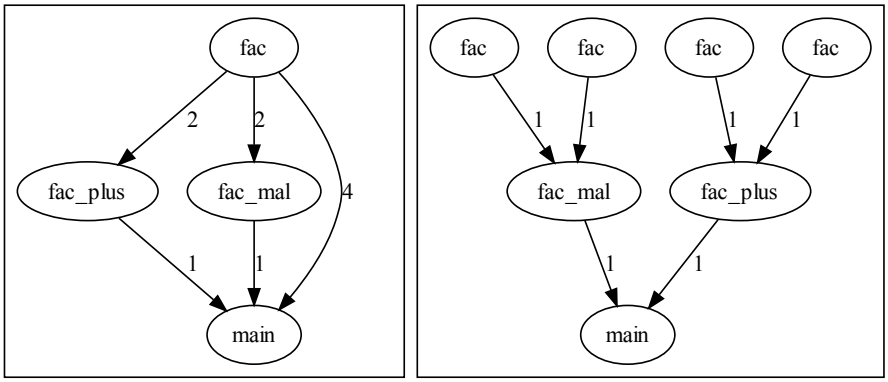


Abbildung 9.9: Ein Vergleich beider Darstellungsformen.
❑ Links ist jede Methode nur einmal im Graphen dargestellt. Die transitiven Abhängigkeiten können direkt visualisiert werden (z. B. `fac` → `main`). Kantengewichte präsentieren die Anzahl der Aufrufe.
❑ Rechts wird jeder Aufruf einer Methode mit einem eigenen Knoten dargestellt. Durch diese Darstellungsform kann auf Kantengewichte verzichtet werden.

9.7 *Simulationsansatz von Szenario 2 – Methodenaufrufe*

Das Illustrationsszenario 2 wurde auf Seite 24 (Listing 2.2) als didaktisches Beispiel gegeben. Es eignet sich durch die noch einfachere Struktur als Szenario 1 gut zum Nachvollziehen.
Es ist angelehnt an Szenario 1a aus [Mano7], welches sich zur Präsentation des simulierten Optimierens bewährt hat.
Das Listing aus Anhang B wurde eingewebt, mittels `&&! cflow **.2Simulate` wurde die Methode `CopyOffac` als simuliert optimierte Methode spezifiziert. Diese Methode wird (relativ) um die doppelte Laufzeit verbessert.
Beim Start dieser Anwendung können die Ergebnisse dieser (relativen) Optimierung direkt ausgemessen werden.

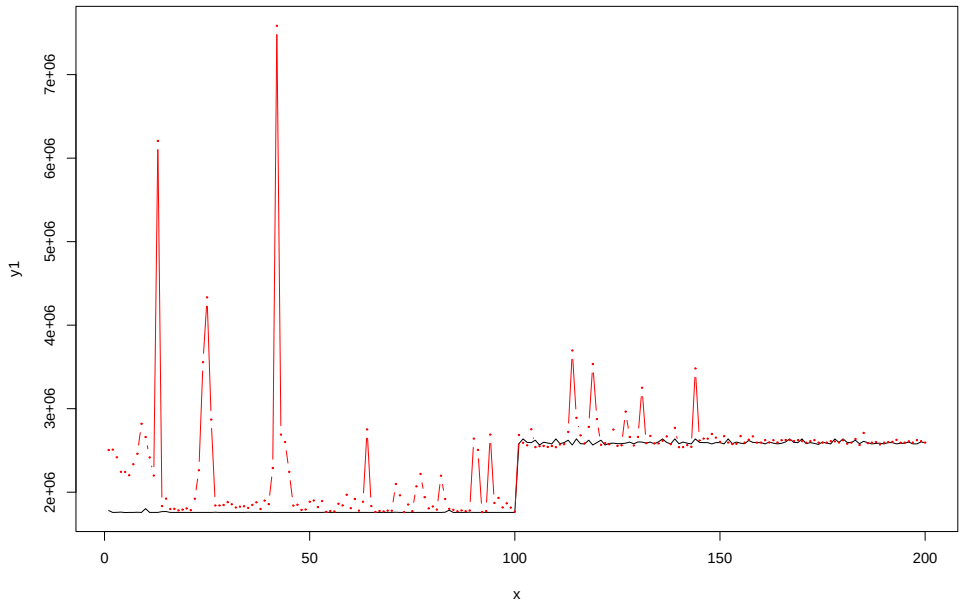


Abbildung 9.10: Unterschiede bei Messungen mit geringer Systemlast und hohe Prozesspriorität vs. hoher Systemauslastung und normaler Prozesspriorität.
❑ Die X-Achse repräsentiert die Nummer der Messung.
❑ Auf die Y-Achse sind die Laufzeiten von **einer** `fac`-Methode von Szenario 2 aufgetragen. Die Laufzeiten sind abhängig von der Hardware, entscheidend ist der relative Abstand.
❑ Ab Messung 100 wurde die Methode um den doppelten Faktor prolongiert.
❑ Die roten Messwerte sind Messwerte von einem System mit nebenläufigen Prozessen und Anwendungen. Die Spitzen in den Messwerten werden verursacht durch nebenläufige Anwendungen und Prozesse. Zu bemerken ist, dass dies keine Meßfehler sind, sondern akurate Messungen. Durch den *Scheduler*, oder ähnliches, verzögert sich die Ausführung.
❑ Die schwarzen Messwerte sind die Ausführung des gleichen Szenarios mit hoher Prozesspriorität und keinen nebenläufigen Anwendungen im System.
Die Grafik wurde zitiert von [Mano7].

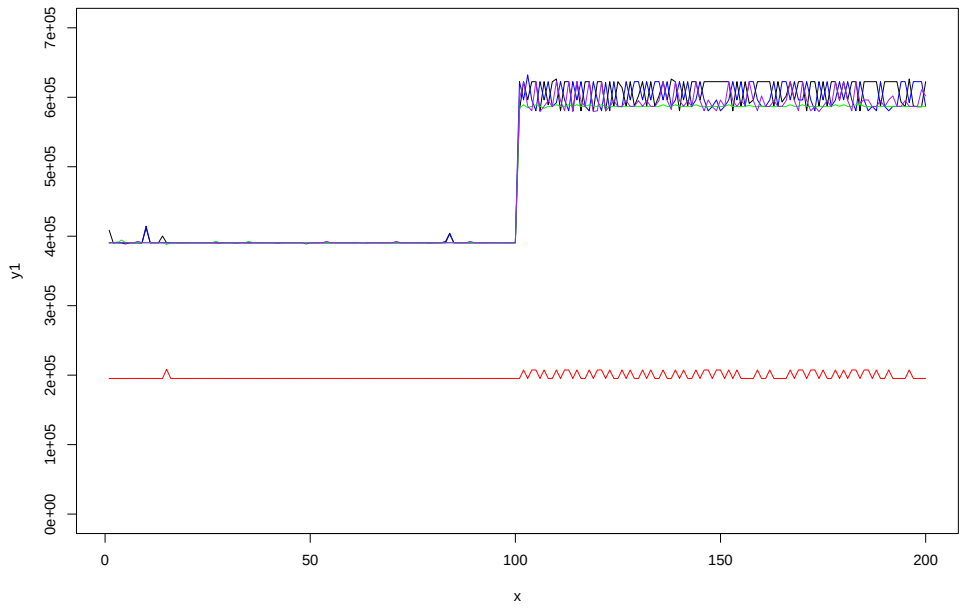


Abbildung 9.11: Die Messwerte des simulierten Optimierens von Szenario 2.
❑ Die X-Achse repräsentiert die Nummer der Messung.
❑ Auf die Y-Achse sind die Laufzeiten der `fac`-Methoden von Szenario 2 **absolut** aufgetragen.
❑ Die rote Linie stellt die Laufzeit von `CopyOffac` dar. In den ersten hundert Messungen wurde sie (relativ) doppelt so schnell simuliert, in den Messungen von 100 bis 200 dreifach so schnell.
Die Grafik wurde zitiert von [Mano7].

9.8 Modellbildende Performanzanalyse von Szenario 3 – Imageshuffle

Abbildung 9.12 zeigt einen mit dem Open Source Tool *GraphViz* [Ell+01] generierten *Resource Dependence Graph* von Szenario 3.
Für eine bessere Darstellung wurden die Methodennamen mit einem eindeutigen Bezeichner versehen, bestehend aus einem „p“ und einer fortlaufenden Nummer. Der sich aus diesen Daten ergebende Graph ist in einem hierarchischen Layout in Abbildung 9.12 dargestellt. Für jede Abhängigkeit wurde eine entsprechende Kante eingezeichnet, auf Kantengewichte wurde wegen graphischen Überschneidungen verzichtet. Das Szenario 3 wurde auf Seite 25 vorgestellt und wurde bereits in [Mano7] als Praxisbeispiel genutzt. [Mano7]

reduzierte Darstellung

Abbildung 9.13 zeigt die gleichen Daten, repräsentiert als Graph mit einem „*radial layout*“ nach Wills [Wil97].

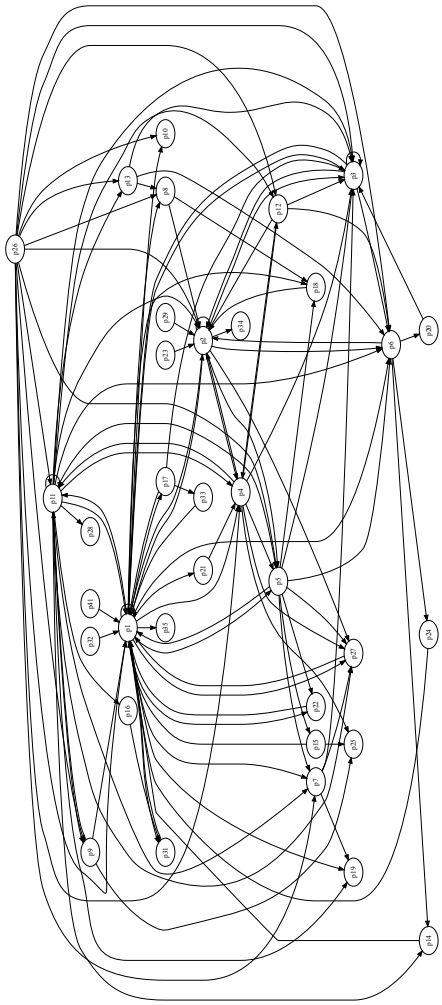


Abbildung 9.12: Gekürzter Resource Dependence Graph zu Szenario 3 – Imageshuffle.
❑ Methodennamen wurden mit einem eindeutigen Bezeichner versehen.
❑ Unabhängige Methoden wurden entfernt um einen besseren Überblick zu schaffen.

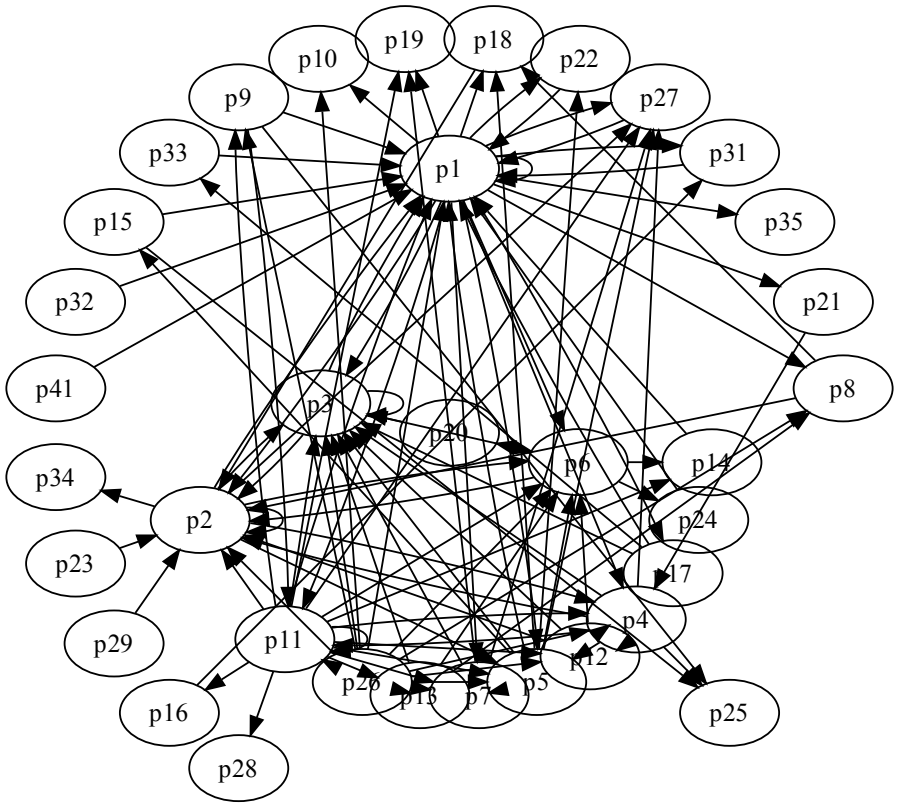


Abbildung 9.13: Gekürzter Resource Dependence Graph zu Szenario 3 (Imageshuffle) im *radial layout* nach Wills [Wil97].

9.9 Strukturentdeckende Performanzanalyse von Szenario 9 – Webframe

Szenario Webframe

Das Szenario 9 wurde in Abschnitt 2.2.7 auf Seite 34 definiert. Der Originalcode ist mit aktuellen Java-Versionen nicht lauffähig und kann von `http://java.sun.com/developer/technicalArticles/ThirdParty/WebCrawler/WebCrawler.java` heruntergeladen werden. Eine momentan lauffähige Version kann von `http://www.florian-mangold.com/diss/WebCrawler.java` bezogen werden.
Wie im Code ersichtlich ist, wird die ganze Verarbeitung in diesem nebenläufigen Webcrawler wird von der `run()`-Methode übernommen.

- ❑ Analog zu Abschnitt 9.5 wird in diesem Szenario eine Faktorenanalyse durchgeführt.
- ❑ Abbildung 9.14 zeigt einen Screeplot der Faktorenanalyse.
- ❑ Abbildung 9.15 zeigt den Biplot von einer Hauptkomponentenanalyse. Hier ist die `run`-Methode entsprechend dominant.
- ❑ Abbildung 9.9 stellt das hierarchische Clustering dar.

Der Optimierungskandidat kann zwar identifiziert werden (`run()`-Methode), dieses Szenario ist ein sehr undankbarer Anwendungsfall für das Analyseinstrumentarium. Es gibt hier (kaum) versteckte Abhängigkeiten.

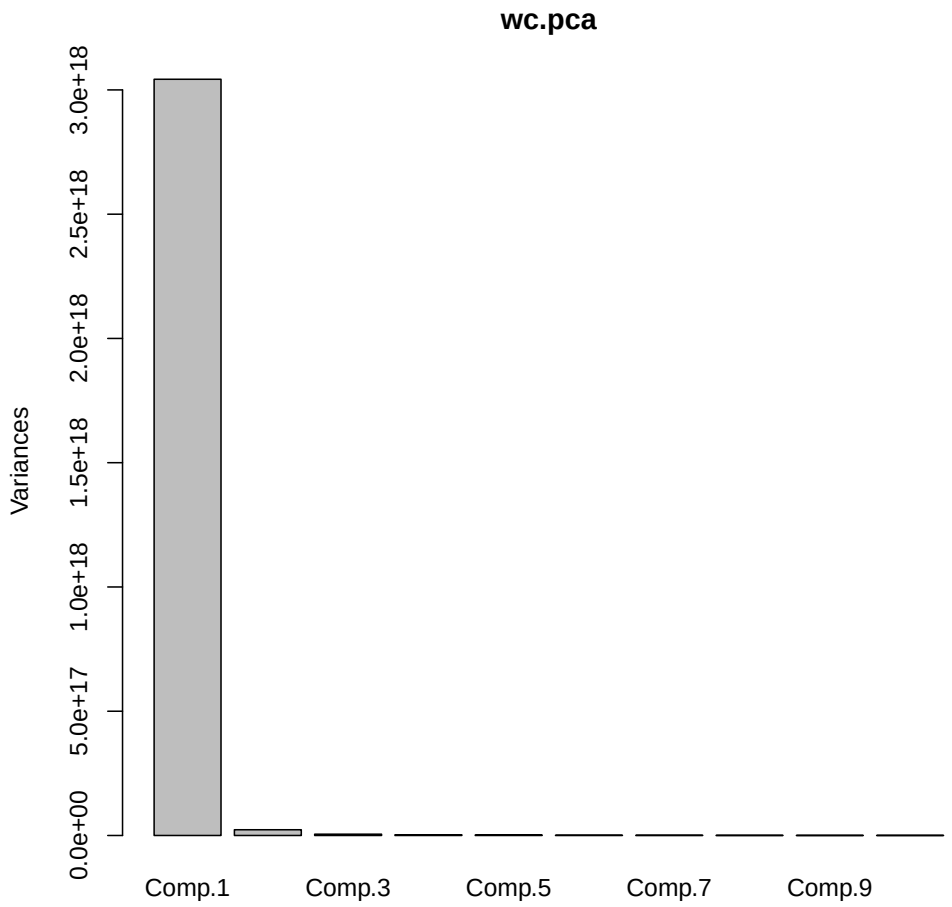


Abbildung 9.14: Screeplot von Szenario 9 – Webframe.

❑ Ein Faktor erklärt die Varianz in diesem Experiment hinreichend genau.

❑ In diesem System wird die Performanz nur durch einen Faktor bestimmt.

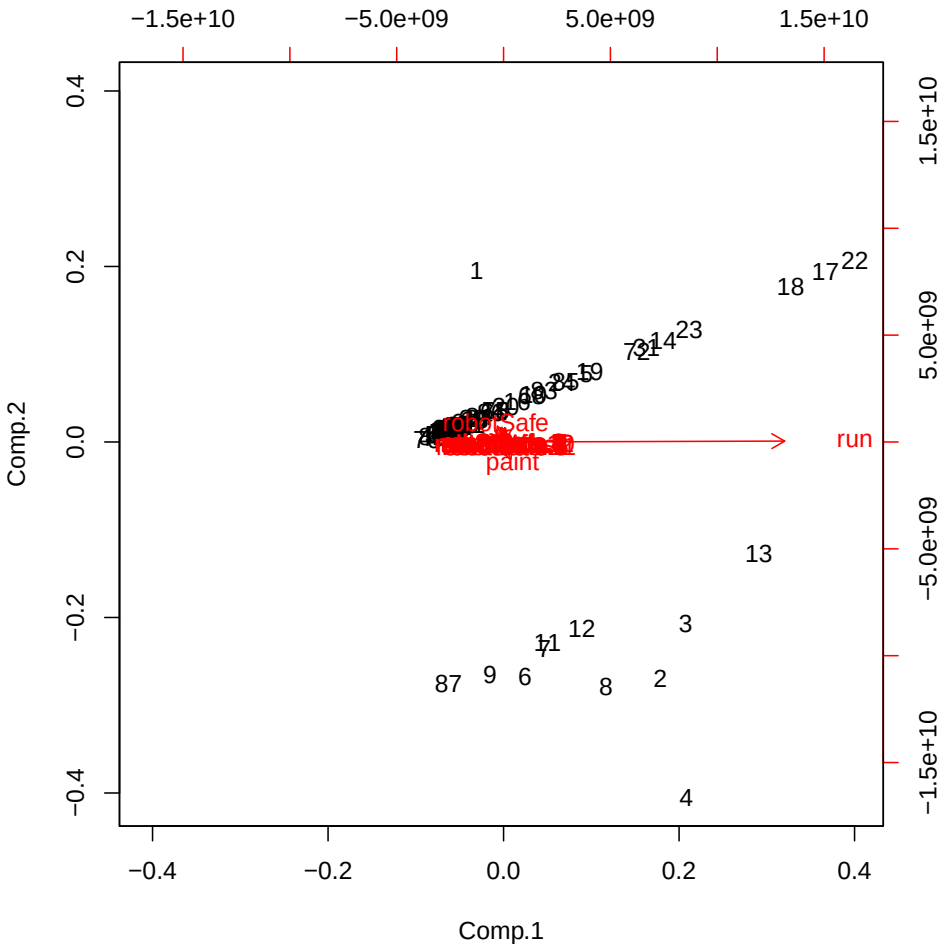


Abbildung 9.15: Biplot von Szenario 9 – Webframe.

❑ Die Laufzeit des Systems wird (fast) nur durch die Methode `run` bestimmt. Ein Blick in den Code bestätigt dies.

❑ Es gibt Methoden zur Initialisierung (`init`, `start`) die kaum Einfluss auf die Performanz in diesem Use-Case haben.

❑ Die einzigen korrelierenden Methoden sind die `robotSafe` und die `robotSafe` Methode. Die Methode `robotSafe` wird in der `run`-Methode aufgerufen um zu überprüfen, ob die entsprechende Seite untersucht („gecrawlt“) werden darf.

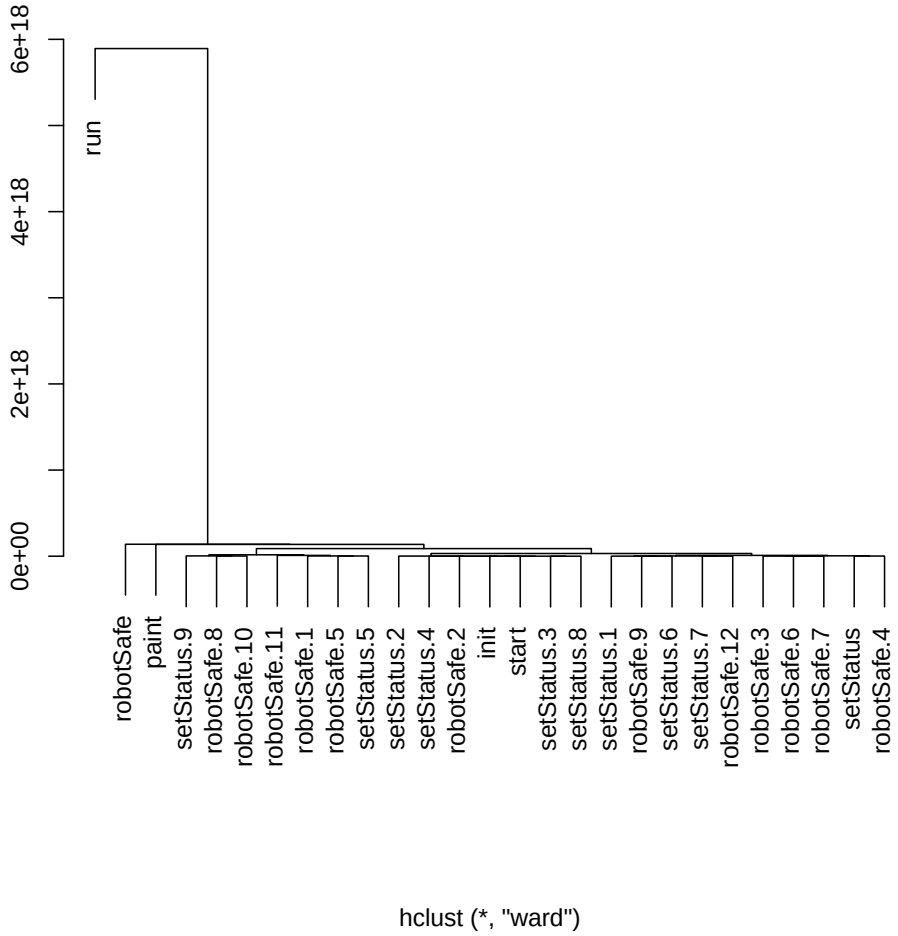


Abbildung 9.16: Das hierarchische Clustering der Kovarianzmatrix von Szenario 9.

❑ Die Methode `run` hat einen entsprechend großen Abstand zu allen anderen Methoden.

❑ Die Performanz des Webcrawlers wird von der Methode `run` bestimmt.

❑ Eine Optimierung der anderen Methoden wird sich unwesentlich auf die Performanz des Webcrawlers auswirken.

9.10 Performanzanalyse von Szenario 10 – Modelchecker cmc

Abbildung 9.17 und Abbildung 9.18 demonstrieren die Anwendung der Analysemethodologie auf Szenario 10. Mittels *Präprozessordirektiven* wurde hier ein Tracing und die Prolongation in dem *C-Programm* realisiert. Eine andere Variante dieses Modelcheckers wurde in [Mano7] einer strukturentdeckenden Performanzanalyse unterzogen. Analog zu Abschnitt 9.5 können die gewonnenen Daten in das Statistikprogramm *R* eingelesen werden (siehe Listing 9.4).

```
cmc <- read.table ("G:\\moout3b.txt", header=TRUE)
cmc.pca <- princomp (cmc)
```

Listing 9.4: R: Daten einlesen

Wie in der Vorstellung von Szenario 10 auf Seite 35 geschrieben wurde, wird beim *Modelchecking* ein Graph durchsucht. Da der Graph nicht zyklenfrei ist, wird beim CMC in einer Hashtabelle bereits besuchte Knoten bzw. Zustände der Suche gespeichert [Mano7, S. 68]. Erfahrungsgemäß benötigt die Berechnung der Hashwerte die größte Dauer im Suchalgorithmus [Mano7, S. 68]. Die gewonnen Daten geben genau die Erfahrungen von Hammer [Hamo9] bezüglich seines Modelcheckers wieder.

```
# Prepare Data: Covariance
cmc.cov <- cov (cmc)
# K-Means Clustering with 3 clusters
fit <- kmeans(cmc.cov, 3)

# vary parameters for most readable graph
library(cluster)
clusplot(cmc.cov, fit$cluster, color=TRUE, shade=TRUE, labels=3, lines=0,
    xlim=c(-5,25), main="cmc␣-␣Kmeans,␣3␣Cluster")
```

Listing 9.5: R: Durchführung und Visualisierung von *KMeans* bei Szenario 10

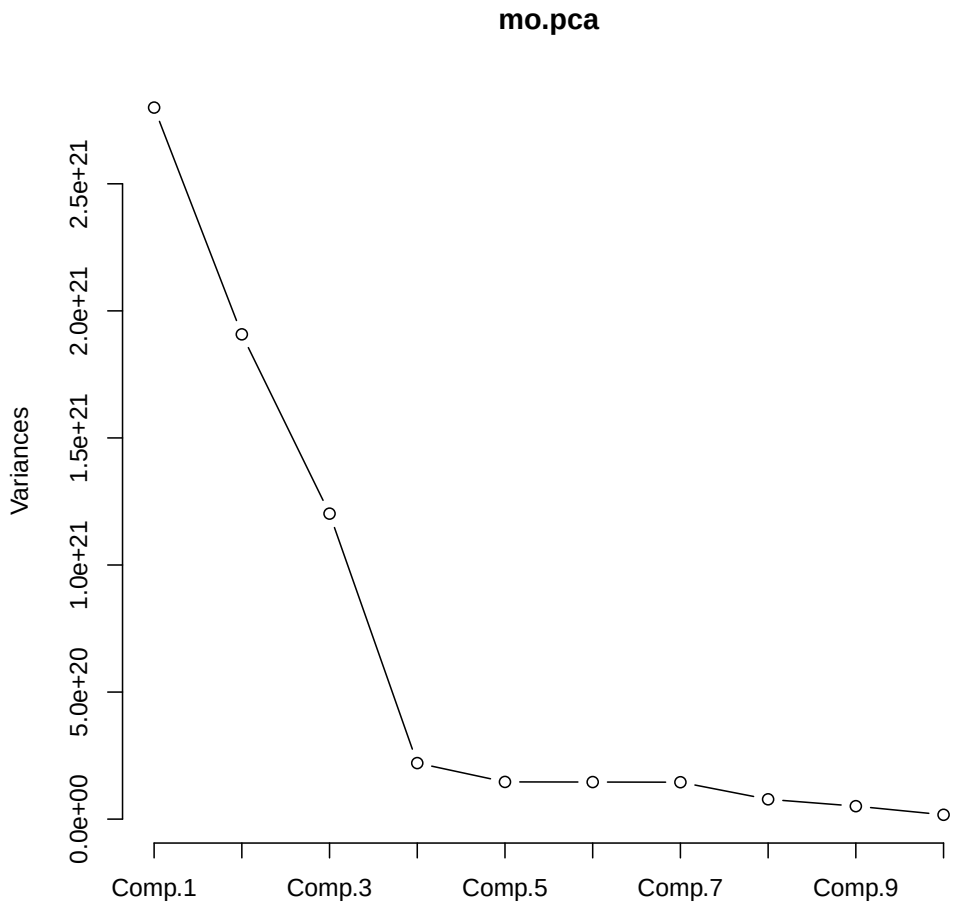


Abbildung 9.17: Screeplot zur Faktorenanalyse von Szenario 10.
❑ Der „Knick“ befindet sich zwischen der dritten und vierten Komponente.
❑ Drei Faktoren beschreiben die Varianz des expliziten Modelcheckers hinreichend genau.
Die Grafik wurde zitiert von [Mano7]

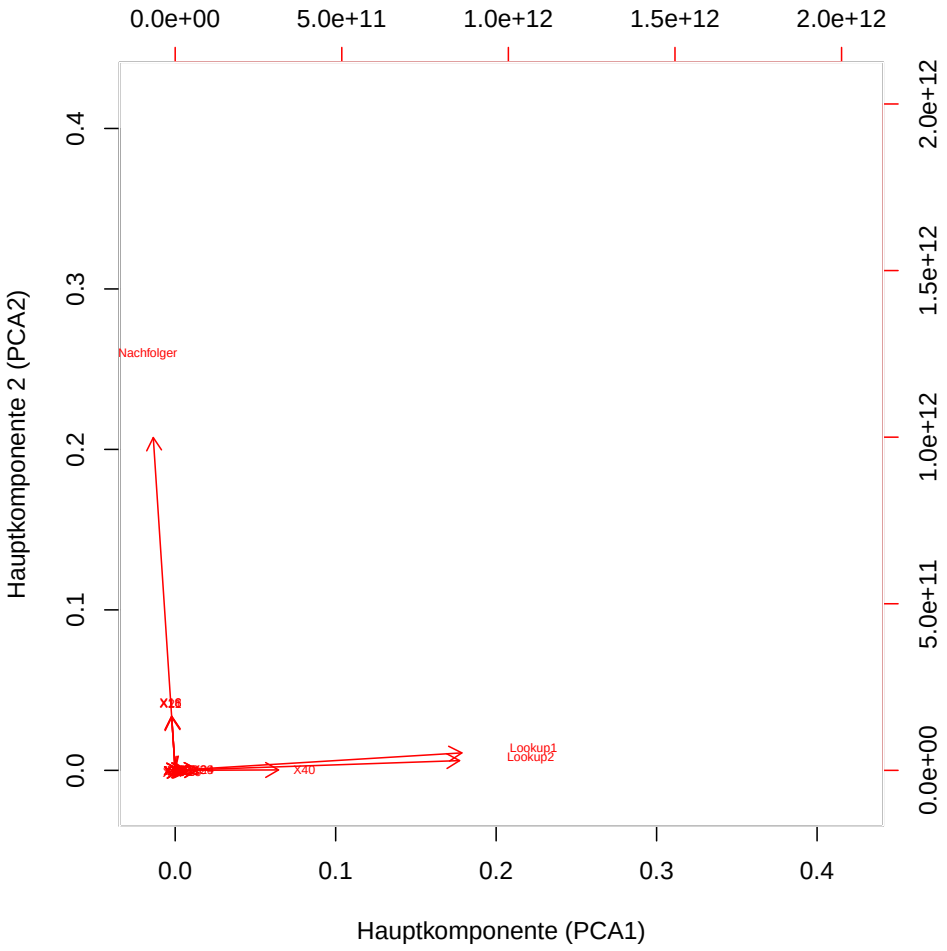


Abbildung 9.18: Biplot erstellt aus der Hauptkomponentenanalyse Szenario 10 – Modelchecker cmc.

❑ Die beiden Faktoren Nachfolger (zur Berechnung der Nachfolgezustände) und Lookup1 und Lookup2 (Lookups in der Hashtabelle) erklären den Gesamtzeitverbrauch des Modelcheckers gut.

❑ Beide Funktionen sind unabhängig, d. h. eine Optimierung der Funktion Nachfolger wird keine Verbesserung auf die Funktionen der Hashtabelle haben.

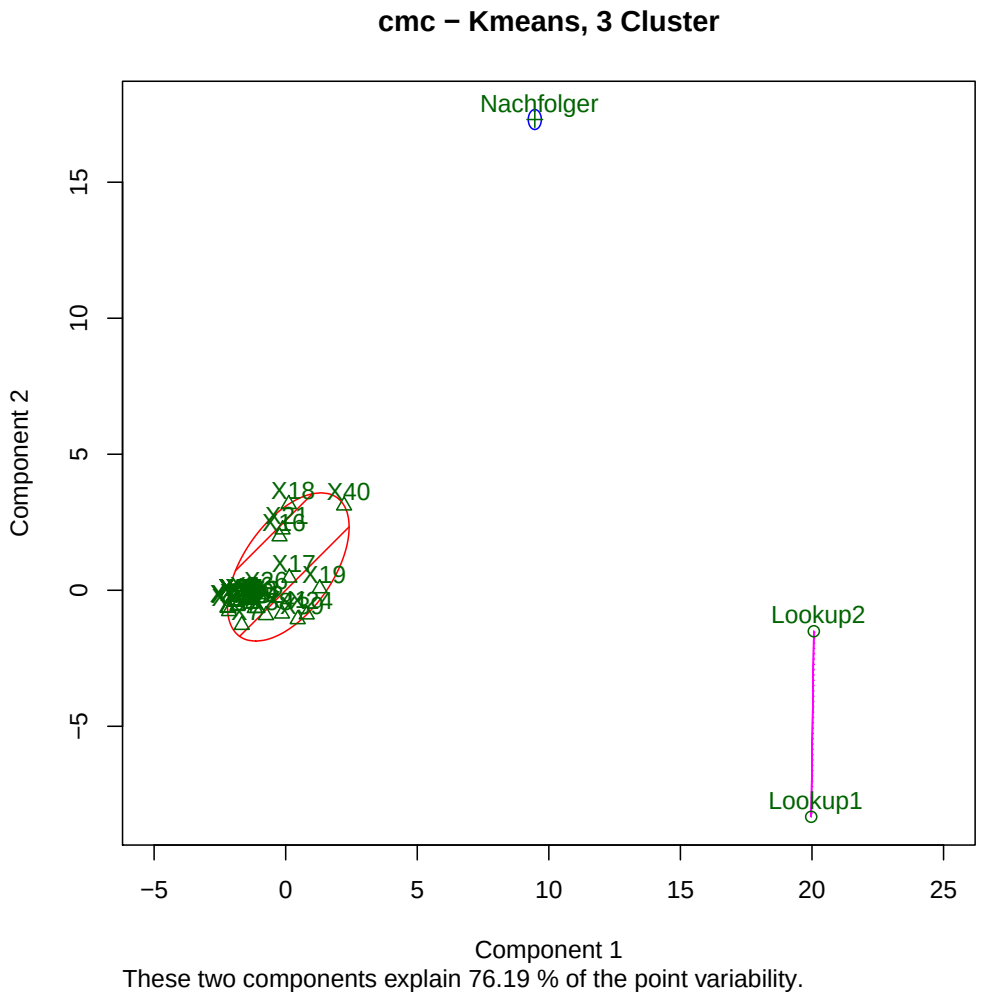


Abbildung 9.19: K-Means angewandt auf die Kovarianzmatrix von Szenario 10.

❑ Mittels des Befehls `kmean <-kmeans(cmc, 3)` wurde ein *KMeans* mit drei initialen *Clustern* durchgeführt.

❑ Anschließend wurde zur Visualiserung das Paket `cluster` (Befehl: **`library`**`(cluster)`) geladen.

❑ Mittels `clusplot` wurde das Ergebnis von KMeans visualisiert.

❑ Hier sind wieder schön die drei Cluster zu erkennen: die Berechnung der Nachfolgezustände (`Nachfolger`), die Berechnungen für die Hashfunktion (`Lookup1` und `Lookup2`) sowie der Rest der Funktionen.

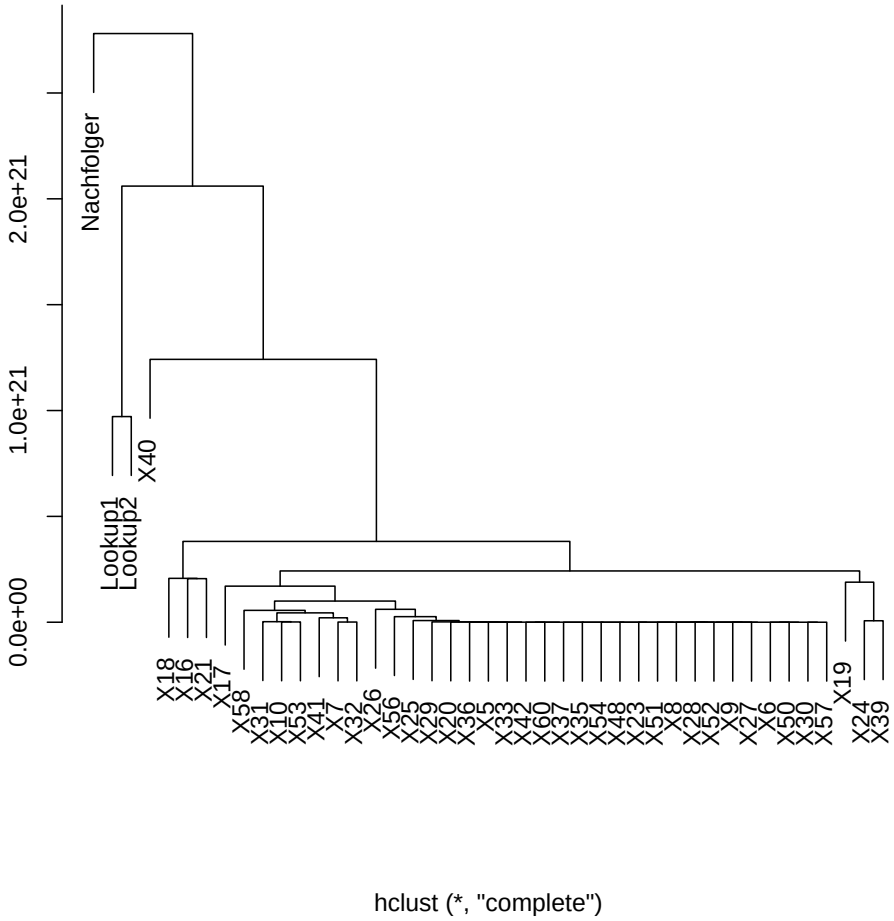


Abbildung 9.20: Hierarchisches Clustering der Kovarianzmatrix von Szenario 10.
❑ Mittels des Befehls **`cov`** wurde eine Kovarianzmatrix der Daten erzeugt.
❑ Auf diese Kovarianzmatrix wurde die Distanzfuntkion `dist` angewandt.
❑ Das Ergebnis wurde mittels `hclust` hierarchisch geclustert.
❑ Analog sind wieder die Funktionen `Nachfolger`, `Lookup1` und `Lookup2` als dominante Funktionen zu erkennen.

9.11 Performanzanalyse von Szenario 11 – Steuerungssoftware

Die Software eines industriellen Steuerungssystems, einem Produkt der Siemens AG, wurde bereits in [Mano7, S. 72] einer Performanzanalyse mit Hilfe der Faktorenanalyse unterzogen. Beschrieben ist das Szenario in dieser Arbeit auf Seite 36. Unzureichende Performanz kann in diesem System zu sicherheitskritischen Konsequenzen führen.
Alle jar-Files des in Java implementierten Systems wurden mit den entsprechenden Aspekten verwebt[4], das System wurde variiert und ausgeführt und die Laufzeiten der Module protokolliert [Mano7, S. 72]. Der entsprechende Biplot ist in Abbildung 9.11 dargestellt.
In einem iterativen Prozess wurde die hohe Anzahl der Module reduziert – für die Suche nach Optimierungskandidaten und zur Übersichtlichkeit wurden unwesentliche Module entfernt – und eine Faktorenanalyse durchgeführt, analog zu [Mano7, S. 72].

[4] siehe Abschnitt 7.1, Seite 139

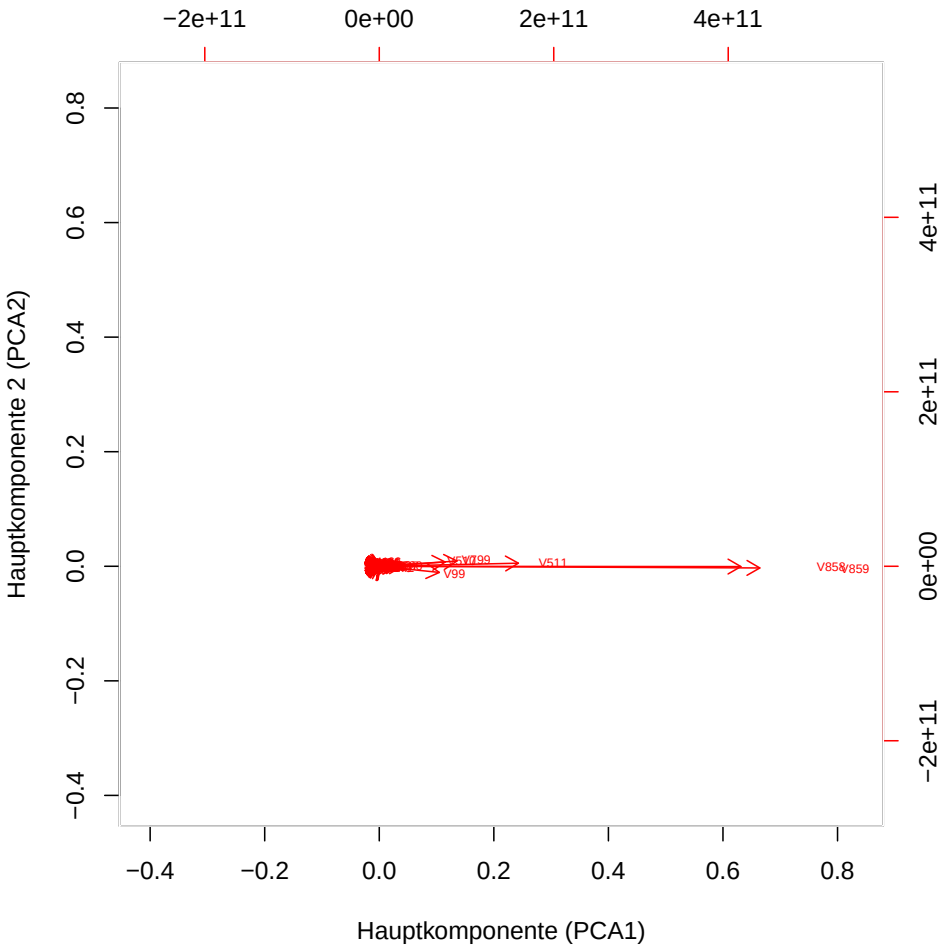


Abbildung 9.21: Biplot erstellt aus der Faktorenanalyse des beschriebenen Steuerungssystems (Szenario 11) mit allen Methodenlaufzeiten als Variablen.
❑ Die Variable V859 repräsentiert die Methode `main`, die Gesamtlaufzeit der Software.
❑ Die Variable V859 repräsentiert die Methode `BootManager`.

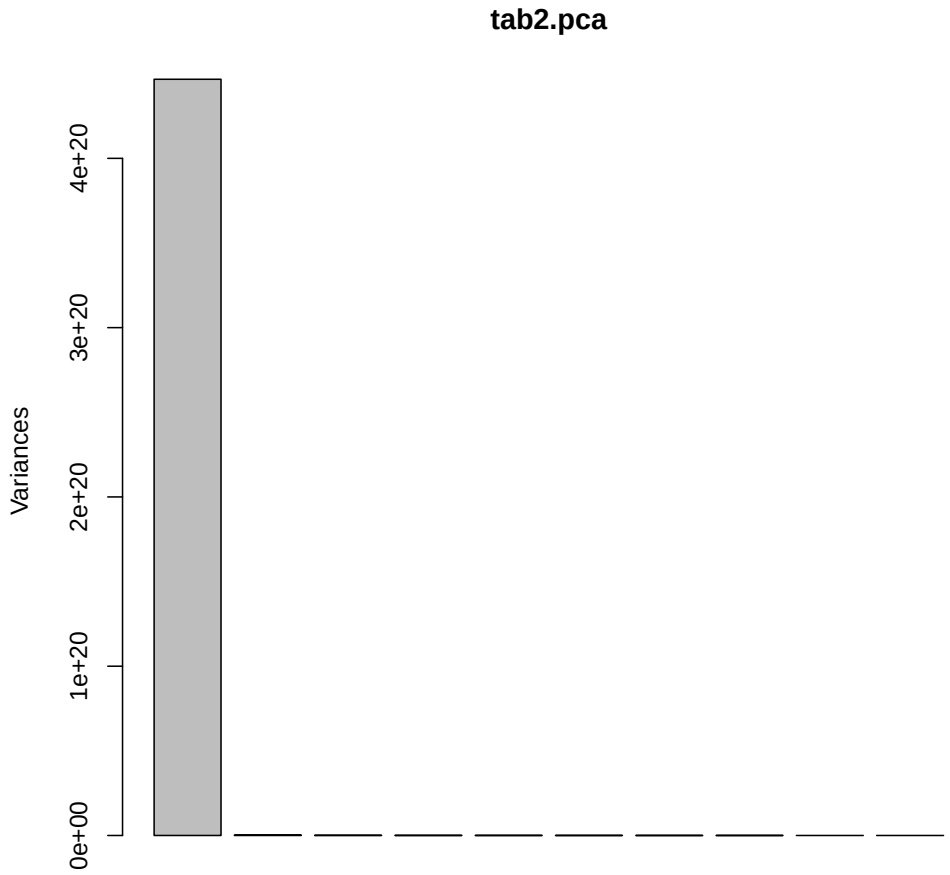


Abbildung 9.22: Screeplot erstellt aus der Faktorenanalyse des beschriebenen Steuerungssystems (Szenario 11) mit allen Methodenlaufzeiten als Variablen. Ein Faktor erklärt die Varianz hinreichend genau.

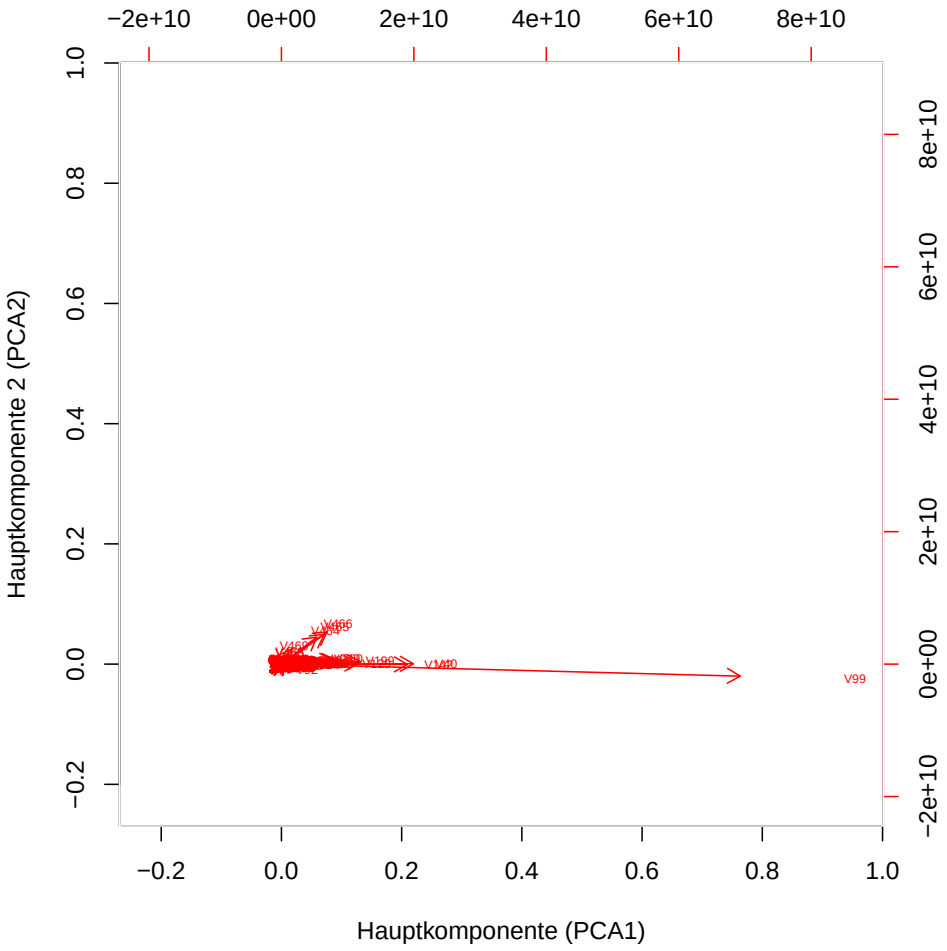


Abbildung 9.23: Biplot erstellt aus der Faktorenanalyse des beschriebenen Steuerungssystems (Szenario 11).

❑ Die Laufzeiten der `main`– und der `BootManager`–Methode wurden entfernt.

❑ Zwei Faktoren sind erkennbar, Methoden die direkt die Laufzeit von `BootManagerImpl.orderByDependency` (V99) beinflussen (3 Uhr), sowie Methoden aus der Klasse `Quicklet` (2 Uhr).

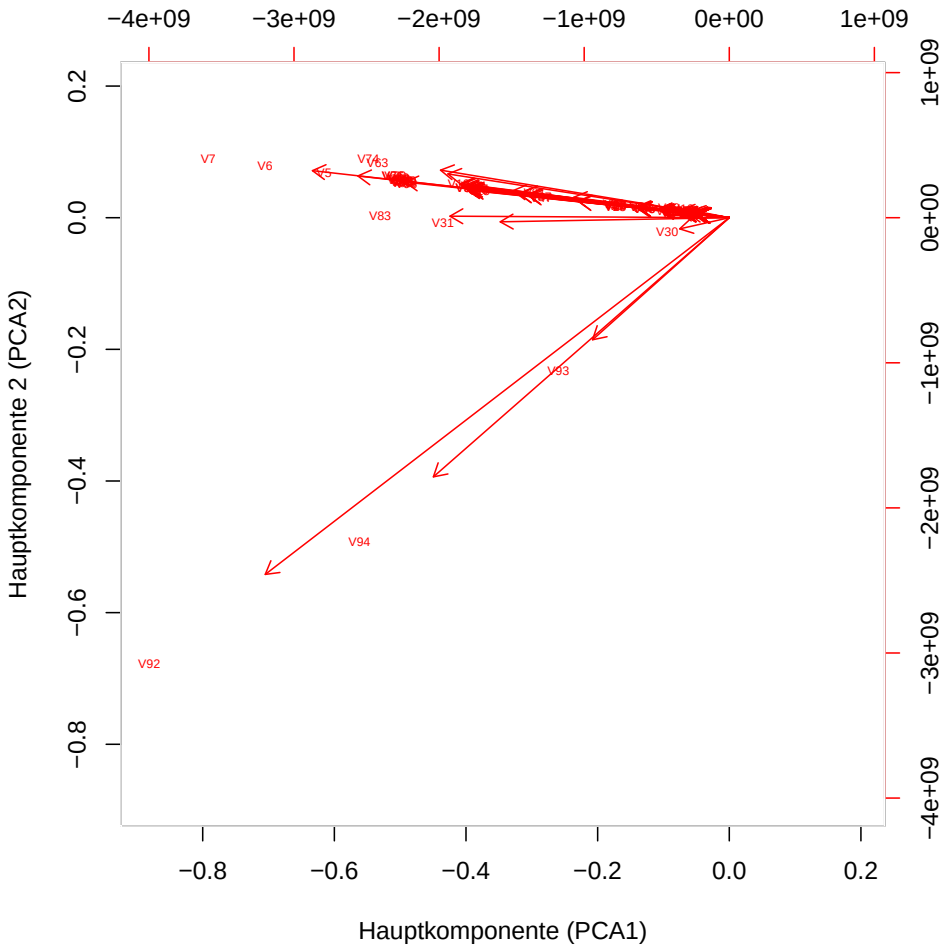


Abbildung 9.24: Biplot erstellt aus der Faktorenanalyse des beschriebenen Steuerungssystems (Szenario 11). Die Laufzeiten weiterer Methoden wurden in einem iterativen Prozess entfernt.

❑ Auf 8 Uhr sind `BootManagerImpl.findImplementation`–Methoden abgebildet, deren Performanz voneinander abhängt.

❑ Methoden aus der `LowLevelLogger`–Klasse (V7,V6,...) sind auf 10 Uhr.

❑ `Quicklet.getSpecification` (V30) ist (weitgehend) unabhängig von den Laufzeiten dieser Methoden.

9.12 *Performanzanalyse von Szenario 12 – komponentenbasierter Webcrawler*

Dieses Szenario wurde auch in [Man+08e] zu Illustrationszwecken genutzt. In Szenario 12 wurde jede Komponente sechsmal um verschiedene Faktoren prolongiert. Alle Kommunikationslinks wurden mit einem Logging-Aspekt instrumentiert, so dass die Ausführungszeiten in ein File protokolliert (getraced) werden konnten. Aus diesem Logfile wurde analog zu den vorherigen Szenarios eine Matrix der erstellt. In *R* führten wir eine *Hauptkomponentenanalyse* (ähnlich zu einer oder eine *Faktorenanalyse*) durch.

Tabelle 9.12 zeigt die einzelnen, dadurch gewonnenen Faktoren des Szenarios. Deutlich zu sehen ist, dass die Komponente `LoadDistributor` und `LinkFilter0` einen starken Zusammenhang haben (Hauptkomponente 1 – PCA1), während die anderen Komponenten nichts auf diesen Faktor hochladen.

	PC1	PC2	PC3	PC4	PC5
BracketStripper0	0,00	0,00	0,00	0,00	0,00
BracketStripper1	0,00	0,00	0,00	0,00	0,00
BracketStripper2	0,00	-0,01	0,00	0,00	0,00
BracketStripper3	0,00	-0,01	0,00	0,00	0,00
LinkFilter0	-0,65	0,02	-0,01	0,01	0,02
LoadDistributor0	-0,75	-0,02	0,00	0,00	-0,01
PageLoader0	0,08	0,01	0,02	0,00	0,02
PageLoader1	0,03	0,00	0,01	0,02	0,00
PageLoader2	-0,05	0,02	0,02	0,02	-0,01
PageLoader3	-0,04	0,01	0,02	0,01	-0,01
PageSeenDatabase0	0,00	0,00	0,00	0,00	0,00
PageSeenDatabase1	0,00	0,00	0,00	0,00	0,00
PageSeenDatabase2	0,00	0,00	0,00	0,00	0,00
PageSeenDatabase3	0,00	0,00	0,00	0,00	0,00
StripperTerminator0	0,02	-0,13	0,39	0,77	0,17
StripperTerminator1	0,02	-0,12	0,15	-0,25	-0,79
StripperTerminator2	0,03	-0,15	0,21	-0,57	0,59
StripperTerminator3	0,03	-0,13	-0,88	0,15	0,08
TokenNormalizer0	0,01	0,58	-0,01	0,00	0,03
WordDatabaseBuilder0	0,01	0,77	-0,02	0,00	0,02

Tabelle 9.6: Faktorladungen der Hauptkomponentananalyse des komponentenbasierten Webcrawlers.

Wenn die Funktion dieser beiden Komponenten betrachtet werden, wird der hinterliegende Faktor, der Zusammenhang klar: dieser Faktor beschreibt das Extrahieren der Links der von den unterschiedlichen Komponenten geladenen Webseiten.

Der zweite Faktor hat hohe Ladungen für die Komponenten `WordDatabaseBuilder0` und `TokenNormalizer0` hingegen eine negative Ladung für die `StripperTerminator` Komponenten.
Die Funktion der Komponenten `WordDatabaseBuilder0` und `TokenNormalizer0` ist, die geladene Website in Tokens zu zerlgen und eine Wort-Datenbank zu erstellen.
Die `StripperTerminator` Komponente hat zur Aufgabe, die asynchronen Tasks in die `WordDatabseBuilder0` Komponente einzureihen, bei einer Prolongation dieser Komponente werden (scheinbar) weniger Konflikte durch die Parallelität verursacht und die `WordDatabaseBuilder0` und `TokenNormalizer0` Komponenten performen besser.
Andere Faktoren können ähnlich analysiert werden, aber das Erklärungspotenzial verringert sich mit jedem Faktor.

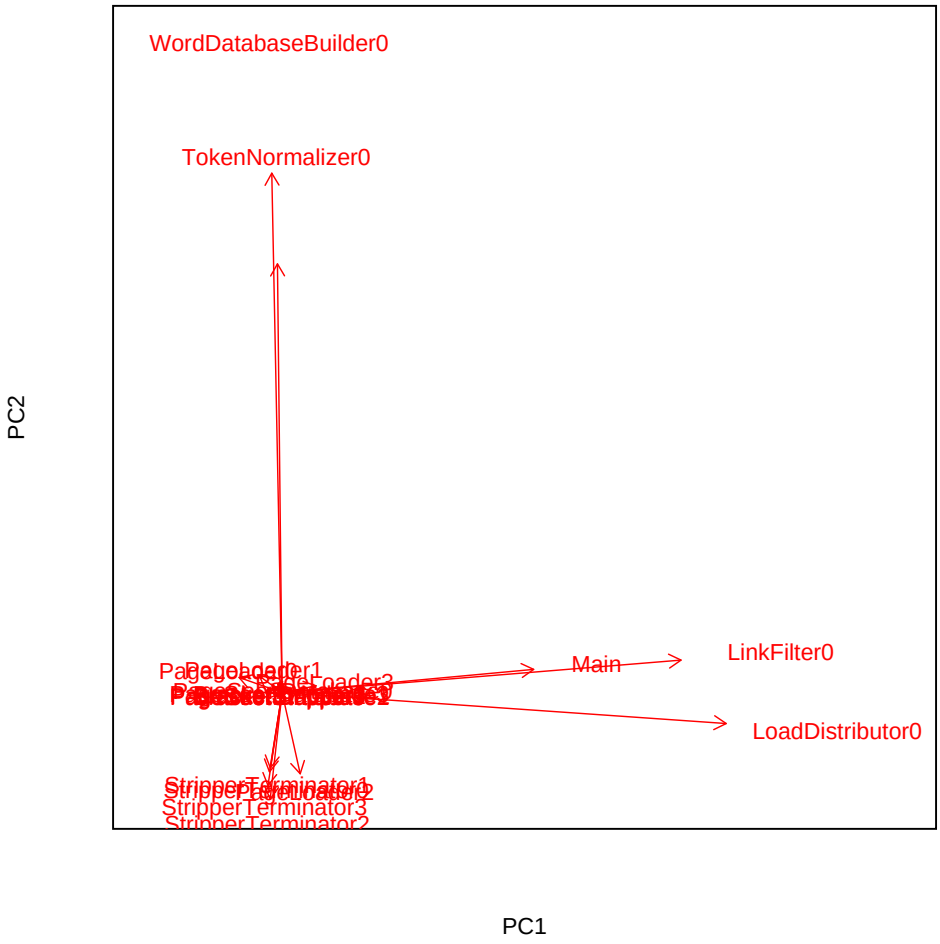


Abbildung 9.25: Biplot von Szenario 12 – kompoentenbasierter Webcrawler.

❑ Die Komponenten `LinkFilter0` und `LoadDistributor` sind fast orthogonal zu `TokenNormalizer0` und `WordDatabaseBuilder0`, was auf ein unabhängiges Verhalten hindeutet.

❑ Eine Optimierung einer dieser Komponenten aus diesem Faktor wird keine Optimierung der Komponenten in dem anderen Faktor nach sich ziehen.

9.13 Zusammenfassung und Beantwortung von Teilfragestellung ι

Status Quo

Abschnitt 9.1 beschreibt kurz aktuelle Forschungsthemen zu statistischen Analysen und zum Data Mining in der Softwareentwicklung.

Klassifizierung

Abschnitt 9.2 klassifiziert die in dieser Arbeit genutzten Datenauswertungstechniken in strukturentdeckende und modellbildende Analysemethoden sowie Simulationsansätze.

stukturentdeckende Analysementhoden

Abschnitt 9.2.1 und Abschnitt 9.3 behandeln die Klasse der stukturentdeckenden Analysementhoden. Die Idee hier ist, dass die gemessenen Daten, die extrem hochdimensionalen sind, mittels (semi-)automatischer Verfahren auf wenige, wichtige Faktoren oder Cluster reduziert werden. Abschnitt 9.3.1 führt in die multivariaten Analysemethoden ein. Nach einer kurzen Definition der benötigten statistischen Basis (Abschnitt 9.3.2) wird in Abschnitt 9.3.3 die Faktorenanalyse und die Hauptkomponentenanalyse vorgestellt. Das Beispiel in Abschnitt 9.3.4 soll die für Informatiker ungewöhnliche Faktorenanalyse explizieren.

Danach werden in Abschnitt 9.3.5 Clusteringansätze aus dem *Data Mining* und dem *Knowledge Discovery* kurz vorgestellt.

Simulationsansätze

Abschnitt 9.2.2 geht kurz auf die in dieser Arbeit entwickelten Simulationsansätze ein. Diese wurden bereits in Kapitel 8 genutzt und detailliert in Abschnitt 4.3.3 beschrieben. Abschnitt 9.7 zeigt einen Ansatz: ein schneller simuliertes Modul in einem einfachen didaktischen Szenario. In Abschnitt 10.4 wird dieser Ansatz zur schnelleren Reproduktion eines *Aging related fault*s genutzt.

Komponenten können mittels einer Prolongation langsamer simuliert werden, wie dies in Kapitel 8 zur Bestimmung der Mindestanforderungen an die Hardware demonstriert worden ist.

Modelle

Abschnitt 9.4 zeigt die in dieser Arbeit entwickelten modellbildenden Analysemethoden. Hier wird der *Resource Dependence Graph* eingeführt.

Szenarios

Abschließend wird die Performanzanalyse und -optimierung an den in Abschnitt 2.2 eingeführten Szenarios demonstriert. Der Analyseprozess wurde an zwei Illustrationsszenarios detailliert illustriert und an fünf Szenarios aus der Praxis exemplifiziert.

Illustrationsszenario Szenario 1

Im Abschnitt 9.5 wurde an einem einfachen Illustrationsszenario (Szenario 1) das Experiment und die Analyse vollständig didaktisch präsentiert: von der Datenerhebung, zur Auswertung und zur Analyse. Alle drei Klassen der Auswertung (strukturentdeckende und modellbildende Analysemethoden sowie Simulationsansätze) werden an diesem Szenario durchgeführt.

Resource Dependence Graph

Im Abschnitt 9.8 wird eine modellbildende Analysetechnik, der *Resource Dependence Graph* an Szenario 3 – dem Schiebepuzzle *Imageshuffle* – präsentiert.

Im Abschnitt 9.9 wird eine strukturentdeckende Analysetechnik an Szenario 9 – dem Java Webcrawler *Webframe* – durchgeführt. Der Webcrawler ist nebenläufig implementiert, nur die Methode `run` verrichtet die vollständige „Arbeit“. Aufgerufene Methoden

können zwar identifiziert werden, jedoch ist in diesen durch die Architektur des Webcrawlers *WebFrame* kein Optimierungspotenzial durch Abhängigkeiten vorhanden. Das volle Analysepotenzial dieser hier entwickelten Technik kann an diesem Szenario nicht vollständig demonstriert werden.

Modelchecker Im Abschnitt 9.10 wurde ein von Hammer und Weber [HW06b] implementierter expliziter Modelchecker einer Performanzanalyse mit Hilfe der Faktorenanalyse und einem Clusteringverfahren unterzogen. Die gefundenen Faktoren entsprachen den Erfahrungswerten, welche Module die Performanz des Modelcheckers bestimmen. Dieses Wissen hat meist nur der Programmierer und auch nur dann, wenn er das ganze System erstellt hat.

SBT FS 20 Im Abschnitt 9.11 wurde ein industrielles Steuerungssystem mit Hilfe der Faktorenanalyse auf Performanzeigenschaften untersucht. Auf Grund der eindeutigen Namen der Methoden konnte sehr gut auf die Ursachen für Performanzprobleme rückgeschlossen werden.

Webcrawler Im Abschnitt 9.12 wird eine strukturentdeckende Performanzanalyse an einem komponentenbasierten, nebenläufigen Webcrawler durchgeführt. Auch hier konnten Optimierungskandidaten der Anwendung sehr gut identifiziert werden.

Durch die Anzahl der Komponenten in einem System sind die gemessenen Daten enorm. Insbesondere durch die hohen Dimensionen sind die Daten schwer zu verstehen. Die Strukturen des Systems sind aus den Experimentdaten schwer ohne vorherige Datenauswertung zu deduzieren.

hochdimensional

Gruppierung Faktoren Mittels multivariater statistischer Methoden (z. B. der Hauptkomponentenanalyse oder der Faktorenanalyse, ...) und Data Mining / Knowledge Discovery Algorithmen können die Daten analysiert werden. Die Wirkzusammenhänge werden auf wenige Faktoren oder Cluster gruppiert, so können Optimierungspotenziale- und Kandidaten identifiziert werden.
Es gibt keinen generellen oder universellen Analyseansatz, die Wahl der Methode hängt vom speziellen Problem ab. In diesem Kapitel wurden einige Methoden vorgestellt. Die benötigte Interpretation erfolgt danach.

Teilfragestellung ι ✔ Somit konnte in diesem Kapitel Teilfragestellung ι beantwortet werden. Multivariate Methoden und Data Mining Algorithmen können zur Auswertung der hochdimensionalen Daten genutzt werden. Die entwickelten Simulationsansätze und modellbildenden Methoden erweitern die Analysemöglichkeiten.

Kapitel 10

Tests

„Wir können nicht beobachten, ohne das zu beobachtende Phänomen zu stören, und die Quanteneffekte, die sich am Beobachtungsmittel auswirken, führen von selbst zu einer Unbestimmtheit in dem zu beobachtenden Phänomen."

Werner Heisenberg

Übersicht zum Kapitel

Dieses Kapitel beschreibt, wie das Analyseinstrumentarium dafür genutzt werden kann, um Softwaretests durchzuführen. Abschnitt 10.1 dient zur Begriffserklärung der verwendeten Begriffe, sowie zur Problemerläuterung. Es werden unterschiedliche, in der Literatur verwendete, Fehlerklassen eingeführt, sowie die Gründe für das Aufkommen dieser Fehler erklärt.
Abschnitt 10.2 zeigt, wie ein Software auf unzreichende Synchronisation (Races) getestet werden kann.
Abschnitt 10.3 zeigt auf, wie durch asynchrone Hardware bedingte „nicht deterministisch reproduzierbare Fehler" in einem System gefunden werden können.
Abschnitt 10.4 erklärt wie *„Aging-related faults"* (Fehler die durch eine „Alterung") im System entstehen und zeigt evtl. Gegenmaßnahmen auf.

10.1 Fehlerklassen

Bezeichnungen für Fehler

Im Informatikjargon hat sich für die im Betrieb von Softwaresystemen auftretenden Fehler eine Kategorisierung mit recht plastischen Namen etabliert. In der Literatur werden für Fehler, die (meist) durch eine Interaktion mit (asynchronen) Hardwarekomponenten oder Parallelität entstehen, unter anderem die Termini „*Heisenbugs*“, „*phase of the moon bugs*“, „*Mandelbugs*“ etc., verwandt. Diese unterschiedlich bezeichneten Fehler werden nachfolgend einheitlich abstrahiert als **nicht deterministisch reproduzierbare Fehler** bezeichnet.

Zu meist wird in der Literatur grob in folgende Fehlerklassen kategorisiert:

- **Bohrbugs** sind reproduzierbare Fehler, die während des Testprozesses gefunden werden können. Gray [Gra86, S. 18] bezeichnet diese Fehler als „hart“, diese Fehler treten bei einer Wiederholung erneut auf. Das Auffinden von Bohrbugs wird nicht in dieser Arbeit betrachtet. Diese sind deterministisch reproduzierbar und werden im Testprozess mit den klassischen Methoden gefunden (siehe z. B. [Per06; Het88]).
- **Heisenbugs, phase of the moon bugs, etc.** (nicht deterministisch reproduzierbare Fehler) sind Fehler, die nur unter bestimmten Bedingungen auftreten (deshalb sind diese schlecht reproduzierbar und werden im Testprozess manchmal übersehen). Diese werden von Gray [Gra86, S. 18] als weich („soft“) bezeichnet, da sie auf seltenen Hardwarebedingungen basieren.

  Diese nicht deterministisch reproduzierbaren Fehler können noch weiter unterschieden werden:

  - **Synchronisationsfehler**

    Der Terminus „Synchronisationsfehler“ bezieht sich (nachfolgend) auf einen Softwarefehler durch fehlerhafte oder nicht ausreichende Synchronisation in der Software.
  - **Race Condition**

    Der Terminus „Race Condition“ bezieht sich auf die Möglichkeit, dass dieser Softwarefehler auftreten **kann**. Eine Race Condition kann also ein Synchronisationsfehler sein, der nicht bei jeder Systemausführung (durch nichtdeterministische Seiteneffekte) auftreten muss. Race Conditions sind nichtdeterministische und (meist) ungewünschte Effekte aufgrund von Parallelität im System.

    Netzer und Miller [NM92, S. 75] präzisieren, formalisieren und erweitern den Begriff der Race Conditions:

    - **Race Conditons**

      sind Fehler, die in Programmen auftreten und bewirken, dass diese ein nichtdeterministisches Verhalten zeigen.

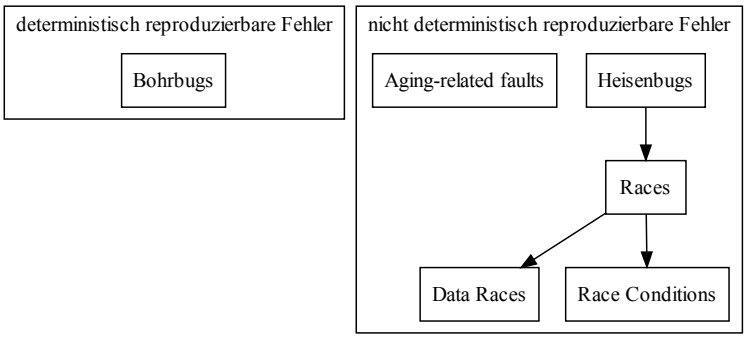


Abbildung 10.1: Schematische Darstellung der unterschiedlichen Fehlerklassen. In dieser Arbeit wird zwischen deterministisch reproduzierbaren Fehlern (Bohrbugs) und nicht deterministisch reproduzierbaren Fehlern unterschieden. Fokus dieses Kapitels sind Tests mittels des Analyseinstrumentariums auf nicht deterministisch reproduzierbare Fehler.

- **Data Races**
  sind Fehler in Programmen, bei denen gemeinsam benutzte Daten in einem kritischen verändert werden.
- **Races**
  als Überbegriff wird in dieser Arbeit, bedingt durch den aktuellen Informatikjargon, sowohl für Race Conditions und Data Races verwandt.

- **Aging-related faults** als Erweiterung wurde von Vaidyanathan und Trivedi [VT01] eingeführt. Dies sind Heisenbugs, die nach einer gewissen Zeit im System auftreten, beispielsweise durch *Speicherlecks* (*memory leaks*), Akkumulation von Rundungsfehlern, *Fragmentation* etc.

Nach Gray [Gra86] wird die industriell gefertigte Software erst nach einigen qualitätssichernden Prozessen (Reviews, Alpha-, Beta-, Gammatests) freigegeben. Sie sind damit zu meist frei von Bohrbugs, Heisenbugs können aber weiterhin bestehen. Das Programmier- und Hardwareparadigma wechselt immer mehr hin zu einer asynchronen Nutzung von Komponenten. Heisenbugs werden durch diese asynchrone Nutzung von Komponenten (Hard- und Software) verursacht [Shei03]. Eine Testmethode ist die hier vorgeschlagene Kombination aus Softwaretest und empirischem Experiment. Freigabe

Beispiel 10.1.1 Asynchrone Hardware
Die unterliegende Hardware des Systems wird immer mehr von Parallelität geprägt, die die nicht deterministisch reproduzierbaren Fehler verursachen. Die Entwicklung hin zu parallelen Hardwarearchitekturen hat mehrere Gründe, u.a.:

- **Multikern-Rechnerarchitekturen**
  Ein überwiegendes Ziel bei der Herstellung von Prozessoren ist es, diese performanter und leistungsfähiger zu entwickeln. Mehrkernprozessoren sind eine logische Weiterentwicklung, die Produktion und der Erwerb der Mehrkernprozessoren ist proportional zur erhaltenen Leistung günstiger. Für speziell auf Multikern entwickelte Software ergibt sich ein Leistungsgewinn.
- **Spezialkomponenten**
  Spezialhardware, wie beispielsweise Grafikkarten, haben eigene, spezialisierte Prozessoren integriert, die Spezialaufgaben übernehmen, wie beispielsweise die grafische Darstellung. Alle diese Komponenten werden zur Geschwindigkeitssteigerung asynchron bedient.

nicht deterministische Hardware

Die unterliegende Hardware des Systems ist zu meist nicht deterministisch. Computersysteme arbeiten auf der Basis einer Systemuhr, asynchrone Komponenten haben jedoch zu meist eigene Taktgeber. Des Weiteren gibt es Hardwarekomponenten die nicht getaktet sind. Dies erhöht die Anzahl der möglichen Zustände im System immens, bei ungetakteten Komponenten impliziert dies überabzählbar viele Zustände des Systems (siehe Beispiel 10.1.2 und Beispiel 10.1.3).

Beispiel 10.1.2 Variable Suchzeiten (Seek Time) und unterschiedliche Speichermedien
Festplatten haben variable Suchzeiten, die sich sehr unterscheiden können. Auch wird Software auf Systemen ausgeführt, die sich von ihrer Hardware sehr von den Entwicklungs- und Testsystemen unterscheiden. Ein weiterer wichtiger Aspekt ist, dass Aufwärtskompatibilität zumeist gewünscht ist, d. h. insbesondere weil sich die Hardwareeigenschaften verändern und verbessern werden. Sind die einzelnen parallelen Abarbeitungsfäden nicht sauber synchronisiert können hier Fehler auftreten, die nie während der Entwicklung oder dem Testen entdeckt wurden.

Greift die Software nun auf Daten des Speichermediums zu, kann das unterschiedliche Zeiten in Anspruch nehmen. Beispielsweise kann eine Seite bereits im Cache eingelesen sein oder das Speichermedium durch einen anderen Prozess nicht zur Verfügung stehen. Die unterschiedlichen Zustände erschweren ein Debugging extrem bzw. ermöglichen keine vollständige Testfallabdeckung des Systems.

Beispiel 10.1.3 Verlust von Netzwerkpaketen
Pakete können im Netzwerkbetrieb verloren gehen. Durch diese nicht deterministische Gegebenheit können Komponenten lange Zeit brauchen und es können Synchronisationsfehler auftreten, die möglicherweise in der Testphase eines Systems nicht gefunden wurden. Tests können also schwerlich auf alle Gegebenheiten des Systems testen, eine vollständige Testfallüberdeckung ist nicht möglich.

parallele Systeme

Zur Effizienzsteigerung und aufgrund (neuer) Rechnerarchitekturen müssen Softwaresysteme nun hinsichtlich paralleler Abarbeitung konstruiert und implementiert werden. Manche Systeme wie beispielsweise Serversysteme basieren in großem Maße auf dieser Parallelität. Die daraus resultierenden vielgestaltigen parallelen Interaktionen in der Software sind schwer zu überblicken. Zusätzlich zu dieser schwer überschaubaren Komplexität ist die unterliegende Hardware sowie der Scheduler des Systems zu meist selbst nicht deterministisch [HK10].

Exkurs 10.1.1 Implementierung paralleler Systeme

Die Programmabläufe in der Software werden immer mehr, anstelle von sequentiell (synchron), parallelisiert implementiert, d. h. logisch und strukturell zusammenhängende Anweisungsblöcke sollen parallel und damit asynchron abgearbeitet werden. Dieses hat mehrere Gründe:

- **Effizienzsteigerung durch bessere Auslastung eines Rechnersystems**
  Während das Softwaresystem (hier kann sich Softwaresystem auch auf das Betriebssystem, das Multitaskingaufgaben erfüllt, und entsprechende Anwendungssoftware beziehen) auf die Eingaben eines Benutzers warten muss, können „im Hintergrund“ notwendige Berechnungen, Dateizugriffe oder ähnliches durchgeführt werden. Bei einer rein sequentiellen Ausführung ist dies nicht möglich. Durch dieses parallele Ausführen steht das gesamte Softwaresystem schneller wieder für eine Benutzerinteraktion bereit, das System „scheint“ performanter zu sein. Die Perspektive hier ist die Performanzsteigerung eines einzelnen Programms.
- **Allgemeine Multitaskingsysteme**
  Multitaskingsysteme sind durch das oben angeführte Argument der besseren Benutzerinteraktion erwünscht. Diese Systeme müssen auch in der Software entsprechend auf Multitasking, also parallele Abarbeitung von zusammenhängenden Anweisungen, konstruiert und implementiert werden. Die Perspektive hier ist die Performanzsteigerung eines Systems mit evtl. vielen gleichzeitig ausgeführten Programmen.

Exkurs 10.1.2 Single- und Multitasking Betriebssysteme

Singletasking	Multitasking	
	kooperativ	preemptiv
DOS [KS07]	Windows 3.1 [Sim01, S. 13]	Windows 98
Palm OS [Goo+05, S. 241]	Windows 3.11 [Sim01, S. 13]	Windows NT [Sim01, S. 12]
		Windows Vista
		Windows XP
		Windows 7
		Mac OS X Leopard
		Unix [Sim01, S. 12]
		Linux

- **Singletasking:**
  Nach dem Start wird die Kontrolle vom Betriebssystem an das Programm weitergegeben. Das Programm behält die Steuerung (im Normalfall) bis zum Programmende und gibt die Kontrolle an das Betriebssystem zurück.
- **kooperatives Multitasking**
  oder auch non-preemptives Multitasking: Hier koordinierten die Prozesse, und nicht das Betriebssystem, „kooperativ" wie lange sie den Prozessor als Ressource nutzen wollen.
- **präemptives Multitasking:**
  Hier koordiniert das Betriebssystem die Nutzung des Prozessors als Ressource für die einzelnen Prozesse. Die einzelnen Prozesse können unterbrochen werden (siehe dazu Exkurs 7.2.1 auf Seite 163).

Validierung mittels des Analyseinstrumentariums

Asynchroner Betrieb bei Hard- und Softwarekomponenten ist gewünscht, dieser kann jedoch zu Fehlern führen. Das Analyseinstrumentariums kann als Instrument zur Validierung (siehe Kapitel 6) genutzt werden. Hierfür wurden zwei Anwendungsfälle mit unterschiedlichen Perspektiven ausgearbeitet:

- **Abschnitt 10.2 – Unzureichende Synchronisation in der Software:**
  Aus der Perspektive der Software (die in einem System ausgeführt wird) wird ein Verfahren vorgeschlagen mit der ein Testen auf Synchronisationsfehlern beziehungsweise ein Testen von Systemen bezüglich Races[1] – ermöglicht werden kann.
- **Abschnitt 10.3 Nicht deterministisch reproduzierbare Fehler im System:**
  Es wird eine Methode zur Validierung von Systemen hinsichtlich der Fehler die typischerweise mit asynchron arbeitender Hardware (z. B. Hardware mit eigener Taktung[2]) entstehen, vorgeschlagen. Mit dieser Methode können schwer identifizierbare, weil nicht deterministisch reproduzierbare Fehler, die auf Grund von Asynchronität von Hardwarekomponenten entstehen [She03] (Heisenbugs, phase of the moon bugs, ...) oder anderer Effekte (z. B. Aging-related faults) gefunden werden.

10.2 Test auf Races

Validierungsmethoden

Es besteht Bedarf an asynchronen Softwarekomponenten, ausreichende Validierungsmethoden sind jedoch noch nicht vorhanden. Nach der Untersuchung von Javacode von David Hovemeyer und William Pugh [HP04] sind Synchronisationsfehler eher die Regel als die Ausnahme. Quantitative Aussagen über ihr Vorkommen und verursachte Kosten wurden in dieser Arbeit nicht erhoben.

[1] siehe Abschnitt 10.2, Seite 258
[2] siehe Abschnitt 10.1.1, Seite 256

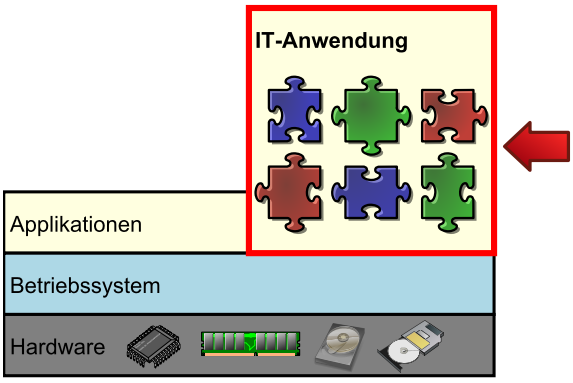


Abbildung 10.2: Test auf Races. Die variierten und validierten Ebenen dieses Abschnitts sind im System sind durch den roten Pfeil gekennzeichnet.

10.2.1 Lösungsansätze für Synchronisationsfehler

Prinzipiell gibt es mehrere Methoden um das Problem der Synchronisationsfehler anzugehen.

10.2.1.1 Vermeiden von Synchronisationsfehlern

spezielle Dialekte

Zustandsunabhängige parallele und deadlockfreie Dialekte einiger Programmiersprachen wurden vorgeschlagen, wie beispielsweise von Bacon, Strom und Tarafdar [BST00] und Boyapati, Lee und Rinard [BLR02], sind jedoch nach Hovemeyer und Pugh [HP04] nicht generell akzeptiert und deshalb nicht verbreitet.

10.2.1.2 Statische Analyse

false positives

unvollständige Abdeckung

Es gibt eine Klasse von Programmierwerkzeugen (beispielsweise die Tools „*FindBug*“ [Aye+08] und „*RacerX*“ [EA03]), die statischen nach Mustern im Code sucht. Neben der Fehleranfälligkeit für „*false positives*“ (es werden Fehler gefunden die überhaupt keine sind) werden aber zu meist nicht alle Synchronisationsfehler entdeckt. Ayewah u. a. [Aye+08] bemerken, dass mit dem Tool „*FindBug*“ nicht alle möglichen Synchronisationsfehler abgedeckt werden können.
Daneben hat dieser Ansatz der statischen Analyse noch folgende, entscheidende Nachteile:

- Der Code muss als ganzes offen vorliegen. Softwareentwicklung großer Projekte basiert jedoch auf Wiederverwendung von bestehenden Codebauteilen wie Bibliotheken, oft auch von *third party vendors* (Drittanbietern), die ihren Code und ihre Bibliotheken nicht offenlegen bzw. offenlegen wollen.
- Konflikte im Zusammenhang mit Betriebssystemroutinen können nicht mit analysiert werden.
- Das aktuelle Verhalten auf einem realen oder potentiell möglichen System kann nicht untersucht werden. So können unnötige Synchronisationsmechanismen vorliegen, die das System unnötig verlangsamen oder Synchronisationsprobleme durch die Hardware entstehen.
- Große Softwareprojekte sind oft in unterschiedlichen Programmiersprachen implementiert, was eine statische Analyse erschwert.

In [HP04] werden weitere statische Analysesysteme im Abschnitt Related Work aufgezeigt. Alle Systeme arbeiten mit einer statischen Analyse.

10.2.1.3 Modelchecking

formales Modell

Beim Modelchecking wird vollautomatisch eine Systembeschreibung gegen ein formales Modell, das als Spezifikation dient, geprüft [Sto02]. Mit dem Modelchecking können Synchronisationsfehler gefunden werden. Zusätzliche Nachteile zu den in Abschnitt 10.2.1.2 aufgezählten sind jedoch:

- Das System muss als Formel und als Systembeschreibung vorliegen, was im industriellen Umfeld zu meist nicht der Fall ist.
- Der Zustandsraum wächst überproportional mit der „Größe“ des Systems. So sind nur kleinere Systeme/Programme mittels Modelchchecking verifizierbar, unter anderem wegen der Speichergröße und dem extrem langem Zeitbedarf.

10.2.1.4 Dynamische Analyse durch Ausführen potentiell möglicher Threadinterleavings

Scheduler

In [Sto02] wird ein Verfahren verwandt, mit der durch eine spezielle Scheduling-Funktion das Programm getestet wird. Diese Scheduling-Funktion kann einen Kontextwechsel des Threads verursachen. Mittels spezieller Heuristiken wird versucht, Synchronisationsfehler zu entdecken.

CHESS

Eine aktuelle Weiterentwicklung ist das Tool *CHESS* von *Microsoft* [Mus+08; EM10]. Nach den Angaben von Microsoft kann dieses nicht deterministisch reproduzierbare Fehler zielgerichtet finden. Hier wird die Reihenfolge der Abarbeitung durch einen

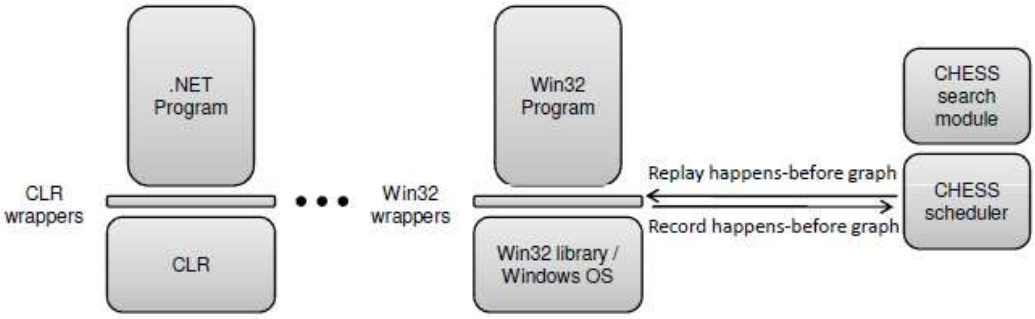


Abbildung 10.3: Die *CHESS* Architektur. Ein Wrapper sitzt zwischen dem zu analysierenden Programm und der für die Parallelität zuständigen *API* (z. B. der Windows API). Entsprechende API-Aufrufe werden protokolliert, das *search module* errechnet andere mögliche *Interleavings*, diese werden durch den *CHESS scheduler* entsprechend repliziert. – Die Abbildung wurde zitiert von [Mus+08].

Wrapper protokolliert und, basierend auf dem *Happened-Before Graphen* von Leslie Lamport [Lam78] (siehe auch Abschnitt 6.5.1 auf Seite 125), mittels des eigenen Schedulers in unterschiedlichen Läufen anders reproduziert. [Mus+08; EM10]

Weitere Werkzeuge, die auch mittels des Schedulers die Reihenfolge der Abarbeitung ändern sind *ConTest* [Ede+01] und *ConTest-lite* [Ede+03] von *IBM*. ConTest

Dieser Lösungsansatz ist im industriellen Umfeld eher anwendbar als die obig präsentierten Ansätze, die Vorteile und Unterschiede zu dem in dieser Arbeit entwickelten Lösungsansatz werden in Abschnitt 10.5 detailliert diskutiert.

10.2.2 Lösungsansätze für Data Races

Auch für den Unterfall der Data Races gibt es einige Werkzeuge zur Analyse. Yu, Rodeheffer und Chen [YRC05] präsentiert mit RaceTrack ein Tool aus einer Klasse von Werkzeugen (beispielsweise auch die Anwendung von Choi u. a. [Cho+02]), mit dem Data Races gefunden werden können. Hierbei wird (auch) eine virtuelle Maschine instrumentiert. Die Aktionen des Programms werden protokolliert und potentielle Data Races werden referiert. RaceTrack

Dieser Lösungsansatz benutzt eine virtuelle Maschine als Analyseinstrumentarium, deren zeitliche Abarbeitungseigenschaften aber nicht zur Analyse verändert wurden.

10.2.3 Testdurchführung

Testvorgehen

Beim Testen wird wie folgt vorgegangen: einzelne Ausführungsstränge bzw. Module des Systems werden gezielt prolongiert, die absolute Ausführungslänge einer Operation wird gezielt verändert bzw. retardiert (der Ausführungszeitpunkt wird verändert). Dafür werden einzelne Aufrufe der Software instrumentiert und hier das Retardieren bzw. Prolongieren in geeigneter Weise realisiert. Das System wird ausgeführt, der zusätzlich instrumentierte Testcode überprüft, ob in jeder möglichen angegebenen Kombination aus Zeitdauer und Zeitpunkt jede Zusicherung gehalten werden kann.

Der Entwickler/Tester gibt vor, welches Modul zum Testen um welche Intervalle verlängert/verlangsamt oder retardiert wird. Dieses Verlängern/Verlangsamen/Retardieren kann sukzessive geschehen, beispielsweise um 5, 10, 20 Millisekunden (ms) bzw. prozentual anhand der Gesamtzeit der Modulausführung. Spezielle Vorgehensweisen wie beispielsweise *simulated annealing* [LA87] oder heuristische Methoden (vgl. [Mus+08]) sind des Weiteren sinnvoll, insbesondere wenn der Test automatisch durchgeführt wird.

Heuristiken

Kombination

Es ist durchaus sinnvoll eine Kombination von Modulen zu prolongieren. Zweckmäßig ist es beispielsweise, wenn der Entwickler für jedes Modul einen Vektor angibt, der beschreibt um welches Intervall oder um wie viel Prozent das Modul prolongiert oder retardiert werden soll. Dies ergibt dann für einen speziellen Testfall einen Vektor von Vektoren, die das Verhalten der Module bzw. des Systems beschreiben. Diese Vektoren und Matrizen können automatisiert erstellt und variiert werden.

Vektor

Die Testdurchführung wird an Szenario 6 demonstriert.

Validierung

Mit speziellem Testcode, wie beispielsweise einem bestehenden xUnit Testframework, wird das Ergebnis der Prolongation (Verlängerung/Verlangsamung) bzw. Retardation überprüft. Gelingt der Test für alle Intervalle, ist die Wahrscheinlichkeit für einen Fehler eher gering, aber nicht ausgeschlossen (der Synchronisationsfehler könnte noch zwischen den Intervallen oder außerhalb der Testintervalle auftreten). Tritt ein Fehler auf, muss dieses Modul einem Reengineering zugeführt werden um die Fehlerquelle zu beseitigen.

10.2.4 Testdurchführung an Szenario 6 – Simpler Data Race

Zur Durchführung des Test wurde folgender Aspekt, der auf das wesentliche gekürzt in Listing 10.1 wiedergegeben wird, implementiert und in das Szenario 6 verwebt:

```
import org.junit.Test;
import static org.junit.Assert.*;

public aspect ProlongSzenarioSimplestRace {

  static long Prolong = 0;
  public static String threadToProlong = "";

  pointcut Prolong_count(): set(public int *.count);
  pointcut Prolong_run(): execution(* *.run());

  Object around(): Prolong_count(){
    if (Thread.currentThread().getName().equals(threadToProlong)) {
      long dt = System.currentTimeMillis();

      while ((System.currentTimeMillis() - dt ) < Prolong)
        ;
    }
    return proceed();
  }

  Object around(): Prolong_run(){

    long dt = System.currentTimeMillis();

    while ((System.currentTimeMillis() - dt ) * 100< Prolong)
      ;

    return proceed();
  }

  public static void main(String[] args) {

    Prolong =

    SzenarioSimplestRace.main(args);
    testDataRace();
  }

  @Test
  public static void testDataRace() {
    assertEquals("Count", SzenarioSimplestRace.NUMBEROFTHREADS
        * SzenarioSimplestRace.INCREMENTS, SzenarioSimplestRace.count);

  }

}
```

Listing 10.1: AspectJ, Junit: Race Condition Checker für Szenario 6 – Simpler Data Race

Der Pointcut `pointcut Prolong_run(): set(public int *.count);` retardiert den Start eines Threads. Somit kann auch eine Fehlervermeidung durch ein „Sequentialisieren“ der Threads stattfinden.
Der Pointcut `pointcut Prolong_count(): set(public int *.count);` prolongiert den schreibenden Zugriff auf die Variable `count`.

```
@Test
public static void testDataRace() {
  assertEquals("Count", SzenarioSimplestRace.NUMBEROFTHREADS
      * SzenarioSimplestRace.INCREMENTS, SzenarioSimplestRace.count);

}
```

Listing 10.2: JUnit: Test für Szenario 6 – Simpler Data Race

`assertEquals` in der Methode `testDataRace()` überprüft, ob ein *Data Race* stattgefunden hat. Hierbei wird auf den zu erwartenden Wert (`SzenarioSimplestRace.NUMBEROFTHREADS*SzenarioSimplestRace.INCREMENT`) getestet. Der Test-Case befindet sich in Listing 10.2.
In Abbildung 10.2.4 (Seite 265) und in Abbildung 10.2.4 befinden sich zwei Diagramme, die zeigen, wie sich die Werte von `count` durch die Prolongation, ohne Test (`assertEquals`), verändern würden.

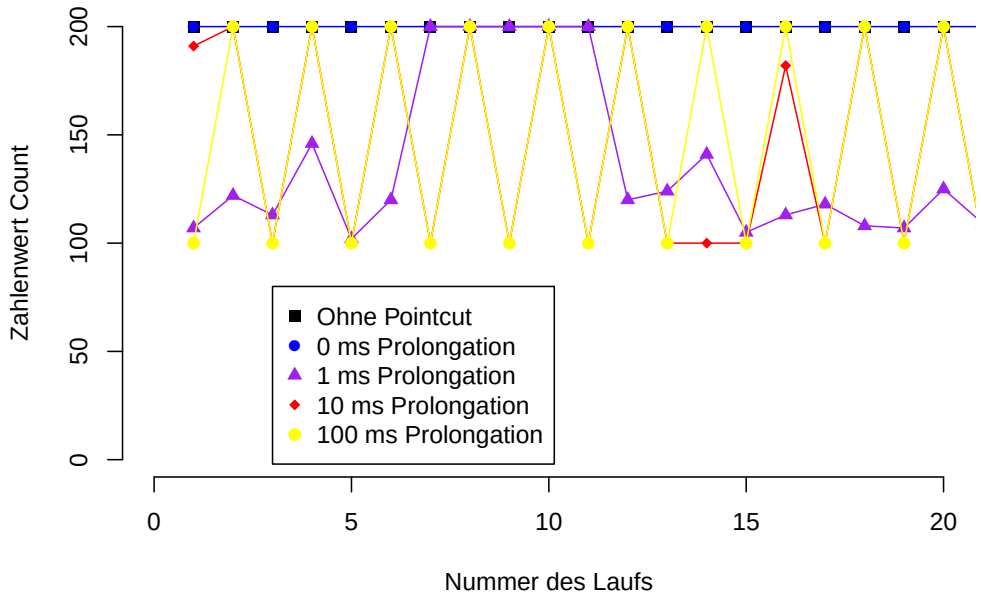


Abbildung 10.4: Szenario 6 mit zwei gestarteten Threads (`NUMBEROFTHREADS` = 2) die jeweils bis hundert (`count` = 100) einen gemeinsamen Zahlenwert inkrementieren. Sequentialisiert würde am Ende des Programmes der Wert 200 (`NUMBEROFTHREADS` * `count`) in der gemeinsamen Variablen stehen. Die Abbildung zeigt die Ergebnisse der Zahlenwerte von 20 Testläufen. Mit einer Prolongation ergibt sich folgendes Bild:

❑ Keine Prolongation (schwarz): Es kommt kein Data Race vor.
❑ 0 Sekunden Prolongation (blau): Deckungsgleich mit der schwarzen Linie. Kein Data Race wird entdeckt.
❑ 1 ms Prolongation (purpur): Hier treten zum ersten Mal die Data Races auf. Der Wert Count ist nach dem Programmlauf nicht immer gleich 20.
❑ 10 ms Prolongation (rot) sowie
❑ 100 ms Prolongation (gelb): hier treten alternierend die Zahlenwerte 100 oder 200 auf. Wurde der zuerst gestartete Thread verlängert, treten durch den zweiten Thread fast nur Data Races auf, was zu dem Wert 100 führt. Wird der zweite Thread verlängert führt dies zu einer Sequentialisierung, was zum Datenwert 200 führt.
Inkorrekterweise wurden zur besseren Darstellung die Punkte durch Linien verbunden.

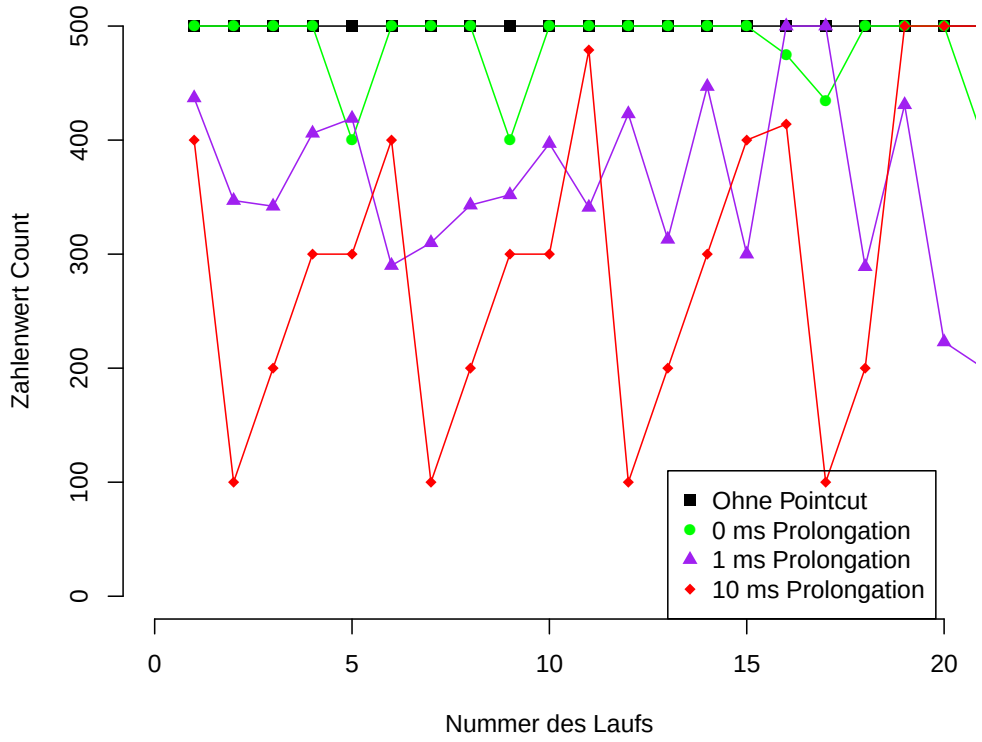


Abbildung 10.5: Szenario 6 mit fünf gestarteten Threads (`NUMBEROFTHREADS = 5`) die jeweils bis zehn (`count = 10`) einen gemeinsamen Zahlenwert inkrementieren. Sequentialisiert würde am Ende des Programmes der Wert 500 (`NUMBEROFTHREADS * count`) in der gemeinsamen Variablen stehen. Die Abbildung zeigt die Ergebnisse der Zahlenwerte von 20 Testläufen. Mit einer Prolongation ergibt sich folgendes Bild:

❑ Keine Prolongation (schwarz): Es tritt kein Data Race auf.
❑ 0 Sekunden Prolongation (grün): In drei Programmläufen kann ein Data Race entdeckt werden. Auch wenn die Blockierung der Prolongation nur 0 Sekunden in Anspruch nimmt, wird das Ergebnis beeinflusst. Der Fehler ist jedoch noch nicht deterministisch reproduzierbar.
❑ 1 ms Prolongation (purpur): Hier treten Data Races auf. Der Wert von `Count` hat ein chaotisches Verhalten.
❑ 10 ms Prolongation (rot): Mit ziemlicher Sicherheit tritt beim Programmlauf ein Data Race auf.
Inkorrekterweise wurden zur besseren Darstellung die Punkte durch Linien verbunden.

10.3 *Test auf nicht deterministisch reproduzierbare Fehler*

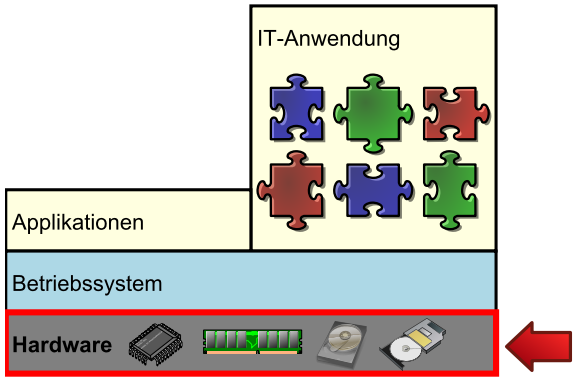


Abbildung 10.6: Test auf nicht deterministisch reproduzierbare Fehler im System. Die variierte Ebene in diesem Abschnitt sind im System durch den roten Pfeil gekennzeichnet.

10.3.1 *Einführung*

Exkurs 10.3.1 Therac-25

Therac-25

Die Therac-25 war ein Linearbeschleuniger zur Bestrahlung von Karzinomen (Krebs) und ist ein berüchtigtes, klassisches (Lehr-)Beispiel für Softwarefehler. Zwischen 1985 und 1987 kam es zu massiven Überdosen [Kle94]. In späteren Untersuchungen wurden „Race Conditions“ (Synchronisationsfehler) als Ursache für die Fehlfunktionen identifiziert [LT93].

Texas-Bug

In Texas kam es im März 1986 zu einer Strahlenverbrennung mit tödlichem Ausgang. Grund hierfür war eine erfahrene Benutzerin, die Daten zu schnell eingab, was nie während der Tests der Therac-25 der Fall war und deshalb nicht entdeckt wurde. [LT93, S. 27]

Die angedachte Funktionsweise der Therac-25 kam nur Zustande wenn gewisse zeitlichen Rahmenbeschränkungen eingehalten wurden. Wurden diese unterschritten kam es zu den Fehlern bei der Bestrahlung. In Tests des Linearbeschleunigers wurden diese wegen der mangelnden Erfahrung bei der Eingabe nie festgestellt und waren später schwierig zu reproduzieren.

klassische PC-Perspektive

Die Testmethode wird mittels der „klassischen“ Perspektive des Systems (beispielsweise das System als normaler PC) beschrieben. Prinzipiell kann diese Testmethode auch auf andere Systeme mit asynchronen Komponenten (z. B. andere Systeme als Systeme von Systemen, asynchrone arbeitende Hardwarekomponenten, etc.) angewandt werden, insbesondere scheint dies bei verteilten Systemen sehr vielversprechend.

Heisenbugs

Bohrbugs sind Fehler in den Aktionen des Programms die auf Grund von Eingabewerten entstehen (die „klassischen“ Fehler), deshalb reproduzierbar sind und beim Testen gefunden werden können.[3] Heisenbugs sind nach Gray [Gra86] jedoch Fehler, die aufgrund von Annahmen des Programms über seine Laufzeitumgebung und der Interaktion der Subkomponenten entstehen.

10.3.2 Stand der Wissenschaft und Technik

fehlertoleranter Router

In der Arbeit „*Building Bug-Tolerant Routers with Virtualization*“ wird ein fehler-toleranter Router vorgestellt, der multiple Kopien der Router Software parallel ausführt lässt [CR08]. Argumentation hier ist, dass jede Kopie sich von der anderen unterscheidet (in Speicherlayout, unterschiedliche Ordnung/Timing von Updates, unterschiedliche Code-Basis, etc. . .), so dass es unwahrscheinlich ist, dass diese zur gleichen Zeit ausfallen [CR08, S. 54]. Der korrekte „Output“ des Routers wird durch einen „voter“, der als Teil des Hypervisors realisiert ist, bestimmt [CR08, S. 54]. Jedoch wird in der Arbeit erwähnt, dass manche Ansätze die Berechnungszeit erhöhen könnten, was die Autoren als unerwünscht ansehen [CR08, S. 54]. Die Arbeit schlägt eine Methode vor durch Virtualisierung den Router fehlerresistenter zu implementieren [CR08]. Im Gegensatz dazu wird, in der hier vorgestellten Methode, gerade die Ausführungsdauer und Ausführungszeitpunkt variiert. Das Analyseinstrumentarium wird benutzt um Fehler zu provozieren.

Langzeittests

Das System kann hinsichtlich einer möglichen, fehlerverursachenden Interaktion der asynchronen Komponenten bzw. externen Faktoren und Aging-related faults bislang nicht ausreichend validiert werden. Auf Systemebene (Interaktion von Hardware und Software) wurden solche Tests bislang nicht ausreichend erforscht, auf (niederer) Hardwareebene sind die einzig in der Literatur vorgeschlagenen Möglichkeiten zyklische Tests und Langzeittests - das System wird ausreichend lange getestet und dadurch wird gehofft, Fehler zu finden [ALF00].

Darstellung an Szenario 7

Die Lösung wird in Abschnitt 10.3.4 an Szenario 7 vorgestellt. Hier werden die Performanzeigenschaften einzelner Hardwarekomponenten zielgerichtet variiert und auf nicht deterministisch reproduzierbare Fehler getestet. Dies ist mittels der Virtualisierungslösung möglich. Der Vorschlag zur Validierung von Software besteht aus dieser speziellen Virtualisierungslösung und eines Testframeworks (siehe dazu Abschnitt 7.3.11 auf Seite 177).

[3] siehe Abschnitt 10.1, Seite 254

10.3.3 Testdurchführungs

Variieren

Der Entwickler/Tester gibt vor, welche Hardwarekomponente zum Testen um welche Intervalle verlängert/verlangsamt oder retardiert wird. Dieses Verlängern/Verlangsamen/Retardieren kann sukzessive geschehen, beispielsweise um 5, 10, 20 Millisekunden (ms) bzw. prozentual anhand der Gesamtzeit. Spezielle Vorgehensweisen aus der Informatik wie Optimierungsverfahren (beispielsweise simulated annealing [LA87]) oder heuristische Methoden und Approximationsalgorithmen sind des Weiteren sinnvoll.

Kombination

Für weitere Tests ist es sinnvoll eine Kombination von Hardwarekomponenten zu prolongieren. Das Prolongieren selbst übernimmt das vorgeschlagene Testframework mit der speziell konstruierten virtuellen Maschine um so dem Entwickler bzw. Tester die Arbeit zu erleichtern und den Test zu automatisieren.

Vektor

Sinnvoll ist es beispielsweise, wenn der Entwickler für jede Hardwarekomponente einen Vektor angibt, der beschreibt um welches Intervall oder um wie viel Prozent die Hardwarekomponente prolongiert oder retardiert werden soll. Dies ergibt dann für einen speziellen Testfall einen Vektor von Vektoren, die das Verhalten der Hardwarekomponenten beschreiben.

Ausführungsverkürzung

Mit einer Erweiterung um das simulierte Optimieren[4] sind des Weiteren auch durch die Prolongation Verkürzungen der Ausführungszeit möglich. Somit hätte beispielsweise der *Texas-Bug* der *Therac-25* (siehe dazu den Exkurs 10.3.1 auf der Seite 267) während eines Tests gefunden werden können.

Testcode

Mit speziellem Testcode, wie beispielsweise einem bestehenden xUnit Testframework [Meso7], wird das Ergebnis der Prolongation (Verlängerung/Verlangsamung) bzw. Retardation überprüft. Gelingt der Test für alle Intervalle, ist die Wahrscheinlichkeit für einen Fehler eher gering, aber nicht ausgeschlossen (der nicht deterministisch reproduzierbare Fehler könnte noch genau zwischen den Intervallen oder außerhalb der Testintervalle auftreten). Tritt ein Fehler auf, muss das System einem Reengineering zugeführt werden um die Fehlerquelle zu beseitigen.

Testdurchführung

Die zu verändernde Performanzeigenschaften der Hardwarekomponenten (Zeitpunkt, Zeitdauer) können als Vektor von Vektoren dargestellt werden. Nach der Testausführung folgt eine *Assertion* [Flo67], hier wird validiert, ob die Operation mit der prolongierten/retardierten asynchronen Komponente noch das richtige Ergebnis liefert.

Vollständigkeit

In einem deterministischem System können mit dieser Methode Synchronisationsfehler gefunden werden, die durch Verzögerungen in dem prolongierten Zeitraum auftreten könnten.

Langzeittests

Bei hochgradig nichtdeterministischen Systemen sollte die Prolongation/Retardation im Verbund mit Langzeittests verwandt werden. Durch den Nichtdeterminismus sind die Ablaufbedingen nicht immer gleich, ein Fehler muss nicht bei einer Systemausführung

[4] siehe Abschnitt 4.3.3, Seite 67

auftreten. Die Prolongation/Retardation stellt aber ein mächtiges Werkzeug (sozusagen eine zweite Dimension bzw. ein optimales Beeinflussen der Langzeittests) dar und verbessert den Wirkungsgrad der Langzeittests.

Fehler durch eine Prolongation

Durch die Prolongation/Retardation können Fehler auftreten, die ohne diese nicht aufgetreten wären. Dies ist jedoch genau der Sinn dieser Methodik. Ergeben sich bei der Prolongation/Retardation Fehler, können diese Fehler auch auf anderen Hardwareumgebungen auftreten und diese Fehler gilt es zu vermeiden. Das System muss also einem Reengineering zugeführt werden.

Zustandsexplosion

Je nach zu testendem System muss eventuell eine Vielzahl von asynchroner Hardware mitgetestet werden, bei verteilten Systemen kann die Anzahl der Komponenten sehr groß werden was zu einer Zustandsexplosion führen kann.

Gridcomputing

Hier kann zur Testdurchführung evtl. Gridcomputing [CG08; BFH03] verwandt werden, in Zukunft können Konzepte wie Quantencomputing [Her00] zur Durchführung des Tests interessant werden.

10.3.4 Testdurchführung an Szenario 7 – Heisenbug

Variieren von Performanzeigenschaften

Beim Testen werden zusätzlich die Performanzeigenschaften der Hardware (hier dem des Peripheriegerät, welches *Memory-Mapped* ist) zielgerichtet variiert.

Szenario 7

Angenommen, Daten werden von der Prozedur `main` in die Speicherzelle, welche *Memory-Mapped* ist, geschrieben (siehe Szenario 7, Seite 31) Die Operation überprüft nicht mittels *Polled I/O* ob die Speicherzelle schon zur Verfügung steht.

veralteter Wert

Der aktuelle Wert wird **nicht** auf das Peripheriegerät übertragen, das I/O Register des Peripheriegerätes hat einen veralteten Wert gespeichert. Dieser Fehler kann insbesondere durch eine andere Umgebung (weitere Prozesse) oder eine andere Hardwareumgebung mit anderen Performanzeigenschaften zu Stande kommen. Die Situation stellt einen typischen, nicht reproduzierbaren Fehler dar.

Test

Beim Testen wird wie folgt vorgegangen: das System wird mit möglichen Hardwareeigenschaften (langsamere Hardware, unterschiedliche Zugriffszeiten) getestet. Die virtuelle Maschine selber führt das System aus, eigener Testcode (beispielsweise ein *xUnit* Modul) überprüft ob in jeder möglichen Umgebung jede Zusicherung gehalten werden kann. Hierdurch kann der Fehler von Szenario 7 zielgerichtet gefunden werden.

Erweiterung

Somit kann die Testmethodologie grundsätzlich erweitert werden. Systeme sind anfällig für nicht deterministisch reproduzierbare Fehler, da diese Fehler nur schwer im Testprozess entdeckt werden. Die Gefahr besteht, dass diese Fehler trotz ausreichender Tests nicht gefunden werden, da sich Zielsysteme von den Test- und Entwicklungssystemen unterscheiden, auf dem Zielsystem jedoch durch eine andere Konfiguration auftreten können. Insbesondere die große Vielfalt der Rechnersysteme und unterschiedlichen Performanzeigenschaften stellt eine hohe Gefahr für Synchronisationsfehler dar.

So kann es vorkommen, dass trotz ausreichender Tests diese nicht deterministisch reproduzierbaren Fehler nicht entdeckt werden, weil die Voraussetzungen auf dem Testsystem völlig unterschiedlich zu den Voraussetzungen auf den Zielsystemen sind. unentdeckte Fehler

Dies spiegelt die Hardwarebedingungen an den eigentlichen Zielsystemen besser wider. Durch die hohe Hardwareheterogenität ist eine bessere Testfallüberdeckung gewährleistet: so treten nicht deterministischen reproduzierbare Fehler bedingt durch unterschiedliche Performanzeigenschaften der Hardware auf. Hardwareheterogenität

Auf diese Art und Weise kann mit Situationen umgegangen werden, die im Praxisfall zwar auftreten (nicht deterministisch reproduzierbare Fehler bedingt durch unzureichende Sicherheitsmechanismen bei der Programmierung) aber schwer oder überhaupt nicht nachgestellt werden können. Ein Analyst, Tester oder Softwareengineer gewinnt wertvolle Informationen über mögliche Fehler. Teilweise scheint dies auch die einzige Möglichkeit nicht deterministische Fehler bei externen Komponenten oder in sehr großen Projekten/Systemen aufzufinden. Reproduktion bestimmter Fehler

Respektive weil Situationen getestet werden können, die bislang vom Testen ausgeschlossen waren, ergeben sich unter anderem die Vorteile, dass eine komplettere Testfallüberdeckung sowie und eine höhere Sicherheit durch das Testen realisiert wird.

Im Besonderen ermöglicht sich durch die Virtualisierung der Testumgebung eine Automatisierung des Testens für Hardwarekomponenten und Tests unterschiedlichster Systeme. Tests unterschiedlicher Systeme

10.4 Aging-related faults

Vaidyanathan und Trivedi [VT01] erweitern die klassische Fehlerkategorisierung um „Aging-related faults“, also Fehler bedingt durch eine Alterung des Systems.

Exkurs 10.4.1 Fallstudien für Aging-related faults

Grottke, Matias und Trivedi [GMT08] präsentiern drei Fallstudien für Aging-related faults.

- **Cisco Netzwerk Switches:** Bei einer Reihe von Cisco Netzwerk Switches (Catalyst 2900, Catalyst 4000, Catalyst 5000, und Catalyst 6000) trat ein Aging-related fault auf. Durch ein Speicherleck eines telnet Prozesses verringerte sich der zur Verfügung stehende Speicher graduell, bis es schließlich zu einer Fehlfunktion des Switches kam, so dass der Switch keine anderen Prozesse ausführen konnte [Cis06].
- **Apache Web Server:** Grottke, Matias und Trivedi [GMT08] bringen ein Standardbeispiel für die graduelle Degradierung der Performance und Qualität eines Programms mit typischen langen Laufzeiten. Die schlechte Performanz des Beispiels basierte auf graduell wachsenden Auslagerungsspeicher (Swap-Speicher) durch Kindsprozesse und Prozesse mit geringer Priorität, die noch nicht zur Ausführung kamen. [GMT08]
- **Patriot Missile System:** Ein internes Integer-Register der Patriot Missile zählte in Zehntelsekunden die vergangene Zeit. Bei einem georteten Ziel musste das System diese Zeit in eine Gleitkommazahl (real) konvertieren. Dies verursachte einen Konversionsfehler, der

Ursache	Betriebssystem	Applikation
Speicherleck	allozierten Speicher	
keine Wiederfreigabe	Freigabe Filehandler Freigabe Sockets	
nicht (vollständige) Terminierung	Threads Prozesse	
Fragmentation	phys. Speicher Filesystem	Datenbankfiles
Fehler-Akkumulation	Rundungsfehler Datenverschmutzung	

Tabelle 10.1: Gründe für Aging-related faults. Die Tabelle ist angelehnt an [GMT08, S. 3].

> Wert war in der Höhe zu der bereits vergangenen Zeit proportional unpräzise. Nach bereits 8 Stunden konnte eine SCUD Rakete nicht mehr geortet werden. [GMT08; Bal+10] Am 25. Februar 1991 konnte eine Patriot System in Dhahran, Saudi Arabien, nach mehr als 20 Stunden Laufzeit ohne Re-Boot, eine SCUD Rakete nicht abfangen, was zu 28 Toten und 97 Verletzten führte [Bal+10].

Klassischerweise wird hier in der Literatur eine „*Software rejuvenation*“ vorgeschlagen.

36 Software rejuvenation
Software rejuvenation is the concept of gracefully terminaring an application and immediately restating it at a clean internal state. [KF95]

So sollen inkonsistente Zustände des Systems durch einen Neutstart vermieden werden. Gross, Bhardwaj und Bickford [GBB02] schlagen ein proaktives, frühzeitiges, multivariates Erkennen von Aging-related faults vor, um eine *Software rejuvenation* durchzuführen. Die Arbeit von Garg u. a. [Gar+98] schlägt eine Metrik aufgrund von Messdaten in einem realen System bezüglich Aging-related faults, für eine subsequente *Software rejuvenation*, vor. Der in dieser Arbeit präsentierte Ansatz ermöglicht jedoch **einen Test** eines gealterten Systems in kürzerer *wall clock time*, was einen enormen Vorteil darstellt. Zum Testen des Systems hinsichtlich von Aging-related faults werden die Eigenschaften der entsprechenden Hardwarekomponenten so variiert, dass die Hardwarekomponente die entsprechende „Alterung“ widerspiegelt (beispielsweise längere Zugriffszeiten der Festplatte durch eine Fragementierung).

Aging-related faults

Aging-related faults

10.4.1 Testdurchführung an Szenario 8 – Patriot Missile

Das Szenario wurde in Abschnitt 8 auf Seite 32 eingeführt und beschrieben.
Eine entsprechende Transferpartiton wird, wie in Abschnitt C.7 beschrieben, angelegt. Transferpartiton
QEMU^{dt} wurde wie in Abschnitt 7.3.10 um eine entsprechende Zeitrafferfunktion erweitert.
Die virtuelle Maschine wird mit *muLinux* beispielsweise direkt von dem heruntergeladenen Image gestartet.

```
qemu -hda mu.img -cdrom mulinux-14r0.iso -boot d -fda /root/floppy
```

Listing 10.3: Shell: Download und Start von muLinux

Im virtualisierten muLinux wird das Laufwerk gemountet.

```
guest:~ mount /dev/fd0 /a
```

Listing 10.4: Gast Shell: Mount einer Partition

Nachdem in das Verzeichnis `\a` gewechselt wurde, wird Listing 2.5 kompiliert und ausgeführt.

```
guest:~ gcc -o patriot PATRIOT.C
guest:~ ./patriot
```

Listing 10.5: Gast Shell: Kompilierung und Start

Abbildung 10.7 zeigt das Szenario, ausgeführt auf QEMU^{dt} mit einem *Time Warp* mit Time Warp
dem Faktor 50. Die spezielle Virtualisierungslösung QEMU^{dt} mit Zeitraffer konnte den Fehler des Programms innerhalb eines Bruchteils der Zeit nachstellen. Leider zeigt das verwendete muLinux größere Zeitungenauigkeiten. Hier wird jedoch genau ein Aging-related fault sichtbar, die Ungenauigkeiten des virtualisierten Betriebssystems (muLinux) gegenüber der ungenauen Implementierung von Szenario 8.

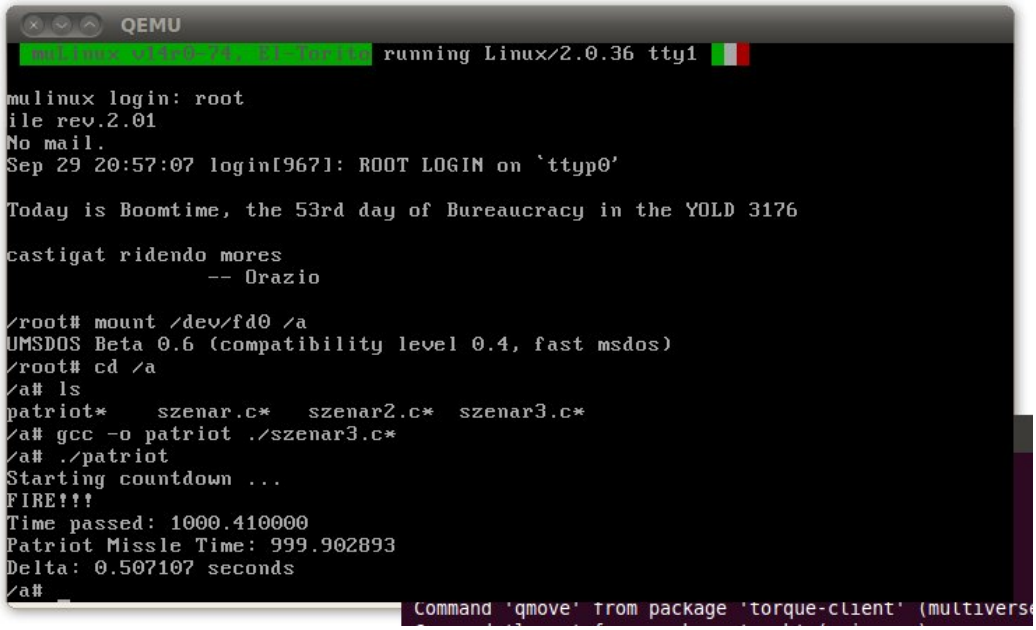


Abbildung 10.7: Testdurchführung von Szenario 8, angelehnt an den *Aging-related fault* bei der *Patriot Missilie*.

❑ Der Deltawert, der Fehlerwert, ist entspricht bei diesen 1000 Sekunden mehr als eine halbe Sekunde.

❑ Dieses Szenario mit mehr als 16 Minuten wurde durch den Time Warp zirka 50 mal so schnell – in ca. einer Sekunde – ausgeführt. ❑ Der Aging-related fault basiert auf der Ungenauigkeiten der Implementierung von Szenario 8 gegenüber der Uhr des virtualisierten Betriebssystems (muLinux).

10.5 Zusammenfassung, Beantwortung von Teilfragestellung κ und Ausblick

Abschnitt 10.1 erklärt die Fehlerklassen in Systemen. Nicht deterministisch reproduzierbare Fehler treten nicht bei jedem Programmlauf auf. In der Literatur haben sich für diese, unter bestimmten Gegebenheiten auftretenden Fehler, recht plastische Namen etabliert, wie „*Heisenbugs*“, „*phase of the moon bugs*“, „*Mandelbugs*“ und viele mehr [Gra86]. Sie sind schwer nachzustellen, im Gegensatz zu den sogenannten *Bohrbugs* [Gra86].

Nicht deterministisch reproduzierbare Fehler

Generell kann man diese „*Heisenbugs*“ bzw. „*Mandelbugs*“ noch weiter untergliedern, beispielsweise in „*Races*“, als Überbegriff für „*Data Races*“ und „*Race Conditions*“, als Instanzen von Heisenbugs in der obersten Schicht im System, der Software (bzw. Anwendungsschicht). Dies sind (meistens) unerwünschte, nicht-deterministische Seiteneffekte durch unzureichende Synchronisationsmechanismen im Code.

Fehlerklassifizierung

Abschnitt 10.2 zeigt bestehende Ansätze zum Testen auf *Race Conditions*. Es wird in diesem Abschnitt erklärt, wie durch das Analyseinstrumentarium ein Test auf *Race Conditions* erfolgen kann. An Szenario 6 wird dies mittels Instrumentierung an einem simplen *Data Race* exemplifiziert.

Races

Abschnitt 10.3 erklärt die Gründe für „nicht deterministisch reproduzierbare Fehler“ in einem System, zumeist bedingt durch asynchrone Hardwarekomponenten. Es wird der Stand der Technik und der Wissenschaft zum Testen und Reproduzieren dieser Fehler präsentiert. An Szenario 7 wird anhand eines Szenarios das Reproduzieren solcher Fehler mittels des Analyseinstrumentariums illustriert.

nicht deterministisch reproduzierbare Fehler

Abschnitt 10.4 erklärt wie „*Aging-related faults*“, Fehler die durch eine „Alterung“ im System entstehen und zeigt evtl. Gegenmaßnahmen auf. An Szenario 8, angelehnt an einen Softwarefehler der *Patriot Missile*, wird das praktikable Reproduzieren (in testbarer Zeit) dieses Fehlers gezeigt.

Aging-Related faults

Somit konnte Teilfragestellung κ beantwortet werden.

Teilfragestellung κ ✔

Das Analyseinstrumentarium kann bei parallelen Systemen Fehler verursachen (siehe Kapitel 6). Diese Fehler, bedingt durch unzureichende Synchronisationsmechanismen, können jedoch auch auf anderen Zielsystemen auftreten. Durch die zeitliche Beeinflussung des Analyseinstrumentariums können Fehler reproduziert werden, die nicht immer im System auftreten und deshalb oft nicht während des Testprozesses gefunden werden. Insbesondere kann eine „Alterung“ emuliert bzw. in kürzerer *wall clock time* reproduziert werden, was einen praktisch durchführbaren Test des Systems auf „Aging-related faults“ zulässt.

Alterung

Das Analyseinstrumentarium kann als Lösungsinstrument zum Auffinden von nicht deterministisch reproduzierbaren Fehler im System genutzt werden. In den Illustrationsszenarios wurde ein „Brute Force“ Ansatz des Analyseinstrumentariums genutzt.

Happend-Before + Heuristiken

Dieses Testwerkzeug kann geschickter eingesetzt werden, was eine Aufwertung des Ansatzes und Potenzial für weitere Arbeiten darstellt. So sollte explizit die „*Happend-Before*“ Relation, analog zu Musuvathi u. a. [Mus+08]; Esparza und Majumdar [EM10] eventuell in Kombination mit Heuristiken genutzt werden, um die Komplexität, und damit die Dauer, des Testansatzes zu reduzieren.

Mit den Erkenntnissen aus Kapitel 6 müssten hierzu „nur“ die globalen Befehle durch einen *Wrapper* (analog zu [Mus+08; EM10]) protokolliert werden. Durch die virtuelle Maschine wäre leicht ein Zurückspringen (ein Reproduzieren aller Zustände des Systems) der Situation vor potenziellen Fehlern möglich. Damit könnte gegenüber dem Stand der Technik und Wissenschaft eine ganzheitliche Testmethodologie geschaffen werden, alle Schichten können auf „nicht deterministisch reproduzierbare“ Fehler getestet werden.

Kapitel 11

Zusammenfassung und Ausblick

Übersicht des Kapitels

Dieses Kapitel fasst die Arbeit zusammen und gibt einen Ausblick auf mögliche, weiterführende Arbeiten.
Abschnitt 11.1 gibt ein Résumé der theoretischen und praktischen Ergebnisse, gegliedert nach den Kapiteln dieser Arbeit.
Abschnitt 11.2 zeigt auf, dass das Gesamtpotenzial der in dieser Arbeit entwickelten Technik noch nicht erschlossen ist, benennt mögliche weiterführende Themen und schließt diese Arbeit ab.

11.1 Zusammenfassung

Übersichtsgrafik

Die Übersichtsgrafik in Abbildung 11.1 illustriert diese Arbeit für die Zusammenfassung. Dunkelgraue Balken repräsentieren die einzelnen Kapitel. Gegliedert wird die Grafik mittels der hellgrauen Balken nach der **Problemstellung** und dem **Lösungsansatz**, den **theoretischen** sowie den **praktischen Ergebnissen**.

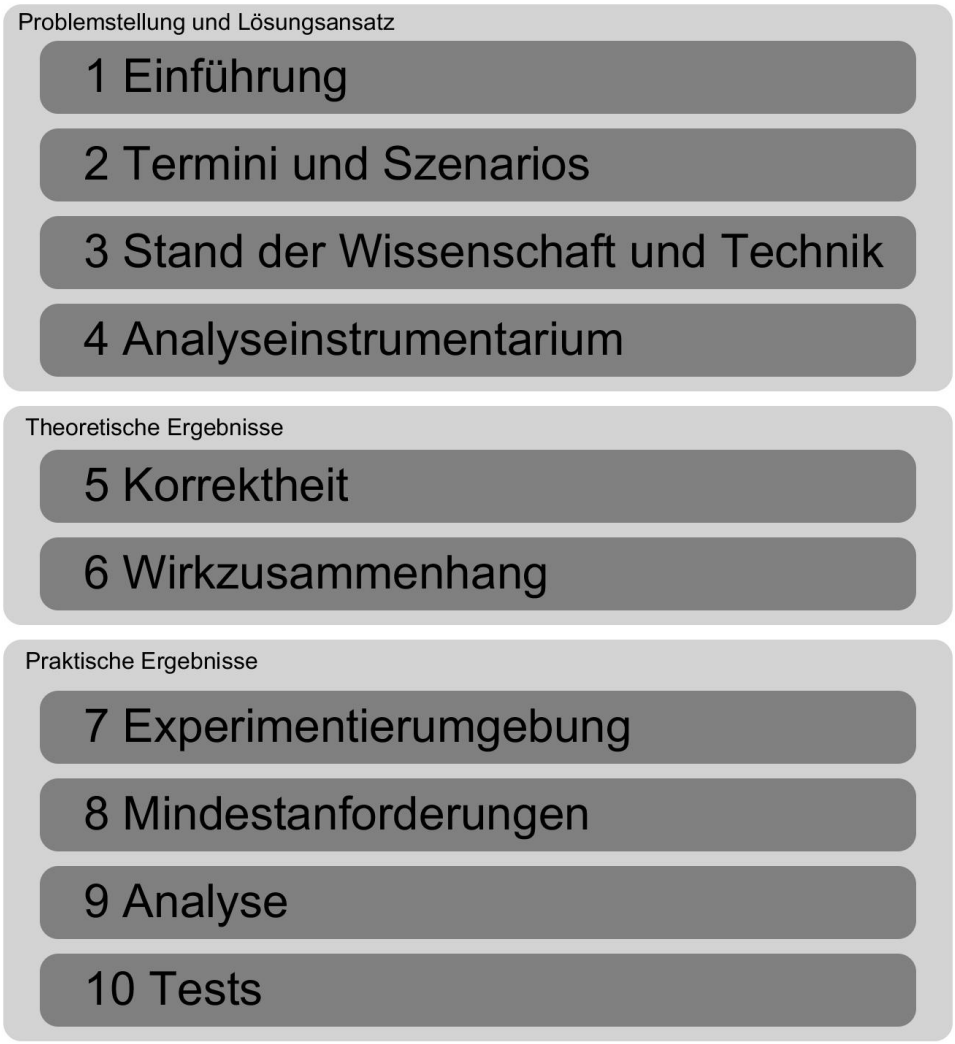


Abbildung 11.1: Zusammenfassung der Arbeit.

❑ **Kapitel** sind durch dunkelgraue Balken repräsentiert, beschriftet mit der Kapitelnummer und dem Titel des Kapitels.

❑ Die Motivation (Problemstellung und Lösungsansatz), die theoretischen und praktischen Ergebnisse dieser Arbeit sind mittels der hellgrauen Blöcke gruppiert.

11.1.1 *Problemstellung und Lösungsansatz*

11.1.1.1 ❑ Kapitel 1 – Einführung

Kapitel 1 motiviert diese Arbeit.

Ausreichende Performanz stellt ein erfolgskritisches Qualititätsmerkmal von IT-Anwendungen dar. Performanzprobleme treten jedoch häufig in der Praxis auf. Durch die hohe Relevanz gibt es im Software-Lebenszyklus verschiede Ansätze ausreichende Performanz sicherzustellen. *A priori* Ansätze werden jedoch durch die benötigte Expertise (der Methodenansätze und des zu erstellenden Systems) nicht generell genutzt, die *ex post* Methoden sind unsystematisch und unstrukturiert.

erfolgskritisches Qualititätsmerkmal Performanz

Ziel dieser Arbeit ist es, Werkzeuge und Methoden für eine Performanzanalyse während und nach der Implementierung zu entwickeln. Dies geschieht auf der Basis eines Experiments. Das Analyseinstrumentarium ist die zielgerichtete Variation von Zeitpunkten und Abläufen im System. Die Analyse erfolgt aus den gemessenen Daten des Experiments. Eine Erweiterung des Analyseinstrumentariums ist eine Testmethodologie auf nicht deterministisch reproduzierbare Fehler, bedingt durch bestimmte zeitliche Bedingungen in Abläufen. Für das Ziel wurden 10 Teil- und Forschungsfragen (α bis κ) benannt. Das Vorgehensmodell der Arbeit wurde beschrieben und grafisch illustriert und die Ergebnisse der Kapitel kurz zusammengefasst.

Ziel dieser Arbeit

11.1.1.2 ❑ Kapitel 2 – Termini und Szenarios

Einleitend wurde in dieser Arbeit der Betrachtungsfokus spezifiziert und die benötigten Termini und Definitionen gegeben. Die in dieser Arbeit zur Demonstration genutzten Szenarios, geeignete Illustrationsszenarios zum einfachen Nachvollziehen sowie ein breites Spektrum an Anwendungsfällen aus der Praxis, wurden vorab gegeben.

Betrachtungsfokus
Termini
Definitionen
Szenarios

❑ α) Die **Teilfragestellung** α — nach dem **Fokus dieser Arbeit** — wurde in Kapitel 2 beantwortet. ✔

Teilfragestellung α

11.1.1.3 ❑ Kapitel 3 – Stand der Wissenschaft und Technik

Zu Beginn der Arbeit wurden die Ansätze nach dem Zeitpunkt im Softwarelebenszyklus kategorisiert. *A priori*, vor der Implementierung des Systems, sollen modellbasierte Ansätze eine Vorhersage der zu erwartenden Performanz ermöglichen. *Ex post* sollen Last- und Stresstests ausreichende Performanz einer IT-Applikation zeigen. Während der Entwicklung werden *Profiling* und *Instrumentierung* eingesetzt, um ein System aus (ausreichend) performanten Komponenten beziehungsweise Modulen zu erstellen.

Ansätze zur Analyse

Ansätze teilweise unzureichend

Die bisherigen Ansätze aus Technik und Wissenschaft sind unzureichend.
Modellbasierten Ansätze abstrahieren oft zu sehr, benötigen eine hohe Expertise und sind, insbesondere deswegen, in der Praxis nicht generell akzeptiert. Implementierungsbasierte Ansätze sind notwendig, aber durch das komplexe Zusammenspiel der entwickelten Komponenten alleine unzureichend. Last- und Stresstests zeigen nur, ob die IT-Anwendung unter Last bzw. Stress die Spezifikation erfüllt, beziehungsweise nutzbar bleibt. Potenzielle Optimierungskandidaten oder Zusammenhänge im System werden nicht identifiziert. Eine nachträgliche Optimierung einer bestehenden IT-Anwendung ist somit ein unsystematischer und unkoordinierter Prozess.

Teilfragestellung β ✔

❑ β) Die **Teilfragestellung** β — ob der **experimentelle Ansatz zur Performanzanalyse (zum Stand der Wissenschaft und Technik) neu ist** — wurde aufbauend auf dem Stand der Wissenschaft und Technik in Kapitel 3 **positiv** beantwortet.

11.1.1.4 ❑ Kapitel 4 – Analyseinstrumentarium

Lösungsansatz Experiment

Im weiteren Verlauf wurde der Lösungsansatz dieser Arbeit präsentiert. Mittels eines Experiments sollen Zusammenhänge in dem untersuchten System erkannt werden. Das

Lösungsinstrument

Lösungsinstrument dieses empirischen Ansatzes, das Analyseinstrumentarium, ist ein zielgerichtetes Ändern von Zeitpunkten oder der Zeitdauer in den Abläufen eines Systems. Drei unterschiedliche Ausprägungen des Analyseinstrumentariums wurden vorgestellt: die *Prolongation* ist eine Verlängerung von Abläufen, die *Retardation* eine Verzögerung des Startzeitpunktes, das *simulierte Optimieren* ein (relatives) Verkürzen von Abläufen. Die Vorteile dieses empirischen Ansatzes zu den bestehenden Ansätzen wurden

Prologation, Retardation, simuliertes Optimieren

Vorteile des empirischen Ansatzes

aufgezählt, z. B. die Integration der Umgebung (spezifische Hardware, Betriebssystem und evtl. gleichzeitig laufende IT-Anwendungen) im Experiment.

Teilfragestellung γ ✔

❑ γ) **Teilfragestellung** γ — nach den **Vorteilen und der Realisation des Analyseinstrumentariums** — wurde beantwortet, in dem die Vorteile des experimentellen Ansatzes zu den bestehenden Ansätzen zur Performanzanalyse diskutiert wurden und die *Prolongation*, die *Retardation* und das *simulierte Optimieren* definiert wurden.

11.1.2 *Theoretische Ergebnisse*

11.1.2.1 ❑ Kapitel 5 – Korrektheit

Ceteris paribus-Validität

Um ein Experiment durchzuführen zu können muss die interne Validität (*Ceteris paribus-Validität*) gegeben sein. Der beobachtete Wirkzusammenhang darf nur durch die Änderung der in dem Experiment variierten Variablen verursacht werden.

Bei einer IT-Anwendung ist die Korrektheit fundamental. Die angedachte, beziehungsweise spezifizierte, Funktionalität muss bei dem Experiment erhalten bleiben. Mittels einer *Turingmaschine*, ein sehr abstraktes und grundlegendes Maschinenmodell der Informatik, wurde bewiesen, dass bei einer Überführungsfunktion ein „*leerer Schritt*“ hinzugefügt werden darf. Ein leerer Schritt ist die Ersetzung der Überführungsfunktion δ in zwei Überführungsfunktionen und kann mit einem *No Operation Befehl* (*NOP*) anderer Maschinenmodelle verglichen werden. Per Induktion wurde dies auf beliebig viele „*leere Schritte*“ erweitert. Mit einer weiteren Induktion wurden diese Ergebnisse auf beliebe Überführungfunktionen abgeschlossen. Man darf bei diesem Maschinenmodell beliebig viele „*leere Schritte*“ („NOPs“) einfügen, ohne dass das Ergebnis einer Berechnung verändert wird.

Korrektheit

Turingmaschine

„leerer Schritt“

Beweis per Induktion

❑ δ) Die **Teilfragestellung** δ — nach der **Validität** bzw. der **Korrektheit**, also ob das Ergebnis einer Berechnung durch das Analyseinstrumentarium nicht verändert wird — wurde mit und an der Turingmaschine bewiesen. Das Ergebnis kann auf sequentielle Berechnungsmodelle übertragen werden. Parallele Interaktionen oder Berechnungen sowie Argumente wie *Stoppuhren* wurden noch nicht betrachtet.

Teilfragestellung δ
✓ bei sequentiellen Berechnungen
✗ bei parallelen Berechnungen

11.1.2.2 ❑ Kapitel 6 – Wirkzusammenhang

Der Wirkzusammenhang und die Erweiterung auf parallele Berechnungen wurde mittels eines praxisnaheren, parallelen Modell, der parallelen Registermaschine oder *Parallel Random Access Machine* (*PRAM*) gezeigt. Vorteilhaft bei diesem Modell sind nicht nur die multiplen, parallelen Prozessoren, sondern auch, dass die Ergebnisse und Erkenntnisse auf Prozesse übertragen werden können. Eingeführt wurde dieses Maschinenmodell mit ihrem sequentiellen Vorgängermodell, der Registermaschine. Der Konsens dieses, in der Literatur sehr unterschiedlich definierten, *PRAM* Maschinenmodells wurde angegeben. Die Konventionen zur Behandlung von Lese- und Schreibkonflikten (unterschiedliche Maschinenmodelle sowie prozessorübergreifende Phasen zum Lesen und Schreiben) wurden gezeigt.

parallele Registermaschine

Registermaschine

Lese- und Schreibkonflikte

Eine praktisch implementierte und gut erforschten parallele Registermaschine stellt die *SB-PRAM* (*Saarbrücken Parallel Random Access Machine*) dar. Aufbauend auf den Arbeiten zur SB-PRAM wurde das formale Maschinenmodell dieser Arbeit, die PRAM^{dt}, formal angegeben. Hierfür wurden Anleihen vom Aufbau und dem Befehlssatz (inklusive der *Multipräfixoperationen* zur einfachen Realisation von Sperren und Barrieren) der *SB-PRAM* gemacht, die Komplexität wurde jedoch stark reduziert durch die gewählte Akkumulator Architektur.

SB-PRAM

PRAM^{dt}

Lamport-Uhren
Happend-Before Relation

Mit den *Lamport-Uhren* und der *Happend-Before Relation* wurde der Wirkzusammenhang gezeigt. Wird bei einer Berechnung im selben Prozess künstlich Zeit, durch eine Instrumentierung mit NOPs, hinzugefügt, kann bei einem Referenzlauf (mit der gleichen Befehlssequenz bis auf die künstlich hinzugefügten Befehle) herausgemessen werden, wie oft die Stelle im Code aufgerufen wird.

Senden und Empfangen

Die Interprozess bzw. Interprozessorkommunikation kann bei diesem Maschinenmodell nur über den gemeinsamen Speicher erfolgen. Wenn vor dem Senden einer Nachricht eines Prozess und das Empfangen einer Nachricht des anderen Prozesses künstlich Zeit durch eine Instrumentierung mit NOPs hinzugefügt wurde, kann bei korrekter Synchronisation der kausale Zusammenhang, bezüglich eines Referenzlaufs, gemessen bzw. herausgemessen werden.

notwendige Synchronisation

Ein Programmierer muss das Senden und das Empfangen von Nachrichten unterschiedlicher Prozesse sicherstellen, d. h. *synchronisieren* (z. B. mittels *Barrieren* oder *Semaphoren*). Ansonsten kann bei realen Maschinen durch unterschiedliche Auslastung oder unterschiedlicher Umgebung ein Fehler entstehen, z. B. ein *Race*, ein *Heisenbug* oder, allgemeiner, ein *nicht deterministisch reproduzierbarer Fehler*. Das Analyseinstrumentarium kann Fehler verursachen, diese können in der Praxis jedoch auch auftreten. Die zeitliche Variation kann demnach zum Validieren von Systemen eingesetzt werden.

Teilfragestellung δ

❑ δ) Die **Teilfragestellung** δ — nach der **Validität** bzw. der **Korrektheit**, also ob das ✔ Ergebnis einer Berechnung durch das Analyseinstrumentarium nicht verändert wird — kann bei parallelen Berechnungsmodellen **nur mit Synchronisationsmechanismen** positiv beantwortet werden. Jedoch muss ein Systementwickler eventuelle Seiteneffekte und andere Zielplattformen einplanen, so dass sich ein Fehler wiederholen könnte. Das Analyseinstrumentarium kann konsequenterweise demnach als eine Technik zum Test auf solche Fehler genutzt werden.

Teilfragestellung ϵ

❑ ϵ) Die **Teilfragestellung** ϵ — nach dem **Wirkzusammenhang des Analyseinstru-** ✔ **mentariums** — konnte mit *Lamport Uhren* und der *Happend-Before Relation* auf Basis der PRAM^{dt} gezeigt werden.

Teilfragestellung ζ

❑ η) Die **Teilfragestellung** ζ — ob **Optimierungspotenzial** im System entdeckt wird ✔ — ergibt sich direkt aus dem Wirkzusammenhang. Durch das Analyseinstrumentarium kann optimierbarer Code und das Nutzen einer gemeinsamen Ressource erkannt werden.

Teilfragestellung κ

❑ κ) Die **Teilfragestellung** κ — nach einer **Testmethodologie** für **Fehler** — wurde in ✗ diesem Kapitel **noch nicht vollständig** beantwortet. Die mögliche Induktion von Fehlern wurde gezeigt, jedoch wurde das Analyseinstrumentarium noch nicht zur Fehlersuche eingesetzt.

11.1.3 Praktische Ergebnisse

11.1.3.1 ❑ Kapitel 7 – Experimentierumgebung

Als praktische Ausgangsbasis für den empirischen Ansatz wurde eine Experimentierumgebung in zwei unterschiedlichen Ebenen eines Systems realisiert.
Auf der „höchsten“ Ebene eines Systems kann der Code einer IT-Applikation instrumentiert werden. Eine *Prolongation*, ein *simuliertes Optimieren* und ein *Tracing* wurden mittels *AspectJ* realisiert und dokumentiert. Die Vor- und Nachteile der Realisation mittels *Blockierung* oder *Busy Waiting* wurden diskutiert. Eine virtuelle Maschine ermöglicht eine bessere Kontrolle über die Hardware, die tiefste Ebene eines Systems, die via Software virtuell repliziert wird. Die bestehende, freie virtuelle Maschine *QEMU* wurde entsprechend instrumentiert, um somit die Experimentierumgebung QEMU^{dt} zu schaffen.

Experimentierumgebung

Instrumentierung

Blockierung vs. Busy Waiting

virtuelle Maschine

❑ η) Die **Teilfragestellung** η — nach der **praktischen Durchführbarkeit** der Variation von Ressourceneigenschaften (mittels des Analyseinstrumentariums) — kann durch die Realisation einer Experimentierumgebung durch Instrumentierung und der Implementierung von QEMU^{dt} positiv beantwortet werden.

Teilfragestellung η ✔

11.1.3.2 ❑ Kapitel 8 – Mindestanforderungen

Software stellt bestimmte Rahmenbedingungen an die Hardware. Diese Rahmenbedingen werden abgeschätzt oder durch umfangreiche und aufwändige Tests mit unterschiedlichen Hardwarekomponenten ermittelt. Mittels der Experimentierumgebung können virtuell die Performanzeigenschaften der Hardware variiert werden. Die Experimentierumgebung ermöglicht einen strukturierten und effizienten Prozess zur Ermittlung der Hardwarerahmenbedingungen bzw. die Reaktion des Systems auf unterschiedliche Leistungsdaten der Hardware, wie dies an zwei Szenarios, basierend auf dem *Android Emulator*, exemplifiziert wurde.

Hardwarerahmenbedingungen

❑ θ) Die **Teilfragestellung** θ — wie **Mindestanforderungen** an die Hardware bestimmt werden können — wurde in diesem Kapitel gezeigt.

Teilfragestellung θ ✔

11.1.3.3 ❑ Kapitel 9 – Analyse

Die (umfangreichen) Daten des Experiments mittels des Analyseinstrumentariums können vielfältig ausgewertet werden. Hierzu wurden Anleihen aus dem *Knowledge Discovery* bzw. dem *Data Mining* und, mittels von *multivariaten Analysemethoden*, aus der Statistik gemacht. Die Ansätze wurden kategorisiert in modellbildende und *strukturentdeckende Verfahren* sowie *Simulationsansätze*.

Analyse

Data Mining

Ausgewählte Analyseverfahren

Ausgewählte Analysemethodiken wurden prägnant präsentiert, zum Beispiel die *Faktorenanalyse* und *Clusteringverfahren*. Mittels des *simulierten Optimierens* konnten Simulationsansätze realisiert werden: eine Simulation besserer oder schlechterer Performanz in einem Experiment. Der *Ressource Dependence Graph* stellt eine besondere, in dieser Arbeit entwickelte, zeitsparende Methode dar, um die Strukturen eines Systems zu eruieren. Exemplifiziert wurde der gesamte Analyseprozess an zwei Illustrationszenarios und fünf Beispielsszenarios. Die in dieser Arbeit entwickelte Analysemethode ist automatisierbar, nicht zeitaufwendig, skaliert durch die genutzten Analysemethoden und erfordert kaum Expertise.

Teilfragestellung ι ✔

❑ ι) Die **Teilfragestellung** ι — nach **Auswertungsmöglichkeiten der Daten** des Experiments um Optimierungskandidaten und -potenzial zu identifizieren — wurde in diesem Kapitel beantwortet.

11.1.3.4 ❑ Kapitel 10 – Tests

Softwaretests

Am Ende der Arbeit wurde gezeigt, wie das Analyseinstrumentarium für Softwaretests genutzt werden kann. Manche Softwarefehler manifestieren sich nicht deterministisch bei gleichen Eingabewerten (die sogenannten *Bohrbugs*), die Literatur hat für diese recht plastische Namen wie „*Heisenbugs*“, „*phase of the moon bugs*“ oder „*Races*“ geprägt. Das Analyseinstrumentarium kann dazu genutzt werden, Fehler in einem System, bedingt durch mangelnde Synchronisationsmechanismen oder verursacht durch bestimmte Effekte basierend auf zeitlichen Abläufen, zu finden.

Auffinden von Fehlern in einem System

Races und Heisenbugs

Der Test wurde an drei diversen Szenarios exemplifiziert. Hier wurde gezeigt wie *Races* (Fehler verursacht durch fehlende Sychronisation im Code) oder *Heisenbugs* (nicht deterministische reproduzierbare Fehler bedingt durch asynchrone oder ungetaktete Hardware) gefunden werden können.

Aging-related faults Zeitraffer

Als Erweiterung wurde eine Testmethodologie auf *Aging-related faults* angegeben. Durch das Analyseinstrumentarium kann ein Zeitraffer realisiert werden. Hierbei läuft in der virtualisierten Umgebung die Zeit extrem viel schneller ab als die physikalische Zeit (*wall clock time*). Somit kann das Verhalten eines Systems nach einer gewissen Laufzeit reproduziert und der Test praktisch durchgeführt werden.

Teilfragestellung κ ✔

❑ κ) Die **Teilfragestellung** κ — ob nicht deterministisch reproduzierbare Fehler zielgerichtet reproduziert werden können — wurde mit dieser **Testmethodologie** beantwortet.

11.2 Ausblick

Die vorliegende Arbeit führt eine neue Technik in den Softwareentwicklungsprozess bzw. in die Informatik ein und konnte viele wichtige und grundsätzliche Fragestellungen klären.

Das Gesamtpotenzial der entwickelten Technik ist bei weitem noch nicht erschlossen.

Dieser Ausblick zeigt weitere interessante Themen und Forschungsansätze. Diese sind klassifiziert nach der Nähe zum Ansatz dieser Arbeit. Übersicht

❑ Abschnitt 11.2.1 zeigt weitere interessante Erweiterungen im Rahmen des Ansatzes dieser Arbeit, einer **Performanzanalyse durch ein Experiment**. empirische Performanzanalyse

❑ Abschnitt 11.2.2 zeigt Möglichkeiten zum Einsatz des Analyseinstrumentariums in den verschiedenen Ansätzen zur **Performanzanalyse**, gruppiert nach dem Zeitpunkt im Softwarelebenszyklus. Performanzanalyse

❑ Abschnitt 11.2.3 zeigt mögliche Forschungsansätze und potenzielle Verwendungsmöglichkeiten der hier entwickelten Technik in der **Informatik**. Informatik

11.2.1 Ausblick zur Performanzanalyse durch ein Experiment

Im Rahmen dieses Ansatzes zur Performanzanalyse durch ein Experiment gibt es noch weitere interessante Möglichkeiten:

11.2.1.1 ❑ Performanzoptimierung durch Prolongation

Exkurs 7.1.1 auf Seite 152 präsentierte einen Ansatz zur Optimierung von Systemen durch künstlich hinzugefügte Blockaden. Durch künstlich hinzugefügte Zeit kann sich der Gesamtdurchsatz eines Systems erhöhen. Die Technik dieser Arbeit kann dazu genutzt werden, eine Bewertungsfunktion für dieses Optimierungsproblem zu finden. Durch das Analyseinstrumentarium, durch künstlich hinzugefügte Zeit, kann die Performanz eines Systems optimiert werden. Optimierung durch Prolongation

11.2.1.2 ❑ Versuchsplanung – *Design of Experiments* (*DOE*)

statistische Versuchsplanung DOE

Design of Experiments [HH04a; GC07; Mon05; BHH05; MMAC09] ist eine wissenschaftliche Vorgehensweise um methodisch Versuche zu planen und deren subsequente Auswertung. Der Aufwand für einen Versuch soll möglichst gering gehalten werden. Dafür werden mehrere Faktoren oder Variablen gleichzeitig variiert. Der in dieser Arbeit präsentierte Ansatz skaliert, der Versuchsaufwand sollte jedoch möglichst minimiert werden. Eine Integration der Theorie und Praxis der statistischen Versuchsplanung (*Design of Experiments*) wäre eine weitere Aufwertung des hier erarbeiteten empirischen Ansatzes.

11.2.1.3 ❑ Data Mining Algorithmen

Data Mining

Der Einsatz von *Data Mining* Algorithmen in der Softwareentwicklung ist ein aktuelles und interessantes Forschungsgebiet. So können und sollten unterschiedliche Clusteringverfahren verwandt werden, die hier frei definierbaren *Metriken* bieten ein enormes Potenzial. So können unterschiedliche Anwendungsfälle bei der Analyse durch spezielle Metriken abgedeckt werden. Insbesondere scheint der Einsatz der *Regressionsanalyse* [FKL07; UM08] und die Verwendung von *neuronalen Netzen* [Hay08; Len09] zur Analyse von Systemen sehr lohnenswert.

11.2.1.4 ❑ Einsatz in unterschiedlichen Softwareentwicklungsmodellen

Die a priori oder modellbasierten Performanzanalysemethoden sind die am weitesten entwickelte Herangehensweisen, insbesondere da sie die Vorgehensweisen mit der meisten wissenschaftlichen Forschung darstellen. Vor der Implementierung, in der Design- und Analysephase, wird ein Performanzmodell erstellt, nach der Implementierung, basierend auf diesem Modell, soll die entwickelte IT-Anwendung die geforderte Performanz aufweisen. Somit wurde stillschweigend ein eher linearer Ansatz zur Softwareentwicklung angenommen. Modelle mit einem iterativen Vorgehen (z. B. das Spiralmodell nach Boehm [Boe86]) eignen sich demnach nicht besonders für ein a priori Performanzmodell, besonders gilt dies für agile Vorgehensweisen der Softwareentwicklung [BW08b] (z. B. für *SCRUM* [SB01; Sut05], der *testgetriebenen Entwicklung* [Bec02; Ast03] oder dem *Extreme Programming* (*XP*) [BA04; Bec+01]).

Annahme: linearer Ansatz iterative Ansätze

Performanzanalyse für iterative Softwareentwicklungsansätze

Die hier in dieser Arbeit entwickelte Ansatz zur Performanzanalyse durch ein Experiment ist flexibel und weitgehend automatisierbar und stellt somit ein hohes Potenzial für iterative und agile Vorgehensweisen in der Softwareentwicklung dar. Diese nachhaltige und breite Verwendbarkeit in unterschiedlichen Vorgehensweisen der Softwareentwicklung sollte in weiterführenden Arbeiten untersucht werden.

11.2.1.5 ❑ Visualisierung

Erweiterung der Visualisierung

Viele Softwareentwickler hatten Akzeptanzprobleme mit den verwendeten statistischen Visualisierungsformen (dem *Biplot*, Clustervisualisierungen bzw. den Diagrammen im Allgemeinen) bzw. wollten sich nicht auf diese Darstellung einlassen. Die nach der Analyse gewonnen Daten sollten in bekannten und bewährten Darstellungsformen der Softwareentwicklung, wie z. B. in *UML Diagrammen* [Gro10; Stö05] dargestellt werden, um eine höhere Akzeptanz bei Softwareentwicklern und Analysten zu erreichen. Dies wäre eine weitere Aufwertung der Analysemethodik, hätte jedoch durch den benötigten *Reverse Engineering* Aufwand den Umfang dieser Arbeit gesprengt.

11.2.2 *Ausblick zu weiteren Ansätzen zur Performanzanalyse*

Zur Performanzanalyse gibt es noch weitere interessante Möglichkeiten:

11.2.2.1 ❑ Erweiterung der modellbasierten Ansätze

- Der hier entwickelte Ansatz könnte dazu genutzt werden, bestehende Modelle zu validieren.
- Ein Reverse Engineering von Drittpartei Komponenten (*Third-Party Vendor* Kompenten) zur Integration in Performanzmodelle ist überaus lohnenswert. Somit könnte der gut erforschte modellbasierte Ansatz genutzt werden zur Integration und Wiederverwendung von bestehenden Komponenten, obwohl die zur Performanzanlyse benötigten Interna ausgeblendet sind.
- Die hier entwickelte Technik könnte mit zielgerichteter Variation von Zeitpunkten und zeitlicher Eigenschaften von Abläufen eine Erweiterung und zusätzliche Analyse **in** Performanzmodellen des modellbasierten Ansatzes darstellen.

11.2.2.2 ❑ Erweiterung der implementierungsbasierten Ansätze

Toolunterstützung Profiler

Implementierungsbasierte Ansätze könnten stark durch eine Toolunterstützung der hier eingeführten Technik profitieren. So sollten Profiler um die hier entwickelte Analysetechnik erweitert werden. Dadurch können insbesondere die stark zunehmende Parallelität und Komplexität bei IT-Anwendungen gemeistert werden.

11.2.2.3 ❑ Erweiterung der messungsbasierten Ansätze

Erweiterung von Last- und Stresstesttools

Messungsbasierte Ansätze führen bereits Messungen durch. Werden nicht nur Eingabewerte qualitativ und quantitativ variiert, sondern auch gezielt die zeitlichen Eigenschaften der Abläufe, können *Bottlenecks* und Zusammenhänge im System leichter identifiziert werden.

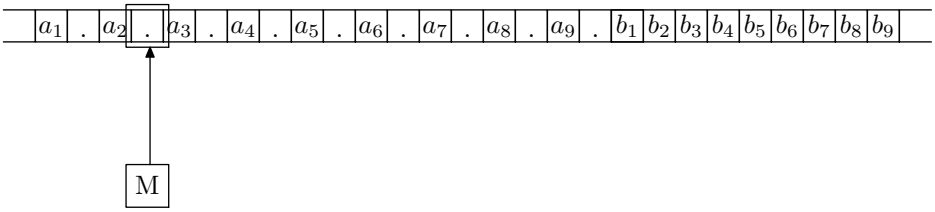


Abbildung 11.2: Turingmaschine mit einer Ressourcenexpansion.

Round Trip Engineering

Insbesondere wäre so durch den experimentellen Ansatz ein phasenübergreifendes, performanzbasiertes *Round Trip Engineering* [Lar03; Het10] realisierbar.

11.2.3 Weitere Ansätze

Auch bezüglich des Analyseinstrumentariums gibt es noch weitere interessante Anwendungsmöglichkeiten. Insbesondere könnte das Analyseinstrumentarium selbst erweitert werden.

11.2.3.1 ❑ Ressourcen Expansion

Speichervariation

Nicht nur eine Variiation der „zeitlichen“, sondern auch der „räumlichen“ Eigenschaften, z. B. eine Ressourcenexpansion, stellt eine interessante Erweiterung des Analyseinstrumentariums dar. So kann beispielsweise der Speicherverbrauch (z. B. die „Größe“ von Objekten) variiert werden, was weitere wertvolle Analysemöglichkeiten erlaubt.

Betriebssystemebene

Da die *Allokation* während der Laufzeit durch das Betriebssystem erfolgt [TG98], bietet sich das von Engel [Eng05] realisierte aspektorientiertes Betriebssystem an. Insbesondere kann in Kombination der Ergebnissen von [Eng05] das Betriebssystem als weitere, beinflussbare (prolongierbare) Ebene im Kontext dieser Arbeit genutzt werden.

Ressourcenerweiterung

Analog zu Kapitel 5 muss dafür die *Validität* gezeigt werden. Dies kann mit einer Turingmaschine mit unterschiedlichen Datenbereichen bewiesen werden (siehe Abbildung 11.2).

Ressourcenverbauch

Mit diesem Ansatz zur Erweiterung des Analyseinstrumentariums kann der Ressourcenverbauch eines Systems allgemein variiert werden. Nicht nur die Performanz von Komponenten als Ressource (z. B. CPU, Festplatte etc.), sondern jede andere Ressource kann verändert werden.

11.2.3.2 ❑ Vorhersagen über die Entwicklung des Ressourcenverbrauchs

Laufzeitverhalten

Mit der in dieser Arbeit entwickelten Methode lassen sich Vorhersagen über die Entwicklung des Ressourcenverbrauchs in einem System treffen. So können Aussagen über das Laufzeitverhalten von Systeme prognostiziert werden, beispielsweise den zu erwarteten Ressourcenverbrauch nach einer bestimmten Laufzeit. Systeme könnten hiermit erweitert werden und auf zukünftige Änderungen reagieren, also ein adaptives System [Che+09] realisiert werden. Denkbar wäre beispielsweise, dass in Stoßzeiten des Systems *Algorithmen* oder Komponenten ausgetauscht werden. Beispielsweise der Austausch der für die Speicherung verantwortlichen Komponenten z. B. die Speicherung via einer Hashfunktion durch einen Baum-Algorithmus oder ähnliches.

11.2.3.3 ❑ Einsatz in der Softwarewartung, Reengineering und Plagiatserkennung

Hoher Aufwand bei der Softwarewartung

Zirka 70% bis 90% der Gesamt**kosten** eines Softwaresystems müssen für die Wartung aufgewendet werden [Boe+78; SRA97; LS80; BP06], , wobei hier bis zu 50% der benötigten **Zeit** zum Verstehen des Programmcodes benötigt werden [Sta84; WS00]. Der Aufwand für ein Reverse Engineering beträgt ca. 50% [HA93; BP06], Fallstudien und Erfahrungen zeigen, dass zusätzlich der Programmcode oft zu einem sehr großen Anteil redundant ist [BP06, S. 12].

Reengineering

Das Analyseinstrumentarium kann neben der Performanzanalyse zu einem Reengineering von Systems genutzt werden. Durch eine geschickte Synthese von Mustererkennung [Bis07; Sch96], Profiling und Analyseinstrumentarium könnten redundante Programmabläufe gefunden werden, ohne ein langwieriges Verstehen oder einer textueller Suche.

Plagiatserkennung

Analog zu Liu u. a. [Liu+06] (siehe Seite 203) können Plagiate erkannt werden. *Kontrollflußabhängigkeiten* können durch das Analyseinstrumentarium implizit erkannt werden ohne dass der Code (vollständig) vorliegen muss. *Datenabhängigkeiten* können erkannt werden, in dem der Zugriff auf Datenbereiche in einem Experiment verlangsamt und Reaktionen gemessen werden.

nicht benutzter Code

Oftmals bestehen Altlastsysteme, insbesondere durch Wartungsfälle, aus einem großen Teil von Programmcode, der oftmals nicht relevant ist, d. h. niemals ausgeführt wird. Das System wird dadurch aufgebläht und der zur Pflege benötigte Auwand explodiert. Durch das Analyseinstrumentarium kann sehr leicht eine automatische beziehungsweise semi-automatische Erkennung durch eine Vorselektion von diesem „toten“ Code erfolgen. Sollten bei den untersuchten Anwendungsfällen bestimmter Code keine Reaktion auf eine Prolongation aller Module zeigen, wird dieser Code nie ausgeführt.

11.2.3.4 ❑ Roboustness Testing

Defekte Hardware

Insbesondere Tests an sicherheitskritischen Softwareprodukten sollten die Möglichkeit in Betracht ziehen, dass Teile des Hardwaresystems, auf dem die Software ausgeführt wird, defekt sind oder plötzlich Defekte aufweisen. „Defekte“ meint hier alle Fehlfunktionen einer Hardwarekomponente, vom sporadischen Fehlverhalten bis hin zum Totalausfall. Kompletterweise sollte jedes Softwareprodukt auf diese Eventualitäten getestet werden, so dass das Verhalten des Systems bei schadhafter Hardware, Hardware die nicht bzw. plötzlich nicht mehr ihrer Spezifikation entspricht bzw. Hardware die nicht den Anforderungen des Softwareprodukts gerecht wird, abgeschätzt werden kann.

Robustheit

Eine wesentliche Anforderung an sicherheitskritische Systeme ist die Robustheit. Der IEEE Standard 610 von 1990 definiert einen Robustness eines Systems als

> **37 Robustness**
> Robustness is defined as the degree to which a system operates correctly in the presence of exceptional inputs or stressful environmental conditions. [IEE90]

Test auf Hardwaredefekte

Auf Basis einer Idee von Herrn *Dr. Harald Rölle* wurde im Rahmen dieser Arbeit mit Hilfe der Virtualisierungstechnologie ein Verfahren zum Robustness Testing entwickelt und als Patent angemeldet [MR10a; MR09a]. Somit können Softwareprodukte auf real auftretende Hardwaredefekte getestet und eingeschätzt werden, was bislang nicht oder nur schwer möglich war. Damit wird eine umfangreichere Testabdeckung ermöglicht. Die getestete Software ist sicherer, der Testprozess wird effizienter und kostengünstiger. Es muss weder das Betriebssystem, noch die zu testende Applikation verändert werden. Die Testmethodologie kann Systeme auf ihre Robustheit bzw. auf zeit- und ressourcenkritische Aspekte testen, was durch die Virtualisierung und der in dieser Arbeit entwickelten Technik zielgerichtet, effizient und strukturiert möglich wird.

Forschungspotenzial

Dieser Ansatz, die Synthese von *Fault injections* [HTI97; Arl+90] zum *Robustness Testing* oder Kompatibilitätstesten birgt noch enormes Forschungspotenzial und vielgestaltige Möglichkeiten für die Systementwicklung.

11.2.3.5 ❑ Maß für die Parallelisierbarkeit

Parallelität bei der Hardware

Nach dem *Moor'schen Gesetz* verdoppelt sich die Integrationsdichte integrierter Schaltkreise alle achtzehn Monate [Mol06]. Die Grenzen der Miniaturisierung sind jedoch bald erreicht. Bei gleichbleibender Skalierung würde ab dem Jahr 2020 physikalisch weniger als ein Elektron für einen Transistor zur Verfügung stehen [BW00]. Konsequenterweise und aus wirtschaftlichen Aspekten werden Chips mit mehreren Prozessoren ausgestattet, auch hier erhöhen sich die Anzahl der Prozessoren (von hunderten Prozessoren [HBK06] bis auf tausende Prozessoren [Asa+06]).

Parallelisierung von Software

Um die Effizienz von Multicore ausnützen zu können, muss die Software optimal parallelisiert werden können [HM08, S. 2]. Dies soll mittels speziellen Architekturen oder Compilern bzw. Kompilertechniken und ähnlichem automatisiert werden. Der Entwickler soll vom langwierigen, langweiligen und fehlerbehafteten Prozess der Adaption seines Codes auf Multicorerechnern befreit werden.

Performanzprobleme

Jedoch erlangt die gemessene Performanz bei realen IT-Applikationen zu meist nur ein Bruchteil der theoretisch erreichbaren Performanz. Gründe hierfür liegen in der komplexen Interaktion zwischen den einzelnen Komponenten bzw. Ebenen des Systems. Die Effekte der einzelnen Interaktionen werden hierdurch nicht verstanden. [Wol02]

Experiment

Mit Hilfe der Virtualisierungstechnologie können zeitliche Eigenschaften einzelner Codefragmente und Ausführungsstränge dediziert varriert werden. Durch ein Experiment mit dem Analyseinstrumentarium können auf Abhängigkeiten dieser Ausführungsstränge und Codefragmente untereinander geschlossen werden. Somit ist es möglich eine Metrik zu entwickeln, die eine Vorhersage zulässt, ob sich eine Verteilung der Software auf verteilte Hardware lohnt, oder ob der Synchronisationsoverhead eine Verteilung eher verbietet.

Zusammenhangsgraph

Die Idee zum Maß für die Parallelisierbarkeit ist nun folgende: Das Analyseinstrumentarium erkennt durch ein Experiment die Zusammenhänge der Systemkomponenten. Weisen zwei Komponenten nach einer Prolongation keine Laufzeitvariation auf, arbeiten diese nicht zusammen, sie sind „echt“ asynchron. Alle diese Komponenten können durch einen gewichteten Graph verbunden werden, wobei die Wichtung ein Maß für den Zusammenhang, evtl. auch abhängig von der Hardware, darstellt. Der Graph kann nun anhand der (geringsten) Gewichte in Teilgraphen zerschnitten werden. So kann auf eine optimale Verteilung der Komponenten auf die verteilte Hardware geschlossen werden.

11.2.4 Fazit

Erweiterung der Arbeit

Die eben genannten Vorschläge dienen nur als Beispiele, die aufzeigen sollen, wie diese Arbeit fortgeführt und erweitert werden kann. Sie demonstrieren die weiteren, vielgestaltigen Einsatzmöglichkeiten der hier eingeführten Technik. Viele weitere Themengebiete und Ansätze sind denkbar und sollten realisiert werden.

weiteres Forschungspotenzial

Mit den oben kurz beschriebenen Ansätzen wurde gezeigt, dass diese Arbeit nicht nur einen eigenen wissenschaftlichen Beitrag leistet, sondern auch ein enormes Potenzial für weitere Forschung und wertvolle Werkzeuge für den Softwareentwicklungsprozess bietet.

Anhang A

PRAM^{dt} Mikrobefehle

A.1 Lokale Operationen

38 AND Befehlsformat: `AND` `lokale Adresse`

$$\left.\begin{array}{l} Acc_i \leftarrow Acc_i \wedge L_i(IA) \\ IC_i \leftarrow IC_i + 2 \end{array}\right\} \text{lokale Operation}$$

39 OR Befehlsformat: `OR` `lokale Adresse`

$$\left.\begin{array}{l} Acc_i \leftarrow Acc_i \vee L_i(IA) \\ IC_i \leftarrow IC_i + 2 \end{array}\right\} \text{lokale Operation}$$

40 NAND Befehlsformat: `NAND` `lokale Adresse`

$$\left.\begin{array}{l} Acc_i \leftarrow Acc_i \mathbin{\overline{\wedge}} L_i(IA) \\ IC_i \leftarrow IC_i + 2 \end{array}\right\} \text{lokale Operation}$$

41 XOR Befehlsformat: `XOR` `lokale Adresse`

$$\left.\begin{array}{l} Acc_i \leftarrow Acc_i \oplus L_i(IA) \\ IC_i \leftarrow IC_i + 2 \end{array}\right\} \text{lokale Operation}$$

42 LOAD Befehlsformat: `LOAD` `lokale Adresse`

$$\left.\begin{array}{l} Acc_i \leftarrow L_i(IA) \\ IC_i \leftarrow IC_i + 2 \end{array}\right\} \text{lokale Operation}$$

43 STORE Befehlsformat: `STORE` `lokale Adresse`

$$\left.\begin{array}{l} L_i(IA) \leftarrow Acc_i \\ IC_i \leftarrow IC_i + 2 \end{array}\right\} \text{lokale Operation}$$

44 LOADK Befehlsformat: `LOADK` `Konstante`

$$\left.\begin{array}{l} Acc_i \leftarrow Konstante \\ IC_i \leftarrow IC_i + 2 \end{array}\right\} \text{lokale Operation}$$

45 ADD Befehlsformat: `ADD` `lokale Adresse`

$$\left.\begin{array}{l} Acc_i \leftarrow Acc_i + L_i(IA) \\ IC_i \leftarrow IC_i + 2 \end{array}\right\} \text{lokale Operation}$$

46 SUB Befehlsformat: `SUB` `lokale Adresse`

$$\left.\begin{array}{l} Acc_i \leftarrow Acc_i - L_i(IA) \\ IC_i \leftarrow IC_i + 2 \end{array}\right\} \text{lokale Operation}$$

47 NOP Befehlsformat: `NOP`

$$\left.\begin{array}{l} IC_i \leftarrow IC_i + 1 \end{array}\right\} \text{lokale Operation}$$

48 JUMP Befehlsformat: `JUMP` `Label`

$$\left.\begin{array}{l} IC_i \leftarrow label \end{array}\right\} \text{lokale Operation}$$

49 JZERO Befehlsformat: `JZERO` `label`

$$\left.\begin{array}{l} \textit{if } Acc_i \leq 0 \textit{then} \\ \quad IC_i \leftarrow label \\ \textit{else} \\ \quad IC_i \leftarrow IC_i + 2 \end{array}\right\} \text{lokale Operation}$$

50 PUSH Befehlsformat: `PUSH` `Registeridentifikator`

$$\left.\begin{array}{l} S(SP_i) \leftarrow R \\ IC_i \leftarrow IC_i + 2 \end{array}\right\} \text{lokale Operation}$$

51 POP Befehlsformat: `POP` `Registeridentifikator`

$$\left.\begin{array}{l} R \leftarrow S(SP_i) \\ IC_i \leftarrow IC_i + 2 \end{array}\right\} \text{lokale Operation}$$

52 CALL Befehlsformat: `CALL` `Label`

$$\left.\begin{array}{l} PUSH\ PSW_i \\ PUSH\ IC_i + 2 \\ IC_i \leftarrow label \end{array}\right\} \text{lokale Operation}$$

53 RET Befehlsformat: `RET`

$$\left.\begin{array}{l} POP\ IC_i \\ POP\ PSW_i \end{array}\right\} \text{lokale Operation}$$

A.2 Globale Operationen

54 HALT Befehlsformat: `HALT`

$$\left.\begin{array}{l} PSW_i.inaktiv \leftarrow 1 \end{array}\right\} \text{lokale Operation}$$

55 LOADINDEX Befehlsformat: `LOADINDEX`

$$\left.\begin{array}{l} Acc_i \leftarrow PSW_i.INDEX \\ IC_i \leftarrow IC_i + 1 \end{array}\right\} \text{lokale Operation}$$

56 FORK

P_j bezeichnet den ersten freien Prozessor.
P_i bezeichnet den Prozessor, der den *FORK* Befehl ausführt.
$R_{0,j}$ bezeichnet den Akkumulator des ersten freien Prozessors.
$R_{0,i}$ bezeichnet den Akkumulator des ausführenden Prozessors.

Befehlsformat: `FORK` `label`

- FOR $j = 0$ to ∞
  if $PSW_j.inaktiv$ then
  $IC_j \leftarrow label$
  $Acc_j \leftarrow Acc_i$
  $IC_i \leftarrow IC_i + 2$
  $PSW_j.inaktiv \leftarrow 0$
  exit for

57 WRITE Befehlsformat: `WRITE` `globale Adresse`

$$\left.\begin{array}{l} M(gA) \leftarrow Acc_i \\ IC_i \leftarrow IC_i + 2 \end{array}\right\} \text{Schreiben}$$

58 READ Befehlsformat: `READ` `globale Adresse`

$$\left.\begin{array}{l} Acc_i \leftarrow M(gA) \\ IC_i \leftarrow IC_i + 2 \end{array}\right\} \text{Lesen}$$

A.3 Multipräfixoperationen

Zur Vereinfachung kann jeweils nur einer der vier Multipräfixoperationen pro Zyklus auf eine identische, gemeinsame Speicherzelle ausgeführt werden.

59 MPADD Befehlsformat: `MPADD` `globale Adresse`

Die Prozessoren $P_a, \ldots, P_m$ $(j < j+1)$ führen **SYNCHRON** die Operation $MPADD < adress >, ACC_j$ aus.

Der aktuelle Wert des Akkumulators jedes Prozessors ($ACC_j = D_j$) fließt in die Berechnung als Operand der Addition ein. Auf eine Speicherzelle M(gA) wird von Prozessor P_j die Multipräfix-Addition mit Operand D_j durchgeführt.

$$\begin{array}{l} \textit{For } k \in \{PSW_a.INDEX, \ldots, P_m.PSW.INDEX\} \\ \quad \left.\begin{array}{l} MP_k \leftarrow Acc_k \\ Acc_k \leftarrow M(gA) \end{array}\right\} \text{Lesen} \\ \textit{For } k \in \{PSW_a.INDEX, \ldots, P_m.PSW.INDEX\} \\ \quad \left.\begin{array}{l} IC_k \leftarrow IC_k + 2 \end{array}\right\} \text{lokale Operation} \\ \textit{For } k \in \{PSW_a.INDEX, \ldots, P_m.PSW.INDEX\} \\ \quad \left.\begin{array}{l} M(gA) \leftarrow M(gA) + MP_k \end{array}\right\} \text{Schreiben} \end{array}$$

60 MPAND Befehlsformat: `MPAND` `globale Adresse`

Die Prozessoren $P_a, \ldots, P_m$ $(j < j+1)$ führen **SYNCHRON** die Operation $MPAND < adress >, ACC_j$ aus.

Der aktuelle Wert des Akkumulators jedes Prozessors ($ACC_j = D_j$) fließt in die Berechnung als Operand in die Konjunktion ein. Auf eine Speicherzelle M(gA) wird von Prozessor P_j die Multipräfix-Konjunktion mit Operand D_j durchgeführt.

$$\begin{array}{l} \textit{For } k \in \{PSW_a.INDEX, \ldots, P_m.PSW.INDEX\} \\ \quad \left.\begin{array}{l} MP_k \leftarrow Acc_k \\ Acc_k \leftarrow M(gA) \end{array}\right\} \text{Lesen} \\ \textit{For } k \in \{PSW_a.INDEX, \ldots, P_m.PSW.INDEX\} \\ \quad \left.\begin{array}{l} IC_k \leftarrow IC_k + 2 \end{array}\right\} \text{lokale Operation} \\ \textit{For } k \in \{PSW_a.INDEX, \ldots, P_m.PSW.INDEX\} \\ \quad \left.\begin{array}{l} M(gA) \leftarrow M(gA) \wedge MP_k \end{array}\right\} \text{Schreiben} \end{array}$$

61 MPOR Befehlsformat: `MPOR` `globale Adresse`

Die Prozessoren $P_a, \ldots, P_m$ $(j < j+1)$ führen **SYNCHRON** die Operation $MPOR < adress >, ACC_j$ aus.

Der aktuelle Wert des Akkumulators jedes Prozessors ($ACC_j = D_j$) fließt in die Berechnung als Operand in die Disjunktion ein. Auf eine Speicherzelle M(gA) wird von Prozessor P_j die Multipräfix-Disjunktion mit Operand D_j durchgeführt.

$$
\begin{array}{l}
For\ k \in \{PSW_i.INDEX, \ldots, P_n.PSW.INDEX\} \\
\quad \left.\begin{array}{l} MP_k \leftarrow Acc_k \\ Acc_k \leftarrow M(gA) \end{array}\right\} \text{Lesen} \\
For\ k \in \{PSW_a.INDEX, \ldots, P_m.PSW.INDEX\} \\
\quad \left.\begin{array}{l} IC_k \leftarrow IC_k + 2 \end{array}\right\} \text{lokale Operation} \\
For\ k \in \{PSW_a.INDEX, \ldots, P_m.PSW.INDEX\} \\
\quad \left.\begin{array}{l} M(gA) \leftarrow M(gA) \vee MP_k \end{array}\right\} \text{Schreiben}
\end{array}
$$

62 MPMAX Befehlsformat: `MPMAX` `globale Adresse`

Die Prozessoren $P_a, \ldots, P_m$ $(j < j+1)$ führen **SYNCHRON** die Operation $MPMAX < adress >, ACC_j$ aus.

Der aktuelle Wert des Akkumulators jedes Prozessors ($ACC_j = D_j$) fließt in die Berechnung als Operand in die Maximumsfunktion ein. Auf eine Speicherzelle M(gA) wird von Prozessor P_j die Multipräfix-Maximumsfunktion mit Operand D_j durchgeführt.

$$
\begin{array}{l}
For\ k \in \{PSW_i.INDEX, \ldots, P_n.PSW.INDEX\} \\
\quad \left.\begin{array}{l} MP_k \leftarrow Acc_k \\ Acc_k \leftarrow M(gA) \end{array}\right\} \text{Lesen} \\
For\ k \in \{PSW_a.INDEX, \ldots, P_m.PSW.INDEX\} \\
\quad \left.\begin{array}{l} IC_k \leftarrow IC_k + 2 \end{array}\right\} \text{lokale Operation} \\
For\ k \in \{PSW_a.INDEX, \ldots, P_m.PSW.INDEX\} \\
\quad \left.\begin{array}{l} max\,(M(gA), MP_k) \end{array}\right\} \text{Schreiben}
\end{array}
$$

Anhang B

Simuliertes Optimieren

komplette Dokumentation

Der hier wiedergegebene Aspekt bzw. *Aspect* wurde in [Mano7] erarbeitet.
Der Code wird hier, leicht variiert, zur Vollständigkeit dokumentiert. Somit sollen Prolongation, Tracing und simuliertes Optimieren beziehungsweise die Experimentierumgebung durch eine Instrumentierung via AspectJ leicht reproduziert werden können.
Eine Beschreibung des simulierten Optimierens befindet sich im Abschnitt 7.1.7 auf Seite 155.

B.1 Der Aspekt zum simuliertem Optimieren

Simulation

Alle Methoden die mittels **`&&! cflow`** in Listing B.1 spezifiziert sind, werden optimiert simuliert (relativ verkürzt). [Mano7]

```
package verlaengern;

import trace.Trace2;

public aspect Verlaengern {
  declare precedence :Trace2,Verlaengern;

  pointcut Verlaengern(): call (* *.* (..)) && ! cflow simulierte Methode
  && ! within(Trace2) ... weitere nicht simulierte Packages, Klassen etc.

  Object around(): Verlaengern()
  {
    long dt;

    tvll.entry("" + Thread.currentThread(), "" + thisJoinPoint);
    dt = System.nanoTime();
```

```
    dt = dt / 1000;

    try {
      return proceed();
    }
    finally {
      // Umschliessende Zeit messen
      dt = (System.nanoTime() / 1000 - dt);

      long ausgeschlosseneZeit = tvll.exitTime("" +
       Thread.currentThread(), dt);

      // richtig verlaengern (eingeschlossene Zeit ist schon weg)!

      dt = dt - ausgeschlosseneZeit;
      dt = dt * verlaengerungsfaktor;

      // ms, ns berechnen
      long ms = (dt) / 1000l;
      int ns = (int) (dt) % 1000;

      // .. und verlaengern!
      hier geschieht das Einfügen von Zeit bzw. künstlichem Ressourcenverbrauch
           durch beispielsweise ein Busy Waiting (siehe Abschnitt 7.1)
    }
  }

  public static int verlaengerungsfaktor = 0;
  protected static ThreadVerlaengernLinkedList tvll;

  public static void main(String[] args) {
      // verlaengerungsfaktor bzw. Optimierungsfaktor setzen!!!
      verlaengerungsfaktor = j;
      // zu simulierendes System aufrufen
      program.vv1.maincall();
    }
  }

  // Hilfsklasse
  static {
    tvll = new ThreadVerlaengernLinkedList();
  }
}
```

Listing B.1: AspectJ: Aspect zum simulierten Optimieren

B.2 Die Verkettete Liste

Detailliert beschrieben ist die verkettete Liste in Abschnitt 7.1.2.1 auf Seite 143. Sie wird für ein multithreading-fähiges Simulieren benötigt. In jedem Element sind Informationen über den aktuellen Thread gespeichert. Nur so kann ein akkurates simuliertes Optimieren sichergestellt werden. [Mano7] multithread

```
package verlaengern;

import java.math.BigInteger;
import java.util.LinkedList;
import java.util.ListIterator;

public class ThreadVerlaengernLinkedList {

  LinkedList ll;

  ThreadVerlaengernLinkedList() {
    ll = new LinkedList();
  }

  public synchronized void add(Object o) {
    ll.add(o);
  }

  public synchronized long entry(String thread) {
    int index;
    ThreadVerlaengern tl_temp;

    index = this.getIndex(thread);
    {
      if (index >= 0) {

        tl_temp = (ThreadVerlaengern) ll.get(index);
        tl_temp.logtiefe++;

        tl_temp.stack.push(new BigInteger("0"));
        tl_temp.methodenStack.push(methode);
        return tl_temp.logtiefe;
```

```
    } else {
      tl_temp = new ThreadVerlaengern(0, thread);
      tl_temp.stack.push(new BigInteger("0"));

      this.add(tl_temp);
      return 0;
    }
  }
}

public synchronized long exitTime(String thread, long time) {
  int index;
  long rw;
  ThreadVerlaengern tl_temp;
  BigInteger bi;

  // SUCHEN
  index = getIndex(thread);

  // LOGTIEFE VERAENDERN!!!
  tl_temp = (ThreadVerlaengern) ll.get(index);

  tl_temp.logtiefe--;
  // return rw;

  if (tl_temp.logtiefe < 0) {
    tl_temp.stack.clear ();
    return 0;
  } else {
    // aktuelle Methodenzeit holen
    // poppen
    bi = (BigInteger) tl_temp.stack.pop();

    // Rueckgabewert ist die aktuelle Methodenzeit !!!
    rw = bi.longValue();

    // und Zeit dazurechnen ...
    bi = bi.add(new BigInteger("" + time));
```

```
    // ... fuer die naechste Methode dieses speichern !!!
    bi = bi.add((BigInteger) tl_temp.stack.pop());
    tl_temp.stack.push(bi);

    return rw;
  }
}

public synchronized int getIndex(String thread) {
  ThreadVerlaengern tl_temp;

  if (ll.size() == 0) {
    return -1;
  }

  ListIterator li = ll.listIterator(0);
  tl_temp = (ThreadVerlaengern) ll.get(0);

  if (tl_temp.thread.equals(thread))
    return li.previousIndex() + 1;
  else
    while (li.hasNext()) {

      tl_temp = (ThreadVerlaengern) li.next();

      if (tl_temp.thread.equals(thread))
        return li.previousIndex();
    }
  return -1;
}

public synchronized void exit(ThreadVerlaengern tl) {

  ThreadVerlaengern tl_temp;

  if (ll.contains(tl)) {

    tl_temp = (ThreadVerlaengern) ll.get((ll.indexOf(tl)));
    tl_temp.logtiefe--;
```

```
    } else
      ll.add(tl);
  }
}
```

Listing B.2: Java: Die erweiterte verkettete Liste

B.3 Hilfsklasse ThreadVerlaengern

einmaliges Verlängern

Für jeden Thread wird ein Objekt (siehe B.3 erstellt, dass in die verkettete Liste eingefügt wird. So wird vermieden, dass bereits verlängerte Methoden erneut verlängert/prolongiert werden. [Mano7]

```
package verlaengern;
import java.math.BigInteger;
import java.util.Stack;

public class ThreadVerlaengern {

  public long logtiefe = 0;
  public String thread = "";
  public Stack  stack;

  ThreadVerlaengern (long logtiefe, String thread)
  {
    this.logtiefe = logtiefe;
    this.thread = thread;
    this.stack = new Stack();
  }

}
```

Listing B.3: Java: Elemente der verketteten Liste

Anhang C

Betriebssysteme für QEMU^{dt}

Betriebssysteme

Zum einfachen Nachvollziehen wird hier der Download und der Boot verschiedener Betriebssysteme beschrieben.

Download als .iso

Unterschiedliche Betriebssysteme für QEMU bzw. QEMU^{dt} können beispielsweise von `http://free.oszoo.org/` bezogen werden. Vorteilhaft ist ein Download als *ISO image* (dem herkömlichen CD-ROM Format (*ISO 9660 file system*) [Int88]). Mit dieser Datei kann QEMU bzw. QEMU^{dt} mit dem Schalter `-cdrom` wie von einer CD-Rom gebootet werden (`-boot d`).

Installation

Die Installation auf ein zuvor erstelltes QEMU-Image (mit `qemu-img create`) ist betriebssystemspezifisch und wird hier nicht beschrieben. Nach Beendigung der Installation kann man durch ein gleichzeitiges Drücken von Ctrl + Alt + 2 (`STRG + ALT + 2`) in den QEMU Monitor gelangen. Dort gibt man `quit` ein. Das installierte Betriebssystem kann nachfolgend gestartet werden.

Abbildung C.1: Screenshot: Installation von FreeDos auf einer QEMU[dt] Virtual Machine.

C.1 DOS

DOS *DOS*, die Abkürzung für *Disk Operating System* [Tano9], beschreibt eine Familie von Betriebssystemen, die den PC Markt (*IMB kompatibel*) von 1980 bis in das Jahr 2000 beherrschte. DOS wird immer noch gerne für eingebettete Systeme genutzt [Edw03].

ODIN Dos Für den Download von *ODIN Dos* [Nic] werden folgende Befehle in die Shell eingegeben:

```
wget http://odin.fdos.org/odin2005/odin1440.img
qemu-img create -f qcow2 DOS.img 200M
qemu -fda odin1440.img -hda DOS.img -boot a
```

Listing C.1: Shell: Download und Boot von ODIN Dos

FreeDos Das umfangreiche *FreeDos* des *FreeDOS-Projektes* [HFT] kann wie folgt geladen und gebootet werden:

```
wget http://www.ibiblio.org/pub/micro/pc-stuff/freedos/files/distributions/
     1.0/fdfullcd.iso
qemu-img create -f qcow2 DOS.img 200M
qemu -fda /root/floppy -hda fdfullcd.iso -boot a
```

Listing C.2: Shell: Download und Boot von von FreeDos

Nach der Installation kann *FreeDos* mit `qemu DOS.img -fda /root/floppy -cdrom fdfullcd .iso -boot d` direkt von der Imagedatei gestartet werden.

Der Schalter `-fda /root/floppy` ist zum Transfer von Daten zwischen *Host* und *Guest*/*Gast* (siehe Abschnitt C.7).

C.2 Linux - cfLinux

cfLinux ist ein kleines, *embedded Linux* [Koj]. Mittels `wget` kann von `http://www.cflinux.hu/` eine *ISO-Distribution* heruntergeladen werden. cfLinux

```
wget ftp://ftp.cflinux.hu/pub/cflinux/iso/cflinux-1.0.iso
qemu-img create -f qcow disk.img 200M
qemu -hda disk.img -cdrom cflinux-1.0.iso -boot d
```

Listing C.3: Shell: Download von cfLinux

Nun der Installationsanweisung von *cfLinux* folgen. Nachfolgend kann man sich dem Account `root` und dem Passwort `cfdef` eingeloggen.

C.3 Linux – Puppy Linux

Puppy Linux ist eine kleine, performante und graphische Linux-Distribution, welche gerne als Betriebssystem für *Netbooks* [Boro9] sowie für eingebettete Systeme (*embedded Systems*) [GDSo9] eingesetzt wird. Insbesondere gibt es eine Synthese aus QEMU und Puppy Linux namens *QEMU-Puppy* mit der direkt von einem USB-Stick oder einer CD (auch im laufenden Betrieb eines Gastbetriebssystem) das emulierte Puppy Linux gestartet werden kann. Puppy Linux

```
wget http://distro.ibiblio.org/pub/linux/distributions/puppylinux/puppy-5.0/
     lupu-500.iso
qemu-img create -f qcow2 puppy.img 500M
qemu -hda puppy.img -cdrom lupu-500.iso -boot d
```

Nach der Installation kann mittels `qemu -hda puppy.img` direkt von der Imagedatei gestartet werden.

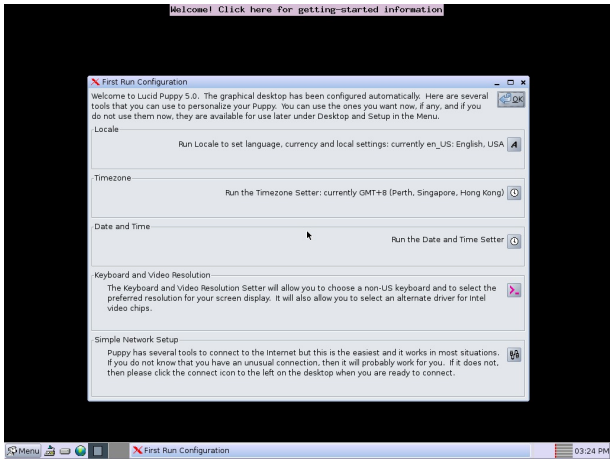


Abbildung C.2: Screenshot: Installation von Puppy Linux auf einer QEMU^{dt} Virtual Machine.

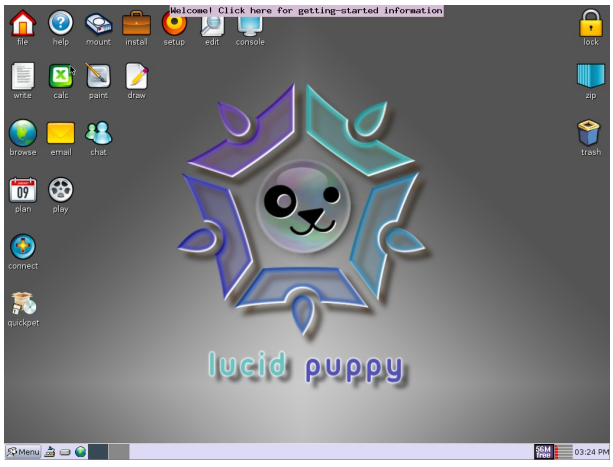


Abbildung C.3: Screenshot: Puppy Linux auf einer QEMU^{dt} Virtual Machine.

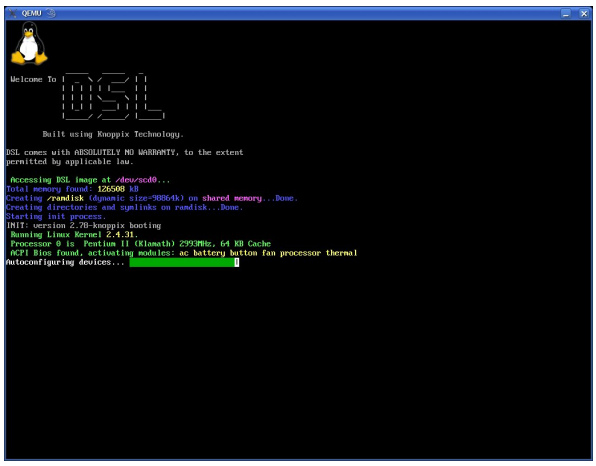


Abbildung C.4: Screenshot: Bootvorgang von DSL (Damned Small Linux) auf einer QEMU^{dt} Virtual Machine.

C.4 *Linux – Damn Small Linux*

Damn Small Linux (*DSL*) ist ein graphisches, linuxbasiertes Betriebssystem [SANo8], welches speziell für ältere, nicht leistungsstarke Rechner entwickelt wurde. Gerade deswegen hatte es bei Tests unter QEMU^{dt} eine annehmbare Geschwindigkeit trotz des Emulationsoverheads, im Gegensatz zu *Puppy Linux*. Es kommt mit wenig Speicher aus und wird gerne als Betriebssystem für *Netbooks* genutzt.

Damn Small Linux

```
wget ftp://ibiblio.org/pub/Linux/distributions/damnsmall/current/dsl-4.4.10.
     iso
qemu-img create -f qcow2 dsl.img 500M
qemu -hda dsl.img -cdrom dsl-4.4.10.iso -boot d
```

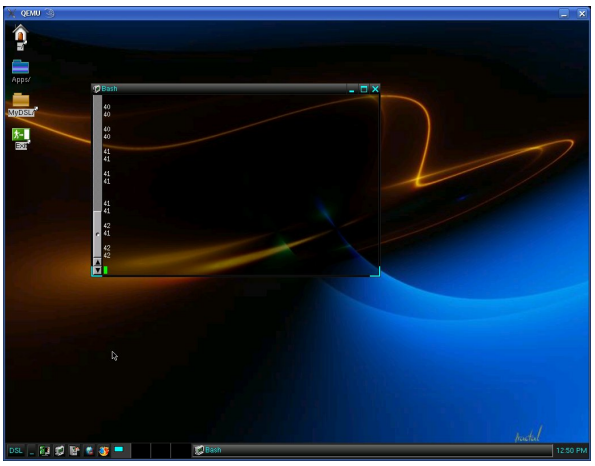


Abbildung C.5: Screenshot: DSL (Damned Small Linux) auf einer QEMU^{dt} Virtual Machine.

C.5 *Linux – muLinux*

muLinux

muLinux, oft auch μLinux geschrieben, ist eine ältere Linux-Distribution. Sie ist extrem klein und für ältere Rechner (*i386* mit 8MB RAM) geeignet [Ando9]. Durch den Emulationsoverhead eignet sich diese Distribution besonders für QEMU^{dt} unter Standard-PCs. Extrem vorteilhaft ist der integrierte *gcc* (*Gnu C Compiler*).

```
wget http://ftp.gwdg.de/linux/mulinux/mu/iso/mulinux-14r0.iso
qemu-img create -f qcow2 mu.img 500M
qemu -hda mu.img -cdrom mulinux-14r0.iso -boot d
```

Listing C.4: Shell: Download und Start von muLinux

muLinux wird nicht installiert, sondern „gecloned“. Hierzu werden folgende Befehle in die Shell eingegeben.

```
guest:~ fdisk
guest:~ clone
```

Listing C.5: Gast Shell: Klonen von muLinux

Hier sollte als Installationspartition `/dev/hda1/` gewählt werden um mit `qemu mu.img` von der (virtuellen) Festplatte zu starten.

C.6 *Android*

Android ist ein Betriebssystem oder Framework, von Google entwickelt für mobile Geräte wie *Smartphones* und *Netbooks* [Sha+10]. In Szenario 4 und Szenario 5 auf Seite 27 wird Android beschrieben. Android

```
wget http://android-x86.googlecode.com/files/eeepc-v0.9.iso
qemu-img create -f qcow2 android.img 1G
qemu -soundhw es1370 -net nic -net user -cdrom eeepc-v0.9.iso -hda android.
     img
-boot d
```

Listing C.6: Shell: Download und Boot von Google Android

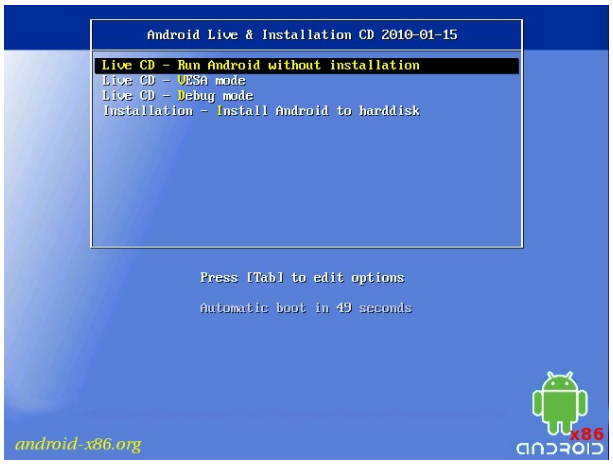


Abbildung C.6: Screenshot: Installation von Google Android auf einer QEMU^{dt} Virtual Machine.

Die Versionsnummer `-v0.9` muss für aktuellere Distributionen ausgetauscht werden und kann in der Shell durch ein Wildcard ('$*$') ersetzt werden.

C.7 Filetransfer zwischen Host und Gast

Anlegen, formatieren und mount im Host

Auf bzw. in Ubuntu, dem Hostbetriebssystem, wird eine Partiton wie folgt angelegt, formatiert und gemountet.

```
sudo -s
cd /root
dd if=/dev/zero of=floppy bs=1440K count=1
mkfs.vfat floppy
mkdir /mnt/tmp
mount -t vfat -o loop /root/floppy /mnt/tmp
```

Listing C.7: Shell: Transferpartition erstellen

Mount im Gast

Die entsprechenden Files in `/mnt/tmp` kopieren und mit `umount /mnt/tmp` unmounten. Zum Start der FreeDos Emulation mit der Transferpartition wird `qemu DOS.img -fda /root/floppy` eingegeben. Analog erfolgt der Start für muLinux mit `qemu mu.img -fda /root/floppy`. Direkt von der CD kann auch gestartet werden (siehe Listing C.8).

```
qemu -hda mu.img -hdb /root/floppy -cdrom mulinux-14r0.iso -boot d -fda /
     root/floppy
```

Listing C.8: Shell: Start von muLinux mit Transferpartition direkt von CD

In muLinux (das Gastbetriebssystem) kann nun das entsprechende Laufwerk zum Transfer mit `mount /dev/fd0 /a` gemountet werden.

Literatur

[Agr92] William W. Agresti. „Session 1 Summary: The Experimental Paradigm in Software Engineering“. In: *Experimental Software Engineering Issues*. Hrsg. von H. Dieter Rombach, Victor R. Basili und Richard W. Selby. Bd. 706. Lecture Notes in Computer Science. Springer, 1992, S. 33–37. ISBN: 3-540-57092-6.

[AHU74] Alfred V. Aho, John E. Hopcroft und Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AIo8] Syed Ahsona und Mohammad Ilya. „Smart Phones: Operating Systems“. In: *Encyclopedia of Wireless and Mobile Communications*. 2008, S. 1073 –1085.

[AK+08] Ashok Argent-Katwala u. a. „Safety and Response-Time Analysis of an Automotive Accident Assistance Service“. In: *ISoLA*. Hrsg. von Tiziana Margaria und Bernhard Steffen. Bd. 17. Communications in Computer and Information Science. Springer, 2008, S. 191–205. ISBN: 978-3-540-88478-1.

[Alb09] Sönke Albers, Hrsg. *Methodik der empirischen Forschung*. 3., überarb. und erw. Aufl. Wiesbaden: Gabler, 2009. XVII, 445. ISBN: 978-3-8349-0469-0.

[ALF00] Lörinc Antoni, Régis Leveugle und Béla Fehér. „Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes“. In: *DFT '00: Proceedings of the 15th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems*. Washington, DC, USA: IEEE Computer Society, 2000, S. 405–413. ISBN: 0-7695-0719-0.

[Ali+04] Aravindh Anantaraman Ali u. a. „EDF-DVS Scheduling on the IBM Embedded PowerPC 405LP“. In: *In Proceedings of the IBM Pa=c 2 Conference*. 2004.

[Alt+10] E. Altman u. a. „Observations on tuning a Java enterprise application for performance and scalability“. In: *IBM Journal of Research and Development* 54 (2010), 2:1 –2:12.

[And09] Michele Andreoli. „MuLinux Home Site: muLinux version 14r0“. http://www.micheleandreoli.it/mulinux/. zuletzt abgerufen am 03.08.2010. März 2009.

[And10a] Android Open Source Project (AOSP). *Android Emulator | Android Developers*. http://developer.android.com/guide/developing/tools/emulator.html. Aug. 2010.

[And10b] Android Open Source Project (AOSP). *Get Android Source Code | Android Open Source*. http://source.android.com/source/download.html. Aug. 2010.

[And10c] Android Open Source Project (AOSP). *Using Repo and Git | Android Open Source*. http://source.android.com/source/git-repo.html. Aug. 2010.

[Ank+06] Mihael Ankerst u. a. „OPTICS: Ordering Points To Identify the Clustering Structure.“ In: *SIGMOD Conference*. Hrsg. von Alex Delis, Christos Faloutsos und Shahram Ghandeharizadeh. ACM Press, 31. März 2006, S. 49–60. ISBN: 1-58113-084-8.

[AP10] A. AL Abdullatif und R. J. Pooley. „UML-JMT: A Tool for Evaluating Performance Requirements“. In: *Engineering of Computer-Based Systems, IEEE International Conference on the* 0 (2010), S. 215–225.

[Arl+90] Jean Arlat u. a. „Fault Injection for Dependability Validation: A Methodology and Some Applications“. In: *IEEE Trans. Softw. Eng.* 16.2 (1990), S. 166–182. ISSN: 0098-5589.

[AS00] Leonardus Budiman Arief und Neil Speirs. „A UML tool for an automatic generation of simulation programs“. In: *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*. New York, NY, USA: ACM, 2000, S. 71–76. ISBN: 1-58113-195-X.

[Asa+06] Krste Asanovic u. a. *The Landscape of Parallel Computing Research: A View from Berkeley*. Techn. Ber. UCB/EECS-2006-183. EECS Department, University of California, Berkeley, Dez. 2006.

[Ass03] Walter Assenmacher. *Deskriptive Statistik*. Springer-Verlag Berlin Heidelberg New York, 2003.

[Ast03] Dave Astels. *Test Driven development: A Practical Guide*. Prentice Hall Professional Technical Reference, 2003. ISBN: 0131016490.

[Aye+08] Nathaniel Ayewah u. a. „Using Static Analysis to Find Bugs“. In: *IEEE Software* 25.5 (2008), S. 22–29. ISSN: 0740-7459.

[Bö5] Oliver Böhm. *Aspektorientierte Programmierung mit AspectJ 5*. Heidelberg: dpunkt.verlag, 2005. ISBN: 10 3-89864-330-1.

[BA04] Kent Beck und Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004. ISBN: 0321278658.

[BA08] Sorav Bansal und Alex Aiken. „Binary Translation Using Peephole Superoptimizers“. In: *OSDI*. Hrsg. von Richard Draves und Robbert van Renesse. USENIX Association, 2008, S. 177–192. ISBN: 978-1-931971-65-2.

[Bac+08] Klaus Backhaus u. a. *Multivariate Analysemethoden. Eine anwendungsorientierte Einführung*. Berlin: Springer, 2008.

[Bac+97] Peter Bach u. a. „Building the 4 Processor SB-PRAM Prototype“. In: *HICSS '97: Proceedings of the 30th Hawaii International Conference on System Sciences*. Washington, DC, USA: IEEE Computer Society, 1997, S. 14. ISBN: 0-8186-7743-0.

[Bal+04] Simonetta Balsamo u. a. „Model-Based Performance Prediction in Software Development: A Survey“. In: *IEEE Trans. Softw. Eng.* 30.5 (2004), S. 295–310. ISSN: 0098-5589.

[Bal+10] Thomas Ball u. a. „Predictable and Progressive Testing of Multi-threaded Code“. In: *IEEE Software* 99.PrePrints (2010). ISSN: 0740-7459.

[Bar+03] Paul Barham u. a. „Xen and the art of virtualization“. In: *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2003, S. 164–177. ISBN: 1-58113-757-5.

[Bar04a] Scott Barber. *Beyond Performance Testing*. IBM DeveloperWorks, Rational Technical Library, `www-128.ibm.com/developerworks/rational/library/4169.html`. 2004.

[Bar04b] Scott Barber. „Creating Effective Load Models for Performance Testing with Incomplete Empirical Data“. In: *WSE '04: Proceedings of the Web Site Evolution, Sixth IEEE International Workshop*. Washington, DC, USA: IEEE Computer Society, 2004, S. 51–59. ISBN: 0-7695-2224-6.

[Bar06] Daniel Bartholomew. „QEMU: a multihost, multitarget emulator“. In: *Linux J.* 2006.145 (2006), S. 3. ISSN: 1075-3583.

[Bas+05] Victor R. Basili u. a. *Foundations of Empirical Software Engineering. The Legacy of Victor R. Basili*. Hrsg. von Barry Boehm, Hans Dieter Rombach und Marvin V. Zelkowitz. Springer, 2005.

[Bas85] Farokh Bastani. „On the Uncertainty in the Correctness of Computer Programs“. In: *IEEE Transactions on Software Engineering* 11 (1985), S. 857–864. ISSN: 0098-5589.

[BB81] Arthur W. Burks und Alice R. Burks. „The ENIAC: First General-purpose Electronic Computer“. In: *IEEE Annals of the History of Computing* 3.4 (Okt. 1981), S. 310–399.

[BCK04] Marco Burkschat, Erhard Cramer und Udo Kamps. *Beschreibende Statistik: Grundlegende Methoden*. Springer-Verlag, Berlin Heidelberg, 2004.

[BCM10] Luca Berardinelli, Vittorio Cortellessa und Antinisca Di Marco. „Performance Modeling and Analysis of Context-Aware Mobile Software Systems“. In: *FASE*. Hrsg. von David S. Rosenblum und Gabriele Taentzer. Bd. 6013. Lecture Notes in Computer Science. Springer, 2010, S. 353–367. ISBN: 978-3-642-12028-2.

[BD06] Kristof Beyls und Erik H. D'Hollander. „Intermediately executed code is the key to find refactorings that improve temporal data locality“. In: *CF '06: Proceedings of the 3rd conference on Computing frontiers*. New York, NY, USA: ACM, 2006, S. 373–382. ISBN: 1-59593-302-6.

[Bec+01] Kent Beck u. a. *Manifesto for Agile Software Development*. 2001.

[Bec02] Kent Beck. *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321146530.

[Bec+10] Steffen Becker u. a. „Reverse Engineering Component Models for Quality Predictions“. In: *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*. 2010.

[Bec10a] Daniel Becker. „Timestamp Synchronization of Concurrent Events“. ISBN 978-3-89336-625-5. Diss. IAS Series 4, Forschungszentrum Jülich: RWTH Aachen University, 2010.

[Bec10b] Steffen Becker. „The palladio component model“. In: *WOSP/SIPEW '10: Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*. San Jose, California, USA: ACM, 2010, S. 257–258. ISBN: 978-1-60558-563-5.

[Bel05] Fabrice Bellard. „QEMU, a fast and portable dynamic translator“. In: *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005, S. 41.

[Bel08] Fabrice Bellard. „QEMU License“. http://www.qemu.org/license.html. zuletzt abgerufen am 27.01.2010. Juli 2008.

[Bel09] Fabrice Bellard. „QEMU Status“. http://www.qemu.org/status.html. zuletzt abgerufen am 27.01.2010. März 2009.

[Bem57] R. Bemer. „How to consider a computer“. In: *Automatic Control Magazine* 3 (1957), S. 66–69.

[BEP96] Klaus Backhaus, Bernd Erichson und Wulff Plinke. *Multivariate Analysemethoden: eine anwendungsorientierte Einführung*. Springer, 1996.

[Ber+10] Tom Bergan u. a. „CoreDet: a compiler and runtime system for deterministic multithreaded execution“. In: *SIGARCH Comput. Archit. News* 38.1 (2010), S. 53–64. ISSN: 0163-5964.

[BFH03] Fran Berman, Geoffrey Fox und Tony Hey. *Grid Computing: Making The Global Infrastructure a Reality*. John Wiley & Sons, Apr. 2003. ISBN: 0470853190.

[BH08] Eric Bodden und Klaus Havelund. „Racer: Effective Race Detection Using AspectJ“. In: *International Symposium on Software Testing and Analysis (ISSTA), Seattle, WA*. ACM, 2008, S. 155–165.

[BHH05] George E. P. Box, J. Stuart Hunter und William G. Hunter. *Statistics for experimenters : design, innovation, and discovery*. Wiley-Interscience, 2005. ISBN: 9780471718130.

[Bis07] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. 1. Aufl. Springer, 2007. ISBN: 0387310738.

[BKR09] Steffen Becker, Heiko Koziolek und Ralf Reussner. „The Palladio component model for model-driven performance prediction“. In: *Journal of Systems and Software* 82.1 (2009), S. 3–22.

[BKS00] Markus M. Breunig, Hans-Peter Kriegel und Jörg Sander. „Fast Hierarchical Clustering Based on Compressed Data and OPTICS“. In: *PKDD '00: Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*. London, UK: Springer-Verlag, 2000, S. 232–242. ISBN: 3-540-41066-X.

[BL73] David Elliot Bell und Leonard J. LaPadula. *Secure Computer Systems: Mathematical Foundations*. Techn. Ber. MTR-2547, Vol. 1. Bedford, MA: MITRE Corp., 1973.

[Ble89] G. E. Blelloch. „Scans as Primitive Parallel Operations“. In: *IEEE Trans. Comput.* 38.11 (1989), S. 1526–1538. ISSN: 0018-9340.

[Bli+08] Michal Bližňák u. a. „Virtualization as a teaching tool“. In: *DIWEB'08: Proceedings of the 8th WSEAS international conference on Distance learning and web engineering*. Stevens Point, Wisconsin, USA: World Scientific, Engineering Academy und Society (WSEAS), 2008, S. 214–217. ISBN: 978-960-474-005-5.

[Bloo8] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008. ISBN: 0321356683, 9780321356680.

[BLR02] Chandrasekhar Boyapati, Robert Lee und Martin Rinard. „Ownership types for safe programming: preventing data races and deadlocks“. In: *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. Bd. 37. 11. New York, NY, USA: ACM Press, Nov. 2002, S. 211–230. ISBN: 1581134711.

[Blu+98] Thom Blum u. a. *Writing a Web Crawler in the Java Programming Language*. `http://java.sun.com/developer/technicalArticles/ThirdParty/WebCrawler/`. Jan. 1998.

[Boe07] Barry W. Boehm. *Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research*. Hrsg. von Richard W. Selby. John Wiley & Son, 2007.

[Boe+78] Barry W. Boehm u. a. *Characteristics of Software Quality*. TRW series of software technology, 1978.

[Boe86] Barry Boehm. „A spiral model of software development and enhancement“. In: *SIGSOFT Softw. Eng. Notes* 11.4 (1986), S. 14–24. ISSN: 0163-5948.

[Bolo4] Georg Bol. *Deskriptive Statistik*. ger. München ; Wien: Oldenbourg, 2004, II, 218 S. ISBN: 3-486-57612-7, 978-3-486-57612-2.

[Bol+06] Gunter Bolch u. a. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. 2nd Edition. Wiley-Blackwell, Mai 2006. ISBN: 0471565253.

[Bop89] R. B. Boppana. „Optimal separations between concurrent-write parallel machines“. In: *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing*. Hrsg. von ACM. ACM order no. 508840. New York, NY, USA: ACM, 1989, S. 320–326. ISBN: 0-89791-307-8.

[Boro8] Günter Born. *Der Eee PC*. Hrsg. von Günter Born. Markt und Technik, 2008.

[Boro9] Günter Born. *Das Netbook - mit Linux: Internet überall, Büro in der Hosentasche, Vergnügen jederzeit*. Markt und Technik, 2009.

[Bos+08] Herbert Bos u. a. „Future threats to future trust“. In: *Proceedings of the "Future of Trust in Computing" Conference*. Berlin, Germany, 2008.

[Bou+04] Craig Boutilier u. a. „CP-nets: A Tool for Representing and Reasoning with Conditional Ceteris Paribus Preference Statements“. In: *JOURNAL OF ARTIFICIAL INTELLIGENCE RESEARCH* 21 (2004), S. 135–191.

[BP02] Don Box und Ted Pattison. *Essential .NET: The Common Language Runtime*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201734117.

[BP06] Benedikt Mas y Parareda und Markus Pizka. *Reengineering Web-basierter und anderer junger Legacy-Systeme — Erfahrungsbericht*. Techn. Ber. Empirical Study for VSEK – www.software-kompetenz.de. Garching, Germany: itestra GmbH, Okt. 2006.

[Bra06] Dana Brand. *Integration of Runtime Profiling and Static Code Analysis in Linux*. Dez. 2006.

[Bre74] Richard P. Brent. „The parallel evaluation of general arithmetic expressions“. In: *Journal of the ACM* 21 (1974), S. 201–206.

[Bro03] Peter J. Brown. „Software portability“. In: *Encyclopedia of Computer Science*. Chichester, UK: John Wiley und Sons Ltd., 2003, S. 1633–1634. ISBN: 0-470-86412-5.

[BS01] Simonetta Balsamo und Marta Simeoni. „Deriving Performance Models from Software Architecture Specifications“. In: *In: Proceedings of the European Simulation Multiconference, Analytical and Stochastic Modelling Techniques. (2001*. 2001, S. 6–9.

[BS76] John Bruno und Ravi Sethi. „Code Generation for a One-Register Machine“. In: *J. ACM* 23.3 (1976), S. 502–510. ISSN: 0004-5411.

[BSH86] V R Basili, R W Selby und D H Hutchens. „Experimentation in software engineering“. In: *IEEE Trans. Softw. Eng.* 12.7 (1986), S. 733–743. ISSN: 0098-5589.

[BST00] David F. Bacon, Robert E. Strom und Ashis Tarafdar. „Guava: A dialect of Java without data races“. In: *In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA*. ACM Press, 2000, S. 382–400.

[BW00] Birnbaum und Williams. „Physics and the information revolution“. In: *Physics Today* 53 (2000), S. 38–42.

[BW08a] Tiller Beauchamp und David Wes. *Dtrace: The reverse engineer's unexpected swiss army knife*. Techn. Ber. Blackhat Europe, 2008.

[BW08b] Wolf-Gideon Bleek und Henning Wolf. *Agile Softwareentwicklung: Werte, Konzepte und Methoden*. Dpunkt Verlag; 2008.

[CCG03] Brian Carrier, Brian D. Carrier und Joe Grand. „A Hardware-Based Memory Acquisition Procedure for Digital Investigations“. In: *Digital Investigation* 1 (2003), S. 2004.

[CG08] Michael Creel und William L. Goffe. „Multi-core CPUs, Clusters, and Grid Computing: A Tutorial“. In: *Comput. Econ.* 32.4 (2008), S. 353–382. ISSN: 0927-7099.

[Cha08] Tim Chartier. *Devastating Roundoff Error*. Techn. Ber. Davidson College, 2008.

[Che+09] Betty H. C. Cheng u. a. „Software Engineering for Self-Adaptive Systems: A Research Roadmap.“ In: *Software Engineering for Self-Adaptive Systems*. Hrsg. von Betty H. C. Cheng u. a. Bd. 5525. Lecture Notes in Computer Science. Springer, 13. Juni 2009, S. 1–26. ISBN: 978-3-642-02160-2.

[Cho+02] Jong-Deok Choi u. a. „Efficient and precise datarace detection for multithreaded object-oriented programs“. In: *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. New York, NY, USA: ACM, 2002, S. 258–269. ISBN: 1-58113-463-0.

[Cho+08] Youngjin Cho u. a. „System-level power estimation using an on-chip bus performance monitoring unit“. In: *ICCAD '08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*. San Jose, California: IEEE Press, 2008, S. 149–154. ISBN: 978-1-4244-2820-5.

[Cho56] N. Chomsky. „Three models for the description of language.“ In: *IRE Transactions on Information Theory* 2 (1956), S. 113–124.

[Chu+99] Lawrence Chung u. a. *Non-Functional Requirements in Software Engineering (THE KLUWER INTERNATIONAL SERIES IN SOFTWARE ENGINEERING Volume 5) (International Series in Software Engineering)*. Springer, Okt. 1999. ISBN: 0792386663.

[Cis06] Cisco Systems. *Cisco Security Advisory: Cisco Catalyst Memory Leak Vulnerability*. http://www.cisco.com/warp/public/707/cisco-sa-20001206-catalyst-memleak.shtml. Dez. 2006.

[CMG08] Olivier Constant, Wei Monin und Susanne Graf. „A model transformation tool for performance simulation of complex uml models“. In: *ICSE Companion '08: Companion of the 30th international conference on Software engineering*. Leipzig, Germany: ACM, 2008, S. 923–924. ISBN: 978-1-60558-079-1.

[Com06] Compuware. *Application Performance Management Survey*. Techn. Ber. Compuware, 2006.

[Con+09] Guogjing Cong u. a. „A Holistic Approach towards Automated Performance Analysis and Tuning“. In: *Euro-Par 2009 Parallel Processing*. Hrsg. von Henk Sips, Dick Epema und Hai-Xiang Lin. Bd. 5704. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, S. 33–44.

[Cor04] Mercury Interactive Corporation. *Load Testing to Predict Web Performance*. Techn. Ber. Mercury Interactive Corporation, 2004.

[Cor84] Inmos Corp. *Occam Programming Manual*. Prentice Hall Trade, 1984. ISBN: 0136292968.

[CP04] Anton Chuvakin Cyrus Peikar. *Kenne deinen Feind*. Deutsch. O'Reilly, ISBN 978-3-89721-376-0, 2004, S. 1–26.

[CP09] Alistair Croll und Sean Power. *Complete Web Monitoring Watching your visitors, performance, communities, and competitors*. O'Reilly Media, 2009.

[CPL09] Lawrence Chung und Julio Cesar Sampaio do Prado Leite. „On Non-Functional Requirements in Software Engineering". In: *Conceptual Modeling: Foundations and Applications*. Hrsg. von Alexander Borgida u. a. Bd. 5600. Lecture Notes in Computer Science. Springer, 2009, S. 363–379. ISBN: 978-3-642-02462-7.

[CR08] Matthew Caesar und Jennifer Rexford. „Building bug-tolerant routers with virtualization". In: *PRESTO '08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*. New York, NY, USA: ACM, 2008, S. 51–56. ISBN: 978-1-60558-181-1.

[CR72] Stephen A. Cook und Robert A. Reckhow. „Time-bounded random access machines". In: *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1972, S. 73–80.

[Cra+06] Jedidiah R. Crandall u. a. „Temporal search: detecting hidden malware timebombs with virtual machines". In: *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2006, S. 25–36. ISBN: 1-59593-451-0.

[CS02] Xuejun Chen und Martin Simons. „A Component Framework for Dynamic Reconfiguration of Distributed Systems". In: *CD '02: Proceedings of the IFIP/ACM Working Conference on Component Deployment*. London, UK: Springer-Verlag, 2002, S. 82–96. ISBN: 3-540-43847-5.

[CSL04] Bryan M. Cantrill, Michael W. Shapiro und Adam H. Leventhal. „Dynamic instrumentation of production systems". In: *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*. Boston, MA: USENIX Association, 2004, S. 2–2.

[Cul+93] David Culler u. a. „LogP: towards a realistic model of parallel computation". In: *SIGPLAN Not.* 28.7 (1993), S. 1–12. ISSN: 0362-1340.

[CZ89] R. Cole und O. Zajicek. „The APRAM: incorporating asynchrony into the PRAM model“. In: *SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 1989, S. 169–178. ISBN: 0-89791-323-X.

[DAK00] R. Dimpsey, R. Arora und K. Kuiper. „Java server performance: a case study of building efficient, scalable Jvms“. In: *IBM Syst. J.* 39.1 (2000), S. 151–174. ISSN: 0018-8670.

[Dav56] M. D. Davis. „A note on universal Turing machines“. In: *Automata Studies*. Princeton University Press, 1956.

[DBR02] Scott W. Devine, Edouard Bugnion und Mendel Rosenblum. „Virtualization system including a virtual machine monitor for a computer with a segmented architecture“. Pat. US 6397242 B1. 2002.

[Del09] Deloitte Research. Technology, Media and Telecommunications (TMT). 2009.

[Del99] Jean-Paul Delahaye. *Pi. Die Story*. Hrsg. von Jean-Paul Delahaye. Birkhäuser Basel, 1999.

[Des09] Don E. Descy. „Netbooks: Small but Powerful Friends“. In: *TechTrends: Linking Research and Practice to Improve Learning* 53.2 (2009), S. 9–10,

[DG08] Mark Dehus und Dirk Grunwald. „STORM: Simple Tool for Resource Management“. In: *LISA*. Hrsg. von Mario Obejas. USENIX Association, 2008, S. 109–119. ISBN: 978-1-931971-63-8.

[DKP02] R. Dementiev, M. Klein und W.J. Paul. „Performance of MP3D on the SB-PRAM prototype“. In: *Europar 2002*. Bd. 2400. LNCS. Paderborn, Germany: Springer, 2002.

[DR00] Patrick W. Dymond und Walter L. Ruzzo. „Parallel RAMs with owned global memory and deterministic context-free language recognition“. In: *J. ACM* 47.1 (2000), S. 16–45. ISSN: 0004-5411.

[DR08] Richard Draves und Robbert van Renesse, Hrsg. *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. USENIX Association, 2008. ISBN: 978-1-931971-65-2.

[DR86] Patrick W. Dymond und Walter L. Ruzzo. „Parallel RAMs with Owned Global Memory and Deterministic Context-Free Language Recognition (Extended Abstract)“. In: *ICALP '86: Proceedings of the 13th International Colloquium on Automata, Languages and Programming*. London, UK: Springer-Verlag, 1986, S. 95–104. ISBN: 3-540-16761-7.

[DR93] Joseph F. Dumas und Janice C. Redish. *A Practical Guide to Usability Testing*. Westport, CT, USA: Greenwood Publishing Group Inc., 1993. ISBN: 089391990X.

[Dra+06] Dirk Draheim u. a. „Realistic Load Testing of Web Applications“. In: *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2006, S. 57–70. ISBN: 0-7695-2536-9.

[Dro91] Günther Drosdowski. *Duden Rechtschreibung der Deutschen Sprache*. 20., völlig neubearb. u. erw. Aufl. Mannheim [u.a.]: Dudenverl., 1991. ISBN: 3-411-04010-6.

[DSP10] Salvatore Distefano, Marco Scarpa und Antonio Puliafito. „From UML to Petri Nets: the PCM-Based Methodology“. In: *IEEE Transactions on Software Engineering* 99.PrePrints (2010). ISSN: 0098-5589.

[DSZ10] Jiaqing Du, Nipun Sehrawat und Willy Zwaenepoel. „Performance Profiling in a Virtualized Environment“. In: *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing*. Boston, Massachusetts, USA, 2010.

[EA03] Dawson Engler und Ken Ashcraft. „RacerX: effective, static detection of race conditions and deadlocks“. In: *SIGOPS Oper. Syst. Rev.* 37.5 (2003), S. 237–252. ISSN: 0163-5980.

[EC00] Josef EICHINGER und Joachim CHARZINSKI. „MOBILE TELEPHONE WITH MUSIC STORAGE“. Pat. WO/2001/043298. 2000.

[Ede+01] Orit Edelstein u. a. „Multithreaded Java program test generation“. In: *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*. New York, NY, USA: ACM, 2001, S. 181. ISBN: 1-58113-359-6.

[Ede+03] Orit Edelstein u. a. „Framework for testing multi-threaded Java programs“. In: *Concurrency and Computation: Practice and Experience* 15.3-5 (2003), S. 485–499.

[Edm91] Jeff Edmonds. „Lower Bounds with Smaller Domain Size On Concurrent Write Parallel Machines“. In: *Structure in Complexity Theory Conference*. 1991, S. 322–331.

[Edw03] Lewin Edwards. *Embedded system design on a shoestring : achieving high performance with a limited budget*. Amsterdam: Newnes, 2003. ISBN: 0750676094.

[Ei90] Institute O. Electrical und Electronics E. (ieee). *IEEE standard glossary of software engineering terminology*. Techn. Ber. Electrical, Institute O. und (ieee), Electronics E., 1990.

[EKT08] Hans-Friedrich Eckey, Reinhold Kosfeld und Matthias Türck. *Deskriptive Statistik. Grundlagen - Methoden - Beispiele*. 5., überarb. Aufl. Lehrbuch. Wiesbaden: Gabler, 2008. XXV, 283. ISBN: 978-3-8349-0859-9.

[Ell+01] John Ellson u. a. „Graphviz – open source graph drawing tools“. In: *Lecture Notes in Computer Science*. Springer-Verlag, 2001, S. 483–484.

[EM10] Javier Esparza und Rupak Majumdar, Hrsg. *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Bd. 6015. Lecture Notes in Computer Science. Springer, 2010. ISBN: 978-3-642-12001-5.

[Eng05] Michael Engel. „Advancing Operating Systems via Aspect-Oriented Programming“. Dissertation. Universität Siegen, Fachbereich Elektrotechnik und Informatik, Juli 2005.

[ES00] Martin Ester und Jörg Sander. *Knowledge Discovery in Databases: Techniken und Anwendungen*. Springer, 2000. ISBN: 3540673288.

[Esp+06] Huáscar Espinoza u. a. „Annotating UML Models with Non-functional Properties for Quantitative Analysis“. In: *Satellite Events at the MoDELS 2005 Conference*. Bd. Volume 3844/2006. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, S. 79–90.

[Est+96] Martin Ester u. a. „A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise“. In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*. Hrsg. von Evangelos Simoudis, Jiawei Han und Usama M. Fayyad. AAAI Press, 1996, S. 226–231.

[Fic+89] Faith Ellen Fich u. a. „Lower Bounds for Parallel Random Access Machines with Read Only Memory“. In: *Information and Computation* 83 (1989), S. 234–244.

[FKL07] Ludwig Fahrmeir, Thomas Kneib und Stefan Lang. *Regression. Modelle, Methoden und Anwendungen*. Springer-Verlag Berlin Heidelberg, 2007.

[Flo67] Robert W. Floyd. „Assigning meanings to programs“. In: *Proc. Sympos. Appl. Math., Vol. XIX*. Providence, R.I.: Amer. Math. Soc., 1967, S. 19–32.

[Fly72] Michael J. Flynn. „Some Computer Organizations and Their Effectiveness“. In: *IEEE Trans. Comput.* C-21 (1972), S. 948+.

[Fok+09] Marios Fokaefs u. a. „Decomposing object-oriented class modules using an agglomerative clustering technique“. In: *ICSM*. IEEE, 2009, S. 93–101.

[For+97] Arno Formella u. a. „Scientific Application on the SB-PRAM“. In: *Proceedings of the International Conference 'Multiscale Phenomena and Their Simulation'*. World Scientific Publishers, 1997.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein und Joe D. Warren. „The Program Dependence Graph and Its Use in Optimization.“ In: *ACM Trans. Program. Lang. Syst.* 9.3 (1987), S. 319–349.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999. ISBN: 0-201-48567-2.

[FP97] Norman E. Fenton und Shari Lawrence Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. Second. PWS Publishing Company, 1997.

[FPSS96] U. M. Fayyad, G. Piatetsky-Shapiro und P. Smyth. „Knowledge discovery and data mining : Towards a unifying framework“. In: *Proceedings of the 2nd international conference on Knowledge Discovery and Data mining (KDD'96)*. AAAI Press, Aug. 1996, S. 82–88.

[Fra03] Gerhard Franken. *DOS GE-PACKT*. MITP, 2003.

[Fri07] Michael Friedewald. *Vom Rechenautomaten zum elektronischen Medium: Eine kurze Geschichte des interaktiven Computers*. Hrsg. von Simone Kimpeler, Michael Mangold und Wolfgang Schweiger. Springer, 2007.

[FRW84] Faith E. Fich, Prabhakar L. Ragde und Avi Wigderson. „Relations between concurrent-write models of parallel computation“. In: *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 1984, S. 179–189. ISBN: 0-89791-143-1.

[FT96] William Frakes und Carol Terry. „Software reuse: metrics and models“. In: *ACM Comput. Surv.* 28.2 (1996), S. 415–435. ISSN: 0360-0300.

[Fuc07] Stefan K. Fuchs. „SAP NetWeaver in der Praxis – Wie gut bewährt sich der Technologie-Stack in der praktischen Arbeit?“ In: *Informatik-Spektrum* 30.6 (2007), S. 428–433. ISSN: 0170-6012.

[Fuc92] Norbert Fuchs. „Software Engineering Still on the Way to an Engineering Discipline“. In: *Experimental Software Engineering Issues*. Hrsg. von H. Dieter Rombach, Victor R. Basili und Richard W. Selby. Bd. 706. Lecture Notes in Computer Science. Springer, 1992, S. 19–22. ISBN: 3-540-57092-6.

[FW78] Steven Fortune und James Wyllie. „Parallelism in random access machines“. In: *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1978, S. 114–118.

[FW89] F. E. Fich und A. Wigderson. „Towards understanding exclusive read“. In: *SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 1989, S. 76–82. ISBN: 0-89791-323-X.

[FW90] Faith E. Fich und Avi Wigderson. „Toward understanding exclusive read“. In: *SIAM J. Comput.* 19.4 (1990), S. 718–727. ISSN: 0097-5397.

[Gam+95] Erich Gamma u. a. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995. ISBN: 978-0-201-63361-0.

[Gar+98] S. Garg u. a. „A Methodology for Detection and Estimation of Software Aging“. In: *Software Reliability Engineering, International Symposium on* 0 (1998), S. 283. ISSN: 1071-9458.

[GBB02] Kenny C. Gross, Vatsal Bhardwaj und Randy Bickford. „Proactive Detection of Software Aging Mechanisms in Performance Critical Computers“. In: *SEW '02: Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*. Washington, DC, USA: IEEE Computer Society, 2002, S. 17. ISBN: 0-7695-1855-9.

[GC07] Jacques Goupy und Lee Creighton. *Introduction to Design of Experiments with JMP Examples, Third Edition*. SAS Publishing, 2007. ISBN: 1599944227, 9781599944227.

[GDS09] Tilo Gockel, Rüdiger Dillmann und Joachim Schröder. „Puppy Linux auf dem Embedded-PC MicroClient Jr./Sr.“ In: *Embedded Linux Das Praxisbuch*. Springer, 2009. Kap. 6, S. 101 –107.

[Gei+10] Markus Geimer u. a. „Recent Developments in the Scalasca Toolset“. In: *Tools for High Performance Computing 2009*. Hrsg. von Matthias S. Müller u. a. 3rd International Workshop on Parallel Tools for High Performance Computing. ZIH. Dresden: Springer, 2010, S. 39–51. ISBN: 978-3-642-11260-7.

[Ger+01] Martin Gergeleit u. a. *A Monitoring-based Approach to Object-Oriented Real-Time Computing*. 2001.

[GF08] Jim Gray und Bob Fitzgerald. „Flash Disk Opportunity for Server Applications“. In: *Queue* 6.4 (2008), S. 18–23. ISSN: 1542-7730.

[GH94] Stephen Gilmore und Jane Hillston. „The PEPA workbench: a tool to support a process algebra-based approach to performance modelling“. In: *Proceedings of the 7th international conference on Computer performance evaluation : modelling techniques and tools*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1994, S. 353–368. ISBN: 3-540-58021-2.

[GHR95] Raymond Greenlaw, H. James Hoover und Walter L. Ruzzo. *Limits to parallel computation: P-completeness theory*. New York, NY, USA: Oxford University Press, Inc., 1995. ISBN: 0-19-508591-4.

[Gib89] P. B. Gibbons. „A more practical PRAM model“. In: *SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 1989, S. 158–168. ISBN: 0-89791-323-X.

[Gil+04] Stephen Gilmore u. a. „Software performance modelling using PEPA nets“. In: *WOSP '04: Proceedings of the 4th international workshop on Software and performance*. New York, NY, USA: ACM, 2004, S. 13–23. ISBN: 1-58113-673-0.

[Gil93] Wolfgang K. Giloi. *Rechnerarchitektur*. Springer-Verlag, 1993.

[GK09] Abdelouahed Gherbi und Ferhat Khendek. „From UML/SPT models to schedulability analysis: approach and a prototype implementation using ATL“. In: *Automated Software Engineering* 16 (3 2009), S. 387–414. ISSN: 0928-8910.

[GLR81] Allan Gottlieb, B. D. Lubachevsky und Larry Rudolph. „Coordinating large numbers of processors“. In: *international Conference on Parallel Processing*. 1981.

[GMT08] Michael Grottke, Rivalino Matias und Kishor S. Trivedi. „The fundamentals of software aging“. In: *Proceedings of the IEEE Workshop on Software Aging and Rejuvenation*. Seattle, WA, 2008.

[Gol78] Leslie M. Goldschlager. „A unified approach to models of synchronous parallel machines“. In: *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1978, S. 89–94.

[Goo+05] Tom Goovaerts u. a. „Assessment of Palm OS Susceptibility to Malicious Code Threats“. In: *Communications and Multimedia Security*. Hrsg. von Jana Dittmann, Stefan Katzenbeisser und Andreas Uhl. Bd. 3677. Lecture Notes in Computer Science. Springer, 2005, S. 240–249. ISBN: 3-540-28791-4.

[Gos+05] James Gosling u. a. *Java(TM) Language Specification, The (3rd Edition)*. Addison-Wesley Professional, 2005. ISBN: 0321246780.

[Gou09] Brian Gough. *GNU Scientific Library Reference Manual - Third Edition*. Network Theory Ltd., 2009. ISBN: 0954612078, 9780954612078.

[GR88a] N. H. Gehani und W. D. Roome. „Concurrent C++: concurrent programming with class(es)“. In: *Softw. Pract. Exper.* 18.12 (1988), S. 1157–1177. ISSN: 0038-0644.

[GR88b] Narain H. Gehani und William D. Roome. „Rendezvous Facilities: Concurrent C and the Ada Language“. In: *IEEE Trans. Software Eng.* 14.11 (1988), S. 1546–1553.

[GR89] Narain Gehani und William D. Roome. *The concurrent C programming language*. Summit, NJ, USA: Silicon Press, 1989. ISBN: 0-929306-00-7.

[Gra86] Jim Gray. „Why Do Computers Stop and What Can Be Done About It?“ In: *Symposium on Reliability in Distributed Software and Database Systems*. 1986, S. 3–12.

[Gro09] Object Management Group. *UML 2.2*. http://www.omg.org/spec/UML/2.2/. zuletzt zugegriffen 18. Februar 2010. Feb. 2009.

[Gro10] Object M. Group. *OMG Unified Modeling LanguageTM (OMG UML), Infrastructure Version 2.3*. Techn. Ber. 2010.

[GRR95] T. Grün, T. Rauber und J. Röhrig. „The Programming Environment of the SB-PRAM“. In: *In Proc. 7th IASTED/ISMM Int.l Conf. on Parallel and Distributed Computing and Systems, Washington DC*. Acta Press, 1995, S. 504–509.

[Gup+05] Diwaker Gupta u. a. „To infinity and beyond: time warped network emulation“. In: *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2005, S. 1–2. ISBN: 1-59593-079-5.

[HA09] Sharon P. Hall und Eric Anderson. „Operating systems for mobile computing“. In: *J. Comput. Small Coll.* 25.2 (2009), S. 64–71. ISSN: 1937-4771.

[HA93] Carl S. Hartzman und Charles F. Austin. „Maintenance productivity: observations based on an experience in a large system environment“. In: *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*. Toronto, Ontario, Canada: IBM Press, 1993, S. 138–170.

[Hab08] Irfan Habib. „Virtualization with KVM“. In: *Linux J.* 2008.166 (2008), S. 8. ISSN: 1075-3583.

[Hag+94] T. Hagerup u. a. „FORK - A High-Level Language for PRAMs“. In: *Future Generation Computer Systems*. Springer Verlag, 1994, S. 304–320.

[Ham06] Moritz Hammer. *CMC model checker*. `http://www.pst.ifi.lmu.de/~hammer/cmc/`. März 2006.

[Ham09] Moritz Hammer. „How To Touch a Running System - Reconfiguration of Stateful Components.“ Diss. Ludwig-Maximilians Universität München, 2009.

[Han75a] Per Brinch Hansen. „The Programming Language Concurrent Pascal“. In: *IEEE Trans. Software Eng.* 1.2 (1975), S. 199–207.

[Han75b] Per Brinch Hansen. „The purpose of concurrent Pascal“. In: *Proceedings of the international conference on Reliable software*. New York, NY, USA: ACM, 1975, S. 305–309.

[Hap+10] Jens Happe u. a. „A Prediction Model for Software Performance in Symmetric Multiprocessing Environments“. In: *Quantitative Evaluation of Systems, International Conference on* 0 (2010), S. 59–68.

[Hay08] Simon Haykin. *Neural Networks and Learning Machines (3rd Edition)*. 3. Aufl. Prentice Hall, 2008. ISBN: 0131471392.

[HBK06] Jim Held, Jerry Bautista und Sean Koehl. *From a Few Core to Many: A Tera-scale Computing Research Overview*. Techn. Ber. Intel, 2006.

[HC01] George T. Heineman und William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together (ACM Press)*. Addison-Wesley Professional, Juni 2001. ISBN: 0201704854.

[Hel02] Sumi Helal. „Pervasive Java, Part II“. In: *IEEE Pervasive Computing* 1 (2002), S. 85–89. ISSN: 1536-1268.

[Hen+04] Andreas Hennig u. a. „Instant Performance Prototyping of EJB/J2EE Applications - A car rental example “. In: *Proceedings des 5. Workshop des Arbeitskreis PEAK, Munich*. 2004.

[Her00] Ulrich Hertrampf. „Quantencomputer - Aktuelles Schlagwort“. In: *Informatik Spektrum* 23.5 (2000), S. 322–324.

[Het10] Thomas Hettel. „Model Round-Trip Engineering“. Diss. Queensland University of Technology, 2010.

[Het88] Bill Hetzel. *The complete guide to software testing (2nd ed.)* Wellesley, MA, USA: QED Information Sciences, Inc., 1988. ISBN: 0-89435-242-3.

[HFT] Jim Hall und FreeDOS-Team. *FreeDOS | The FreeDOS Project*. http://www.freedos.org/. zuletzt abgerufen am 04.08.2010.

[HG06] Bruno Harbulot und John R. Gurd. „A join point for loops in AspectJ“. In: *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2006, S. 63–74. ISBN: 1-59593-300-X.

[HH04a] Richard M. Heiberger und Burt Holland. *Statistical Analysis and Data Display: An Intermediate Course with Examples in S-Plus, R, and SAS*. Springer Texts in Statistics. ISBN 0-387-40270-5. Springer, 2004.

[HH04b] Erik Hilsdale und Jim Hugunin. „Advice weaving in AspectJ“. In: *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2004, S. 26–35. ISBN: 1-58113-842-3.

[Hic08] George D. Hickman. „An overview of Virtual Machine (VM) technology and its implementation in I.T. student labs at Utah Valley State College“. In: *J. Comput. Small Coll.* 23.6 (2008), S. 203–212. ISSN: 1937-4771.

[Hil96] Jane Hillston. *A compositional approach to performance modelling*. New York, NY, USA: Cambridge University Press, 1996. ISBN: 0-521-57189-8.

[Hil98] Stefan Hillenmayer. „Communication device for mobile operation having a telephone and notebook with Display“. Pat. 5719936. 1998.

[HJ88] Roger W. Hockney und C. R. Jesshope. *Parallel Computers 2: Architecture, Programming, and Algorithms*. Bristol, UK, UK: IOP Publishing Ltd., 1988. ISBN: 0852748124.

[HK10] Michal Hocko und Tomas Kalibera. „Reducing performance non-determinism via cache-aware page alconflocation strategies“. In: *WOSP/SIPEW '10: Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*. New York, NY, USA: ACM, 2010, S. 223–234. ISBN: 978-1-60558-563-5.

[HKP06] Jiawei Han, Micheline Kamber und Jian Pei. *Data Mining: Concepts and Techniques, Second Edition (The Morgan Kaufmann Series in Data Management Systems)*. 2. Aufl. Morgan Kaufmann, 2006. ISBN: 1558609016.

[HM08] Mark D. Hill und Michael R. Marty. „Amdahl's Law in the Multicore Era“. In: *Computer* 41.7 (2008), S. 33–38. ISSN: 0018-9162.

[Häm92] Pasi Hämäläinen. *PRAM EMULATOR - User's Manual*. 1992.

[Hol05] Andreas Holzinger. „Usability engineering methods for software developers“. In: *Commun. ACM* 48.1 (2005), S. 71–74. ISSN: 0001-0782.

[HP04] David Hovemeyer und William Pugh. „Finding concurrency bugs in Java“. In: *In Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*. 2004.

[HRW98] Roland HETTRICH, Karsten RUDOLPH und Stephan WINDISCH. „TELEPHONE OPTICAL DISPLAY DEVICE“. Pat. WO/1998/040996. 1998.

[HS01] Steven Homer und Alan L. Selman. *Computability and complexity theory*. New York, NY, USA: Springer-Verlag New York, Inc., 2001. ISBN: 0-387-95055-9.

[HTI97] Mei-Chen Hsueh, Timothy K. Tsai und Ravishankar K. Iyer. „Fault Injection Techniques and Tools“. In: *Computer* 30 (1997), S. 75–82. ISSN: 0018-9162.

[Hu+09] Yabin Hu u. a. „An Optimization Approach for QEMU“. In: *Information Science and Engineering, International Conference on* 0 (2009), S. 129–132.

[HW06a] Moritz Hammer und Michael Weber. „“To Store or Not To Store” Reloaded: Reclaiming Memory on Demand“. In: *The EASST Newsletter Volume* 14 (2006), S. 5–11.

[HW06b] Moritz Hammer und Michael Weber. „“To Store Or Not To Store Reloaded”: Reclaiming Memory On Demand“. In: *Formal Methods: Application and Technology (FMICS'2006)*. Hrsg. von Lubos Brim u. a. Bd. 4346. Lecture Notes in Computer Science. Springer-Verlag, 2006, S. 51–66.

[Iee] *IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610*. New York: IEEE, 1990.

[IEE90] IEEE. „IEEE Standard Glossary of Software Engineering Terminology“. In: *IEEE Std 610121990* 121990 (1990). Hrsg. von Editor.

[Int88] International Organization for Standardization. *Information processing: volume and file structure of CD-ROM for information interchange = Traitement de l'information: structure de volume et de fichier des disques optiques compacts a mémoire fixe (CD-ROM) destinés a l'échange d'information*. Corrected and reprinted. International standard; ISO 9660. 1988, S. x + 31.

[Int96] International Organization for Standardization. *Information technology - Open Systems Interconnection - Basic reference model: The basic model*. Geneva: ISO, 1996.

[Iva02] Ivelin Ivanov. „Parallel processing and parallel algorithms: theory and computation“. In: *SIGACT News* 33.4 (2002), S. 12–14. ISSN: 0163-5700.

[Jos03] José Luis Encarnaç ao. *Kunststück Innovation Praxisbeispiele aus der Fraunhofer-Gesellschaft*. Hrsg. von Hans-Jürgen Warnecke und Hans-Jörg Bullinger. Springer, 2003.

[JR78] S. C. Johnson und D. M. Ritchie. „Portability of C Programs and the UNIX System“. In: *Bell System Tech J* 57 (1978), S. 2021–2048.

[Kar+91] Anna R. Karlin u. a. „Empirical studies of competitve spinning for a shared-memory multiprocessor“. In: *SIGOPS Oper. Syst. Rev.* 25.5 (1991), S. 41–55. ISSN: 0163-5980.

[KB06] Johannes Köbler und Olaf Beyersdorff. *Von der Turingmaschine zum Quantencomputer - ein Gang durch die Geschichte der Komplexitätstheorie*. Springer Berlin Heidelberg, 2006.

[Kel92] Jörg Keller. „Zur Realisierbarkeit des PRAM Modelles“. Diss. Universität des Saarlandes Saarbrücken Postfach 151150 66041 Saarbrücken: Universität des Saarlandes, 1992.

[Ker+79] Brian W. Kernighan u. a. *Unix Seventh Edition Manual*. Bell Telephone Laboratories, Incorporated, 1979.

[Kes] Christoph W. Kessler. *SB-PRAM - Instruction Set Simulator System Software. Quick Reference Card*.

[Kes97] Christoph W. Kessler. *Practical Pram Programming with Fork95: A Tutorial*. Techn. Ber. 97-12. University of Trier, Department for Mathematics und Computer Science, Mai 1997.

[Keß+94] Christoph Keßler u. a. *Making FORK Practical*. Techn. Ber. Univ. Saarbrücken, 1994.

[KF05] Heiko Koziolek und Viktoria Firus. „Empirical Evaluation of Model-Based Performance Prediction Methods in Software Development“. In: *QoSA/-SOQUA*. Hrsg. von Ralf Reussner u. a. Bd. 3712. Lecture Notes in Computer Science. Springer, 2005, S. 188–202. ISBN: 3-540-29033-8.

[KF95] Nick Kolettis und N. Dudley Fulton. „Software Rejuvenation: Analysis, Module and Applications“. In: *FTCS '95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*. Washington, DC, USA: IEEE Computer Society, 1995, S. 381.

[Kic+02] Gregor Kiczales u. a. „Aspect-oriented programming“. US. Pat. US6467086. 2002.

[Kic+97] Gregor Kiczales u. a. „Aspect-oriented programming“. In: *Proceedings European Conference on Object-Oriented Programming*. Hrsg. von Mehmet Akşit und Satoshi Matsuoka. Bd. 1241. Berlin, Heidelberg, und New York: Springer-Verlag, 1997, S. 220–242.

[Kim+06] Hyojun Kim u. a. „Virtual-ROM: A New Demand Paging Component for RTOS and NAND Flash Memory Based Mobile Devices“. In: *ISCIS*. Hrsg. von Albert Levi u. a. Bd. 4263. Lecture Notes in Computer Science. Springer, 2006, S. 677–686. ISBN: 3-540-47242-8.

[KK00] Peter Kacsuk und Gabriele Kotsis, Hrsg. *Distributed and parallel systems: from instruction parallelism to cluster computing*. Norwell, MA, USA: Kluwer Academic Publishers, 2000. ISBN: 0-7923-7892-X.

[KKZ09] Hans-Peter Kriegel, Peer Kröger und Arthur Zimek. „Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering“. In: *ACM Trans. Knowl. Discov. Data* 3.1 (2009), S. 1–58. ISSN: 1556-4681.

[KL94] E. Karrels und E. Lusk. *Performance Analysis of MPI Programs*. Hrsg. von Jack J. Dongarra und Bernard Tourancheau. Philadelphia, PA, USA, 1994.

[Kle94] Trevor A. Kletz. *Learning from accidents*. Hrsg. von Butterworth-Heinemann. Oxford, 1994.

[KM06] Dieter Wieser Kurt Müller Markus Loepfe. „Optical simulations for fire detectors“. In: *Fire Safety Journal* 41 (2006), S. 274 –278.

[Koj] Richard Kojedzinszky. *Cflinux*. http://www.cflinux.hu/. zuletzt abgerufen am 04.08.2010.

[Koz04] Heiko Koziolek. *Empirische Bewertung von Performance-Analyseverfahren für Software-Architekturen*. Diplomarbeit. 2004.

[KPS90] Z. M. Kedem, K. V. Palem und P. G. Spirakis. „Efficient robust parallel computations“. In: *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1990, S. 138–148. ISBN: 0-89791-361-2.

[KR08] Heiko Koziolek und Ralf Reussner. „A Model Transformation from the Palladio Component Model to Layered Queueing Networks“. In: *SIPEW '08: Proceedings of the SPEC international workshop on Performance Evaluation*. Berlin, Heidelberg: Springer-Verlag, 2008, S. 58–78. ISBN: 978-3-540-69813-5.

[Kri04] J. Krinke. „Advanced Slicing of Sequential and Concurrent Programs“. In: *International Conference on Software Maintenance (ICSM 2004)*. Chicago, 2004, S. 464–468.

[Kri06] Diwakar Krishnamurthy. „A Synthetic Workload Generation Technique for Stress Testing Session-Based Systems“. In: *IEEE Trans. Softw. Eng.* 32.11 (2006). Member-Rolia, Jerome A. and Member-Majumdar, Shikharesh, S. 868–882. ISSN: 0098-5589.

[Kri07] Markus Krieser. *Dienstevirtualisierung mit Hilfe von VMware am Beispiel der Astrium GmbH*. Diplomarbeit. Ludwig-Maximilians-Universität München, Geschwister-Scholl-Platz 1, 80539 München, Deutschland, Sep. 2007.

[Kro00] Jason Kroll. „Games We Play: Commodore 64 Game Emulation“. In: *Linux J.* 72 (2000), S. 22. ISSN: 1075-3583.

[Kro07] Klaus Krogmann. „Reengineering of Software Component Models to Enable Architectural Quality of Service Predictions“. In: *Proceedings of the 12th International Workshop on Component Oriented Programming (WCOP 2007*. 2007.

[KRR96] Jörg Keller, Thomas Rauber und Bernd Rederlechner. „Conservative circuit simulation on shared-memory multiprocessors“. In: *PADS '96: Proceedings of the tenth workshop on Parallel and distributed simulation*. Washington, DC, USA: IEEE Computer Society, 1996, S. 126–134. ISBN: 0-8186-7539-X.

[KS07] Gerd Küveler und Dietrich Schwoch. *Schnittstellen zum Betriebssystem*. Springer, 2007.

[KS89] P. C. Kanellakis und A. A. Shvartsman. „Efficient parallel algorithms can be made robust“. In: *PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*. New York, NY, USA: ACM, 1989, S. 211–219. ISBN: 0-89791-326-4.

[KT07] Danny Kopec und Suzanne Tamang. „Failures in complex systems: case studies, causes, and possible remedies“. In: *SIGCSE Bull.* 39.2 (2007), S. 180–184. ISSN: 0097-8418.

[KT96] Christoph W. Kessler und Jesper Larsson Träff. „A library of basic PRAM algorithms and its implementation in FORK“. In: *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 1996, S. 193–195. ISBN: 0-89791-809-6.

[Kuc82] Ludek Kucera. „Parallel Computation and Conflicts in Memory Access“. In: *Inf. Process. Lett.* 14.2 (1982), S. 93–96.

[LA87] P. J. M. Laarhoven und E. H. L. Aarts, Hrsg. *Simulated annealing: theory and applications*. Norwell, MA, USA: Kluwer Academic Publishers, 1987.

[Lam78] Leslie Lamport. „Time, clocks, and the ordering of events in a distributed system“. In: *Commun. ACM* 21.7 (1978), S. 558–565. ISSN: 0001-0782.

[Lar03] Anders Henriksson Henrik Larsson. *A Definition of Round-Trip Engineering*. Techn. Ber. SE-581 83 Linköping, Sweden: Department of ComputerInformation Science, Linköpings Universitet, 2003.

[Law96] Kevin P. Lawton. „Bochs: A Portable PC Emulator for Unix/X“. In: *Linux J.* 29 (Sep. 1996), S. 7. ISSN: 1075-3583.

[LDH03] Cristina Videira Lopes, Paul Dourish und David H. „Beyond AOP: Toward Naturalistic Programming“. In: *in: OOPSLA'03 Special Track on Onward! Seeking New Paradigms & New Thinking (ACM*. Onward! Track, 2003, S. 34–43.

[Lee+08] Sang-Won Lee u. a. „A case for flash memory ssd in enterprise database applications“. In: *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2008, S. 1075–1086. ISBN: 978-1-60558-102-6.

[Len09] Burkhard Lenze. *Einführung in die Mathematik neuronaler Netze - Mit C- und Java-Anwendungsprogrammen im Internet*. Logos Verlag Berlin, 2009.

[LGMC04] Juan Pablo López-Grao, José Merseguer und Javier Campos. „From UML activity diagrams to Stochastic Petri nets: application to software performance engineering“. In: *SIGSOFT Softw. Eng. Notes* 29.1 (2004), S. 25–36. ISSN: 0163-5948.

[Lim94] Wayne C. Lim. „Effects of Reuse on Quality, Productivity, and Economics“. In: *IEEE Softw.* 11.5 (1994), S. 23–30. ISSN: 0740-7459.

[Lin+02] Christoph Lindemann u. a. „Performance analysis of time-enhanced UML diagrams based on stochastic processes“. In: *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*. New York, NY, USA: ACM, 2002, S. 25–34. ISBN: 1-58113-563-7.

[Lin05] Christian Lindig. „Random testing of C calling conventions“. In: *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*. New York, NY, USA: ACM, 2005, S. 3–12. ISBN: 1-59593-050-7.

[Liu+06] Chao Liu u. a. „GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis“. In: *In the Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD06*. ACM Press, 2006, S. 872–881.

[LL92] Michael C. Loui und David R. Luginbuhl. „The Complexity of On-Line Simulations Between Multidimensional Turing Machines and Random Access Machines“. In: *Mathematical Systems Theory* 25.4 (1992), S. 293–308.

[LMT10] Philippe Langlois, Matthieu Martel und Laurent Thévenoux. „Accuracy versus time: a case study with summation algorithms“. In: *PASCO '10: Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*. Grenoble, France: ACM, 2010, S. 121–130. ISBN: 978-1-4503-0067-4.

[Lon06] Karen D. Lontka. „ARRANGEMENT AND METHOD FOR COMMUNICATING WITH NOTIFICATION APPLIANCES“. Pat. 2009/0322526 A1. 2006.

[LPV81] Gavriela Freund Lev, Nicholas Pippenger und Leslie G. Valiant. „A Fast Parallel Algorithm for Routing in Permutation Networks“. In: *IEEE Trans. Computers* 30.2 (1981), S. 93–100.

[LS80] B. P. Lientz und E. B. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, 1980.

[LT93] Nancy G. Leveson und Clark S. Turner. „An Investigation of the Therac-25 Accidents“. In: *Computer* 26 (1993), S. 18–41. ISSN: 0018-9162.

[Luk+05] Chi-Keung Luk u. a. „Pin: building customized program analysis tools with dynamic instrumentation“. In: *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2005, S. 190–200. ISBN: 1-59593-056-6.

[LY09] Feida Lin und Weiguo Ye. „Operating System Battle in the Ecosystem of Smartphone Industry“. In: *IEEC '09: Proceedings of the 2009 International Symposium on Information Engineering and Electronic Commerce*. Washington, DC, USA: IEEE Computer Society, 2009, S. 617–621. ISBN: 978-0-7695-3686-6.

[LY89] Ming Li und Yaacov Yesha. „New lower bounds for parallel computation“. In: *J. ACM* 36.3 (1989), S. 671–680. ISSN: 0004-5411.

[LY99] Tim Lindholm und Frank Yellin. *Java Virtual Machine Specification*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0201432943.

[MA00] Daniel A. Menasce und Virgilio A. F. Almeida. *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning*. Prentice Hall PTR, 2000. ISBN: 0130863289.

[MA98] Daniel A. Menascé und Virgílio A. F. Almeida. *Capacity planning for Web performance: metrics, models, and methods*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998. ISBN: 0-13-693822-1.

[Mac67] J. B. MacQueen. „Some Methods for Classification and Analysis of MultiVariate Observations“. In: *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*. Hrsg. von L. M. Le Cam und J. Neyman. Bd. 1. University of California Press, 1967, S. 281–297.

[Man06] Florian Mangold. *Faktoranalyse*. Fortgeschrittenenpraktikum. 2006.

[Man07] Florian Mangold. *Performanzanalyse mit Hilfe künstlicher Varianz*. Diplomarbeit. Ludwig-Maximilians-Universität München, Geschwister-Scholl-Platz 1, 80539 München, Deutschland, Apr. 2007.

[Man+07] Florian Mangold u. a. „[DE] Verfahren zum rechnergestützten Ermitteln der Abhängigkeiten einer Vielzahl von Modulen eines technischen Systems, insbesondere eines Softwaresystems“. Schutzrecht 102007029133. Juni 2007.

[Man+08a] Florian Mangold u. a. „[DE] VERFAHREN ZUM RECHNERGESTÜTZTEN ERMITTELN DER ABHÄNGIGKEITEN EINER VIELZAHL VON MODULEN EINES TECHNISCHEN SYSTEMS, INSBESONDERE EINES SOFTWARESYSTEMS [EN] METHOD FOR THE COMPUTER-AIDED DETERMINATION OF THE DEPENDENCIES OF A PLURALITY OF MODULES OF A TECHNICAL SYSTEM, ESPECIALLY OF A SOFTWARE SYSTEM [FR] PROCÉDÉ DE DÉTERMINATION ASSISTÉE PAR ORDINATEUR DES DÉPENDANCES D'UNE PLURALITÉ DE MODULES D'UN SYSTÈME TECHNIQUE, NOTAMMENT D'UN SYSTÈME LOGICIEL“. European Patent 2008052641. März 2008.

[Man+08b] Florian Mangold u. a. „[DE] VERFAHREN ZUM RECHNERGESTÜTZTEN ERMITTELN DER ABHÄNGIGKEITEN EINER VIELZAHL VON MODULEN EINES TECHNISCHEN SYSTEMS, INSBESONDERE EINES SOFTWARESYSTEMS [EN] METHOD FOR THE COMPUTER-AIDED DETERMINATION OF THE DEPENDENCIES OF A PLURALITY OF MODULES OF A TECHNICAL SYSTEM, ESPECIALLY OF A SOFTWARE SYSTEM [FR] PROCÉDÉ DE DÉTERMINATION ASSISTÉE PAR OR-

DINATEUR DES DÉPENDANCES D'UNE PLURALITÉ DE MODULES D'UN SYSTÈME TECHNIQUE, NOTAMMENT D'UN SYSTÈME LOGICIEL". European Patent 08717395. März 2008.

[Man+08c] Florian Mangold u. a. „METHOD FOR THE COMPUTER-AIDED DETERMINATION OF THE DEPENDENCIES OF A PLURALITY OF MODULES OF A TECHNICAL SYSTEM, ESPECIALLY OF A SOFTWARE SYSTEM". International Patent WO/2008/113682. Sep. 2008.

[Man+08d] Florian Mangold u. a. „METHOD FOR THE COMPUTER-AIDED DETERMINATION OF THE DEPENDENCIES OF A PLURARITY OF MODULES OF A TECHNICAL SYSTEM, ESPECIALLY OF A SOFTWARE SYSTEM". United States Patent 20100088663. Aug. 2008.

[Man+08e] Florian Mangold u. a. „Skalierbare Performanzanalyse durch Prolongation". In: *Software Engineering*. Hrsg. von Korbinian Herrmann und Bernd Brügge. Bd. 121. LNI. GI, 2008. ISBN: 978-3-88579-215-4.

[Man10] Peter Mandl. „Grundkurs Betriebssysteme, 2., überarbeitete und aktualisierte Auflage Architekturen, Betriebsmittelverwaltung, Synchronisation, Prozesskommunikation". In: Springer, 2010. Kap. 5.

[Mar+10] Anne Martens u. a. „From monolithic to component-based performance evaluation of software architectures". In: *Empirical Software Engineering* (2010), S. 1–36. ISSN: 1382-3256.

[Mat93] Yuri V. Matiyasevich. *Hilbert's tenth problem*. Cambridge, MA, USA: MIT Press, 1993. ISBN: 0-262-13295-8.

[Maz10] Stanley Mazor. „Intel's 8086". In: *IEEE Annals of the History of Computing* 32 (2010), S. 75–79. ISSN: 1058-6180.

[Mce02] William Mcewan. „Virtual Machine Technology and Their Application In The Delivery Of ICT". In: *Accessed 14 March 2003 www.ddj.com/documents/s=882/ddjo008f/ o008f.htm*. 2002, S. 55–62.

[McK04] Kathryn S. McKinley, Hrsg. *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection*. ACM, 2004. ISBN: 1-58113-623-4.

[McN02] Brett McNamara. *Java(TM) Boutique - Image Shuffle*. `http://javaboutique.internet.com/ImageShuffle/`. 2002.

[Mes07] Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[MHo8a] Florian Mangold und Moritz Hammer. „[DE] VERFAHREN UND DATENVERARBEITUNGSSYSTEM ZUR RECHNERGESTÜTZTEN PERFORMANZANALYSE EINES DATENVERARBEITUNGSSYSTEMS [EN] METHOD AND DATA PROCESSING SYSTEM FOR COMPUTER-ASSISTED PERFORMANCE ANALYSIS OF A DATA PROCESSING SYSTEM [FR] PROCÉDÉ ET SYSTÈME DE TRAITEMENT DE DONNÉES POUR L'ANALYSE DE PERFORMANCE ASSISTÉE PAR ORDINATEUR D'UN SYSTÈME DE TRAITEMENT DE DONNÉES". European Patent 08736013. Apr. 2008.

[MHo8b] Florian Mangold und Moritz Hammer. „[DE] VERFAHREN UND DATENVERARBEITUNGSSYSTEM ZUR RECHNERGESTÜTZTEN PERFORMANZANALYSE EINES DATENVERARBEITUNGSSYSTEMS [EN] METHOD AND DATA PROCESSING SYSTEM FOR COMPUTER-ASSISTED PERFORMANCE ANALYSIS OF A DATA PROCESSING SYSTEM [FR] PROCÉDÉ ET SYSTÈME DE TRAITEMENT DE DONNÉES POUR L'ANALYSE DE PERFORMANCE ASSISTÉE PAR ORDINATEUR D'UN SYSTÈME DE TRAITEMENT DE DONNÉES". European Patent 2008054288. Apr. 2008.

[MHo8c] Florian Mangold und Moritz Hammer. „METHOD AND DATA PROCESSING SYSTEM FOR COMPUTER-ASSISTED PERFORMANCE ANALYSIS OF A DATA PROCESSING SYSTEM". International Patent WO/2008/128895. Okt. 2008.

[MHR09] Florian Mangold, Moritz Hammer und Harald Rölle. „Automatable & scalable late cycle performance analysis". In: *WOSP/SIPEW*. Hrsg. von Alan Adamson u. a. ACM, 2009. ISBN: 978-1-60558-563-5.

[Mig+00] Miguel de Miguel u. a. „UML extensions for the specification and evaluation of latency constraints in architectural models". In: *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*. New York, NY, USA: ACM, 2000, S. 83–88. ISBN: 1-58113-195-X.

[MKP07] Florian Mangold, Bernhard Kempter und Michael Pönitsch. „METHOD FOR THE COMPUTER-AIDED DETERMINATION OF AN OPTIMIZATION POTENIAL OF A SOFT-WARE SYSTEM". United States Patent 20100095275. Sep. 2007.

[MMAC09] Raymond H. Myers, Douglas C. Montgomery und C. M. Anderson-Cook: *Response Surface Methodology: Process and Product Optimization Using Designed Experiments*. Wiley-Interscience, 2009.

[MNV94] Yishay Mansour, Noam Nisan und Uzi Vishkin. „Trade-offs between communication throughput and parallel time“. In: *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1994, S. 372–381. ISBN: 0-89791-663-8.

[Mol06] Ethan Mollick. „Establishing Moore's Law“. In: *IEEE Annals of the History of Computing* 28.3 (2006), S. 62–75. ISSN: 1058-6180.

[Mon05] Douglas C. Montgomery. *Design and Analysis of Experiments, Student Solutions Manual*. Wiley, 2005. ISBN: 0471733040.

[MPK08] Florian Mangold, Michael Poenitsch und Bernhard Kempter. „[DE] VERFAHREN ZUM RECHNERGESTÜTZTEN BESTIMMEN DES OPTIMIERUNGSPOTENTIALS EINES SOFTWARESYSTEMS [EN] METHOD FOR THE COMPUTER-AIDED DETERMINATION OF AN OPTIMIZATION POTENTIAL OF A SOFTWARE SYSTEM [FR] PROCÉDÉ DE DÉTERMINATION ASSISTÉE PAR ORDINATEUR DU POTENTIEL D'OPTIMISATION D'UN SYSTÈME LOGICIEL“. European Patent 2008052638. März 2008.

[MR09a] Florian Mangold und Harald Roelle. „[DE] VERFAHREN UND VORRICHTUNG ZUM ANALYSIEREN EINER AUSFÜHRUNG EINES VORGEGEBENEN PROGRAMMABLAUFS AUF EINEM PHYSISCHEN RECHNERSYSTEM [EN] METHOD AND DEVICE FOR ANALYZING AN EXECUTION OF A PREDETERMINED PROGRAM FLOW ON A PHYSICAL COMPUTER SYSTEM [FR] PROCÉDÉ ET DISPOSITIF D'ANALYSE DE L'EXÉCUTION D'UN DÉROULEMENT DE PROGRAMME PRÉDÉFINI SUR UN SYSTÈME INFORMATIQUE PHYSIQUE“. European Patent 2009059340. Juli 2009.

[MR09b] Florian Mangold und Harald Roelle. „[DE] VERFAHREN UND VORRICHTUNG ZUM BESTIMMEN VON ANFORDERUNGSPARAMETERN AN MINDESTENS EINE PHYSISCHE HARDWAREEINHEIT [EN] METHOD AND DEVICE FOR DETERMINING REQUIREMENT PARAMETERS OF AT LEAST ONE PHYSICAL HARDWARE UNIT [FR] PROCÉDÉ ET DISPOSITIF POUR DÉTERMINER LES PARAMÈTRES DE CONTRAINTES D'AU MOINS UNE UNITÉ MATÉRIELLE PHYSIQUE“. European Patent 2009059342. Juli 2009.

[MR09c] Florian Mangold und Harald Roelle. „Framework zum Auffinden von Race Conditions und zur Validierung von Synchronisation mit Hilfe von Virtualisierung“. Schutzrecht Anmeldenummer: 10 2009 050 161.4. Okt. 2009.

[MR09d] Florian Mangold und Harald Roelle. „Framework zur Validierung von Systemen mit asynchroner Hardware mit Hilfe von Virtualisierung“. Schutzrecht Anmeldenummer: 10 2009 049 226.7. Okt. 2009.

[MR10a] Florian Mangold und Harald Roelle. „METHOD AND DEVICE FOR ANALYZING AN EXECUTION OF A PREDETERMINED PROGRAM FLOW ON A PHYSICAL COMPUTER SYSTEM“. International Patent WO/2010/025993. März 2010.

[MR10b] Florian Mangold und Harald Roelle. „METHOD AND DEVICE FOR DETERMINING REQUIREMENT PARAMETERS OF AT LEAST ONE PHYSICAL HARDWARE UNIT“. International Patent WO/2010/025994. März 2010.

[MS10] Gabriel A. Moreno und Connie U. Smith. „Performance analysis of real-time component architectures: An enhanced model interchange approach“. In: *Performance Evaluation* 67.8 (2010). Special Issue on Software and Performance, S. 612 –633. ISSN: 0166-5316.

[MS93a] Bodhisattwa Mukherjee und Karsten Schwan. „Experiments with Configurable Locks for Multiprocessors“. In: *ICPP*. 1993, S. 205–208.

[MS93b] Bodhisattwa Mukherjee und Karsten Schwan. „Improving Performance by Use of Adaptive Objects: Experimentation with a Configurable Multiprocessor Thread Package“. In: *HPDC*. 1993, S. 59–66.

[Mun+02] Cristina Hava Muntean u. a. *An Adaptive Web Server Application*. 2002.

[Mur85] Fionn Murtagh. „Multidimensional Clustering Algorithms“. In: *COMPSTAT*. Lectures 4. Wuerzburg: Physica-Verlag, 1985.

[Mus+08] Madanlal Musuvathi u. a. „Finding and Reproducing Heisenbugs in Concurrent Programs.“ In: *OSDI*. Hrsg. von Richard Draves und Robbert van Renesse. USENIX Association, 2008, S. 267–280. ISBN: 978-1-931971-65-2.

[MV84] Kurt Mehlhorn und Uzi Vishkin. „Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories“. In: *Acta Inf.* 21.4 (1984), S. 339–374. ISSN: 0001-5903.

[MWW10] Bernd Mohr, Brian J. N. Wylie und Felix Wolf. „Performance measurement and analysis tools for extremely scalable systems“. In: *Concurrency and Computation: Practice and Experience* 22 (2010).

[Neu06] Peter G. Neumann. „System and network trustworthiness in perspective“. In: *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*. Alexandria, Virginia, USA: ACM, 2006, S. 1–5. ISBN: 1-59593-518-5.

[Neu93] John von Neumann. „First Draft of a Report on the EDVAC“. In: *IEEE Ann. Hist. Comput.* 15.4 (1993), S. 27–75. ISSN: 1058-6180.

[Nic] Steve Nickolas. *FreeDOS ODIN*. http://odin.fdos.org/. zuletzt abgerufen am 04.08.2010.

[NM92] Robert H. B. Netzer und Barton P. Miller. „What are race conditions?: Some issues and formalizations“. In: *ACM Lett. Program. Lang. Syst.* 1.1 (1992), S. 74–88. ISSN: 1057-4514.

[Off92] United States General Accounting Office. *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*. Techn. Ber. GAO/IMTEC-92-26. Report GAO/IMTEC-92-26, Information ManagementTechnology Division, Washington, D.C., February 1992, 1992.

[Ove+05] Jeffrey Overbey u. a. „Refactorings for Fortran and high-performance computing“. In: *SE-HPCS '05: Proceedings of the second international workshop on Software engineering for high performance computing system applications*. New York, NY, USA: ACM, 2005, S. 37–39. ISBN: 1-59593-117-1.

[PDL06] Martin Pohlack, Björn Döbel und Adam Lackorzynski. „Towards runtime monitoring in real-time systems“. In: *In Proceedings of the Eighth Real-Time Linux Workshop*. 2006.

[Pei+04] Marcus Peinado u. a. „NGSCB: A Trusted Open System“. In: *ACISP*. Hrsg. von Huaxiong Wang, Josef Pieprzyk und Vijay Varadharajan. Bd. 3108. Lecture Notes in Computer Science. Springer, 2004, S. 86–97. ISBN: 3-540-22379-7.

[Per06] William Perry. *Effective methods for software testing, third edition*. New York, NY, USA: John Wiley & Sons, Inc., 2006. ISBN: 9780764598371.

[Per08] Dewayne E. Perry. „"Large" abstractions for software engineering“. In: *ROA '08: Proceedings of the 2nd international workshop on The role of abstraction in software engineering*. Leipzig, Germany: ACM, 2008, S. 31–33. ISBN: 978-1-60558-028-7.

[Pet82] Susan Graham Peter. *gprof: a Call Graph Execution Profiler*. 1982.

[PG74] Gerald J. Popek und Robert P. Goldberg. „Formal requirements for virtualizable third generation architectures“. In: *Commun. ACM* 17.7 (1974), S. 412–421. ISSN: 0001-0782.

[Pön06] Michael Pönitsch. „Verfahren zum rechnergestützten Optimieren des Ressourcenverbrauchs eines Programms“. DE. Pat. DE102006046201A1. 2006.

[Pön09] Michael Pönitsch. „[DE] VERFAHREN ZUM RECHNERGESTÜTZTEN OPTIMIEREN DES RESSOURCENVERBRAUCHS EINES PROGRAMMS [EN] METHOD FOR THE COMPUTER-ASSISTED OPTIMIZATION OF THE RESOURCE UTILIZATION OF A PROGRAM [FR] PROCÉDÉ D'OPTIMISATION ASSISTÉE PAR ORDINATEUR DE LA CONSOMMATION DE RESSOURCE D'UN PROGRAMME". Pat. 07820575. 2009.

[Pol88] Wolfgang Polasek. *Explorative Daten-Analyse. EDA ; Einführung in die deskriptive Statistik*. Heidelberger Lehrtexte. Berlin [u.a.]: Springer, 1988. V, 232. ISBN: 3540194177.

[Poo92] Rob Pooley. *Book review: Performance Engineering of Software Systems by Connie U. Smith (Addison Wesley 1990)*. Bd. 20. 2. New York, NY, USA: ACM, 1992, S. 23–24.

[PPM10] Diego Perez-Palacin und José Merseguer. „Performance Evaluation of Self-reconfigurable Service-oriented Software With Stochastic Petri Nets". In: *Electronic Notes in Theoretical Computer Science* 261 (2010). Proceedings of the Fourth International Workshop on the Practical Application of Stochastic Modelling (PASM 2009), S. 181 –201. ISSN: 1571-0661.

[PS02] Dorina C. Petriu und Hui Shen. „Applying the UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Specifications". In: *Computer Performance Evaluation / TOOLS*. Hrsg. von Tony Field u. a. Bd. 2324. Lecture Notes in Computer Science. Springer, 2002, S. 159–177. ISBN: 3-540-43539-5.

[PVH10] Sangmin Park, Richard W. Vuduc und Mary Jean Harrold. „Falcon: fault localization in concurrent programs". In: *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. Cape Town, South Africa: ACM, 2010, S. 245–254. ISBN: 978-1-60558-719-6.

[PW00] Dorina C. Petriu und Xin Wang. „From UML Descriptions of High-Level Software Architectures to LQN Performance Models". In: *AGTIVE '99: Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance*. London, UK: Springer-Verlag, 2000, S. 47–62. ISBN: 3-540-67658-9.

[PW02] Dorin C. Petriu und C. Murray Woodside. „Software Performance Models from System Scenarios in Use Case Maps". In: *TOOLS '02: Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools*. London, UK: Springer-Verlag, 2002, S. 141–158. ISBN: 3-540-43539-5.

[PW05] Dorin Bogdan Petriu und Murray Woodside. „Software performance models from system scenarios“. In: *Perform. Eval.* 61.1 (2005), S. 65–89. ISSN: 0166-5316.

[PZ97] Marcin Paprzycki und Janusz Zalewski. „Parallel computing in Ada: an overview and critique“. In: *Ada Lett.* XVII.2 (1997), S. 55–62. ISSN: 1094-3641.

[R D] R Documentation. „R: Hierarchical Clustering“. http://sekhon.berkeley.edu/stats/html/hclust.html. zuletzt abgerufen am 18.09.2010.

[Åo0] Ola Ågren. „Virtual machines as an aid in teaching computer concepts“. In: *WCAE '00: Proceedings of the 2000 workshop on Computer architecture education*. New York, NY, USA: ACM, 2000, S. 14.

[Raj00] Sergio Rajsbaum. „Principles of distributed computing: an exciting challenge“. In: *SIGACT News* 31.4 (2000), S. 52–61. ISSN: 0163-5700.

[RBJ88] Abhiram G. Ranade, Sandeep N. Bhatt und Lennart S. Johnsson. „The fluent abstract machine“. In: *Proceedings of the fifth MIT conference on Advanced research in VLSI*. Cambridge, MA, USA: MIT Press, 1988, S. 71–93. ISBN: 0-262-01100-X.

[RBS93] H. Dieter Rombach, Victor R. Basili und Richard W. Selby, Hrsg. *Experimental Software Engineering Issues: Critical Assessment and Future Directions, International Workshop Dagstuhl Castle, Germany, September 14-18, 1992, Proceedings*. Bd. 706. Lecture Notes in Computer Science. Springer, 1993. ISBN: 3-540-57092-6.

[Rei99] Karl Rüdiger Reischuk. *Einführung in die Komplexitätstheorie. Band 1: Grundlagen Maschinenmodelle, Zeit- und Platzkomplexität, Nichtdeterminismus*. Teubner Verlag, 1999.

[Reu+07] Ralf Reussner u. a. *The Palladio component model*. Karlsruhe, 2007.

[Roj97] Raúl Rojas. „Konrad Zuse's Legacy: The Architecture of the Z1 and Z3“. In: *IEEE Annals of the History of Computing* 19.2 (1997), S. 5–16.

[Rot84] Robert Rothstein. „Standard hardware-software interface for connecting any instrument which provides a digital output stream with any digital host computer“. Pat. 4485439. 1984.

[RS] Harigovind V. Ramasamy und Matthias Schunter. *Architecting Dependable Systems Using Virtualization*.

[RS00] Omer Rana und Matthew Shields. „Performance Analysis of Java Using Petri Nets“. In: *High Performance Computing and Networking*. Hrsg. von Marian Bubak u. a. Bd. 1823. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2000, S. 657–667.

[RS99] Claus Rautenstrauch und André Scholz. „Vom Performance Tuning zum Software Performance Engineering am Beispiel datenbankbasierter Anwendungssysteme: Reduktion des performancebedingten Entwicklungsrisikos“. In: *Informatik Spektrum* 22.4 (1999), S. 261–275.

[Ryd79] Barbara G. Ryder. „Constructing the Call Graph of a Program.“ In: *IEEE Trans. Software Eng.* 5.3 (1979), S. 216–226.

[Sac04] Lothar Sachs. *Angewandte Statistik. Anwendung statistischer Methoden ; mit 317 Tabellen und 99 Übersichten*. 11., überarb. und aktualisierte Aufl. Berlin [u.a.]: Springer, 2004. XXXVII, 889. ISBN: 3540405550.

[San+07] Jayshankar Sankarasetty u. a. „Software performance in the real world: personal lessons from the performance trauma team“. In: *WOSP '07: Proceedings of the 6th international workshop on Software and performance*. New York, NY, USA: ACM, 2007, S. 201–208. ISBN: 1-59593-297-6.

[SAN08] Robert Shingledecker, John Andrews und Christopher Negus. *The Official Damn Small Linux Book: The Tiny Adaptable Linux That Runs on Anything*. Negus Live Linux Series. Upper Saddle River, NJ: Prentice-Hall, 2008.

[San+98] Jörg Sander u. a. „Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and Its Applications“. In: *Data Mining and Knowledge Discovery* 2.2 (Juni 1998), S. 169–194.

[Sav01] Alberto Savoia. „Web Load Test Planning: Predicting how your Web site will respond to stress“. In: *STQE* 3 (2001), S. 11–22.

[Sav82] Walter J. Savitch. „Parallel Random Access Machines with Powerful Instruction Sets“. In: *Mathematical Systems Theory* 15.3 (1982), S. 191–210.

[SB01] Ken Schwaber und Mike Beedle. *Agile Software Development with Scrum*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001. ISBN: 0130676349.

[SB10] Girish Suryanarayana und Shantanu Bhattacharya. „Experience report: a knowledge repository-centric approach to performance tuning“. In: *ISEC '10: Proceedings of the 3rd India software engineering conference*. New York, NY, USA: ACM, 2010, S. 131–136. ISBN: 978-1-60558-922-0.

[SB84] C. Slot und P. van Emde Boas. „On tape versus core an application of space efficient perfect hash functions to the invariance of space“. In: *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1984, S. 391–400. ISBN: 0-89791-133-4.

[SBL08] Marwa Shousha, Lionel Briand und Yvan Labiche. „A UML/SPT Model Analysis Methodology for Concurrent Systems Based on Genetic Algorithms“. In: *Model Driven Engineering Languages and Systems*. Hrsg. von Krzysztof Czarnecki u. a. Bd. 5301. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, S. 475–489.

[Sch01] U. Schöning. *Theoretische Informatik - kurz gefasst*. 4. Aufl. Spektrum Akademischer Verlag, 2001.

[Sch95] Dieter Scheerer. „Der Prozessor der SB-PRAM“. Diss. Universität des Saarlandes, 1995.

[Sch96] Jürgen Schürmann. *Pattern classification: a unified view of statistical and neural approaches*. New York, NY, USA: John Wiley & Sons, Inc., 1996. ISBN: 0-471-13534-8.

[Sch97] Robert R. Schaller. „Moore's law: past, present, and future“. In: *IEEE Spectr.* 34.6 (1997), S. 52–59. ISSN: 0018-9235.

[SDJ07] Dag I. K. Sjoberg, Tore Dyba und Magne Jorgensen. „The Future of Empirical Methods in Software Engineering Research“. In: *FOSE '07: 2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, S. 358–378. ISBN: 0-7695-2829-5.

[SE94a] Amitabh Srivastava und Alan Eustace. „ATOM: a system for building customized program analysis tools“. In: *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1994, S. 196–205. ISBN: 0-89791-662-X.

[SE94b] Amitabh Srivastava und Alan Eustace. „ATOM: a system for building customized program analysis tools“. In: *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1994, S. 196–205. ISBN: 0-89791-662-X.

[SG98] Jochen Seemann und Jürgen Wolff von Gudenberg. „UML- Unified Modeling Language“. In: *Informatik-Spektrum* 21 (2 1998). 10.1007/s002870050092, S. 89–90. ISSN: 0170-6012.

[Sha00] J. Shaw. „Web Application Performance Testing — a Case Study of an Online Learning Application“. In: *BT Technology Journal* 18.2 (2000), S. 79–86. ISSN: 1358-3948.

[Sha+10] Asaf Shabtai u. a. „Google Android: A Comprehensive Security Assessment“. In: *IEEE Security and Privacy* 8.2 (2010), S. 35–44. ISSN: 1540-7993.

[She03] Nitin M. Shetti. *Heisenbugs and Bohrbugs: Why are they different?* Techn. Ber. Rutgers, The State University of New Jersey, 2003.

[She93] Thomas J. Sheffler. „Implementing the multiprefix operation on parallel and vector computers“. In: *SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 1993, S. 377–386. ISBN: 0-89791-599-2.

[Shi02] Jack Shirazi. *Java Performance Tuning*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002. ISBN: 0596003773.

[Sim01] Harald C. Simmler. „Preemptive Multitasking auf FPGA-Prozessoren: ein Betriebssystem für FPGA-Prozessoren“. Diss. Universität Mannheim, 2001.

[Sin95] Raghu Singh. „The Software Life Cycle Process Standard“. In: *IEEE Computer* 28.11 (1995), S. 89–90.

[SLG88] Roberto Salama, Wentai Liu und Ronald S. Gyurcsik. „Software experience with concurrent C and LISP in a distributed system“. In: *CSC '88: Proceedings of the 1988 ACM sixteenth annual conference on Computer science*. New York, NY, USA: ACM, 1988, S. 329–334. ISBN: 0-89791-260-8.

[SLP09] Connie Smith, Catalina Llado und Ramon Puigjaner. „Automatic Generation of Performance Analysis Results: Requirements and Demonstration“. In: *Computer Performance Engineering*. Hrsg. von Jeremy Bradley. Bd. 5652. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, S. 73–78.

[SLP10a] Connie U. Smith, Catalina M. Llado und Ramon Puigjaner. „Performance Model Interchange Format (PMIF 2): A comprehensive approach to Queueing Network Model interoperability“. In: *Performance Evaluation* 67.7 (2010), S. 548 –568. ISSN: 0166-5316.

[SLP10b] Connie U. Smith, Catalina M. Llado und Ramon Puigjaner. „PMIF extensions: increasing the scope of supported models“. In: *WOSP/SIPEW '10: Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*. San Jose, California, USA: ACM, 2010, S. 255–256. ISBN: 978-1-60558-563-5.

[SM10] Ajanta De Sarkar und Nandini Mukherjee. „A Study on Performance Analysis Tools for Applications Running on Large Distributed Systems“. In: *CoRR* abs/1006.2650 (2010).

[Smi90] Connie U. Smith. *Performance Engineering of Software Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990. ISBN: 0201537699.

[SN05] Jim Smith und Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 1558609105.

[Sof07] Quest Software. *Datasheet - JProbe Suite*. `www.quest.com/Quest_Site_Assets/PDF/Datasheet_-_JProbe_Suite_3.pdf`. 2007.

[Sok06] Dehla Sokenou. „UML-basierter Klassen- und Integrationstest objektorientierter Programme“. Diss. TU Berlin, Fakultät für Elektrotechnik und Informatik, März 2006.

[Som95] Ian Sommerville. *Software engineering (5th ed.)* Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-42765-6.

[Sou08] Gabriel Southern. *Symmetric Multiprocessing Virtualization*. Diplomarbeit. 2008.

[Spe04] Charles Spearman. „General intelligence objectively determined and measured“. In: *American Journal of Psychology 15*. University of Illinois Press, 1904, S. 201–293.

[Spe27] Charles Spearman. *The Abilities of Man: Their Nature and Measurement*. (Macmillan & Co, 1927.

[Spi10] Spiegel Online. *Neuer Rekord - Franzose berechnet Pi auf 2,7 Billionen Ziffern*. http://www.spiegel.de/wissenschaft/natur/0,1518,664489,00.html. zuletzt abgerufen am 27.01.2010. Jan. 2010.

[SRA97] SRAH. *Software Reengineering Assessment Handbook*. Techn. Ber. JLC-HDBK-SRAH Version 3.0. STSC, U.S. Air Force, 1997.

[SRR02] Nayda G. Santiago, Diane T. Rover und Domingo Rodríguez. „A Statistical Approach for the Analysis of the Relation Between Low-Level Performance Information, the Code, and the Environment“. In: *ICPP Workshops*. IEEE Computer Society, 2002, S. 282–289. ISBN: 0-7695-1680-7.

[SS63] J. C. Shepherdson und H. E. Sturgis. „Computability of Recursive Functions“. In: *J. ACM* 10.2 (1963), S. 217–255. ISSN: 0004-5411.

[Sta+07] William Stallings u. a. *Computer Organization and Architecture: WITH Discrete Mathematics for Computer Scientists AND Digital Design Designing for Performance*. USA: Addison-Wesley Publishing Company, 2007. ISBN: 1405892706, 9781405892704.

[Sta84] Thomas A. Standish. „An Essay on Software Reuse“. In: *IEEE Transactions on Software Engineering* 10.5 (Sep. 1984). Special Issue on Software Reusability, S. 494–497.

[Sto02] Scott D. Stoller. „Testing Concurrent Java Programs using Randomized Scheduling“. In: *Proc. Second Workshop on Runtime Verification (RV)*. Bd. 70(4). Electronic Notes in Theoretical Computer Science. Elsevier, Juli 2002.

[Stö05] Harald Störrle. *UML 2 für Studenten*. München: Pearson Studium, 2005.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0201700735.

[Str59] Christopher Strachey. „Time sharing in large, fast computers“. In: *IFIP Congress*. 1959, S. 336–341.

[Sut05] Jeff Sutherland. „Future of Scrum: Parallel Pipelining of Sprints in Complex Projects“. In: *ADC '05: Proceedings of the Agile Development Conference*. Washington, DC, USA: IEEE Computer Society, 2005, S. 90–102. ISBN: 0-7695-2487-7.

[SW02] C. Smith und L. Williams. *Performance solutions: a practical guide to creating responsive, scalable software*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2002. ISBN: 0-201-72229-1.

[Taf+02] S. Tucker Taft u. a., Hrsg. *Consolidated Ada Reference Manual. Language and Standard Libraries, International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1*. Bd. 2219. Lecture Notes in Computer Science. New York, NY, USA: Springer-Verlag New York, Inc., 2002. ISBN: 3-540-43038-5.

[Tan09] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. 3., aktualisierte Auflage. Pearson Studium, 2009. ISBN: 3827373425.

[TDG09] Mirco Tribastone, Adam Duguid und Stephen Gilmore. „The PEPA eclipse plugin“. In: *SIGMETRICS Perform. Eval. Rev.* 36.4 (2009), S. 28–33. ISSN: 0163-5999.

[Teu04] Christof Teuscher. *Alan Turing: Life and Legacy of a Great Thinker*. Springer, Berlin, 2004.

[TG98] Andrew S. Tanenbaum und James R. Goodman. *Structured Computer Organization*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998. ISBN: 0130959901.

[The05] The Object Management Group. *UML Profile for Schedulability, Performance and Time (SPT)*. 2005.

[The07] The Eclipse Foundation. *The AspectJ Project*. `http://www.eclipse.org/aspectj/`. 2007.

[The07] Eno Thereska. „Enabling what-if explorations in systems“. Adviser-Ganger, Gregory R. Diss. Pittsburgh, PA, USA: Carnegie Mellon University, 2007.

[TP10] Rasha Tawhid und Dorina Petriu. „Integrating Performance Analysis in the Model Driven Development of Software Product Lines“. In: *Model Driven Engineering Languages and Systems*. Hrsg. von Krzysztof Czarnecki u. a. Bd. 5301. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, S. 490–504.

[TPB10] Daniel Thomas, Jean-Pierre Panziera und John Baron. „MPInside: a performance analysis and diagnostic tool for MPI applications“. In: *WOSP/SIPEW '10: Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*. New York, NY, USA: ACM, 2010, S. 79–86. ISBN: 978-1-60558-563-5.

[Tri07] Mirco Tribastone. „The PEPA Plug-in Project“. In: *QEST*. IEEE Computer Society, 2007, S. 53–54. ISBN: 0-7695-2883-X.

[Tri10] Mirco Tribastone. „Scalable Analysis of Stochastic Process Algebra Models“. Diss. School of Informatics, The University of Edinburgh, 2010.

[TT04] The IEEE und The Open Group. *The Open Group Base Specifications Issue 6*. 2004. Kap. 4.14 Seconds Since the Epoch.

[Tur36] Alan M. Turing. „On Computable Numbers, with an application to the Entscheidungsproblem“. In: *Proc. London Math. Soc.* 2.42 (1936), S. 230–265.

[UM08] Dieter Urban und Jochen Mayerl. *Regressionsanalyse: Theorie, Technik und Anwendung*. Wiesbaden: VS Verlag für Sozialwissenschaften / GWV Fachverlage GmbH, Wiesbaden, 2008. ISBN: 9783531911946.

[Var97] Melinda Varian. *VM and the VM Community: Past, Present*. SHARE 89 Sessions. Aug. 1997.

[Ven+09] M. Venables u. a. „News [briefing]“. In: *Engineering & Technology* 4 (2009), S. 3 –14.

[Vig07] Giovanni Vigna. „Malware Detection“. In: Hrsg. von M. Christodorescu u. a. Advances in Information Security. Springer, 2007. Kap. Static Disassembly and Code Analysis.

[Vis83] Uzi Vishkin. „Implementation of Simultaneous Memory Address Access in Models That Forbid It“. In: *J. Algorithms* 4.1 (1983), S. 45–50.

[VT01] K Vaidyanathan und K S Trivedi. „Extended classification of software faults based on aging“. In: *In: Proceedings of the Twelfth International Symposium on Software Reliability Engineering (ISSRE), IEEE*. 2001, S. 27–28.

[Wei+02] Gerhard Weikum u. a. „Self-tuning database technology and information services: from wishful thinking to viable engineering“. In: *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 2002, S. 20–31.

[Wel83] W. Porter Welbourne. „Transportability of software applications on microcomputers (a discussion of the p-system concepts)“. In: *Computer Applications in Medical Care, 1983. Proceedings*. 1983, S. 974–977.

[Wer95] Klaus-Dieter Wernecke. *Angewandte Statistik für die Praxis*. Addison-Wesley, 1995.

[WFP07] Murray Woodside, Greg Franks und Dorina C. Petriu. „The Future of Software Performance Engineering“. In: *FOSE '07: 2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, S. 171–187. ISBN: 0-7695-2829-5.

[Wil97] Graham J. Wills. „NicheWorks - interactive visualization of very large graphs“. In: *Proceedings of Graph Drawing '97*. Springer-Verlag, 1997, S. 403–414.

[Wol02] Felix Wolf. „Automatic Performance Analysis on Parallel Computers with SMP Nodes“. Diss. Forschungszentrum Jülich., 2002.

[Woo+01] Murray Woodside u. a. „Automated performance modeling of software generated by a design environment“. In: *Performance Evaluation* 45.2-3 (Juli 2001), S. 107–123.

[Woo+09] Murray Woodside u. a. „Performance analysis of security aspects by weaving scenarios extracted from UML models“. In: *Journal of Systems and Software* 82.1 (2009). Special Issue: Software Performance - Modeling and Analysis, S. 56 –74. ISSN: 0164-1212.

[WR09] Robert Warnke und Thomas Ritzau. *QEMU & KVM: Wiki http://qemu-buch.de*. Books on Demand, 2009.

[WS00] Jingwei Wu und Margaret-Anne D. Storey. „A multi-perspective software visualization environment“. In: *CASCON '00: Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*. Mississauga, Ontario, Canada: IBM Press, 2000, S. 15.

[WS03] Lloyd G. Williams und Connie U. Smith. „Making the Business Case for Software Performance Engineering“. In: *Int. CMG Conference*. Computer Measurement Group, 2003, S. 349–358.

[Wu+08] Xingfu Wu u. a. „Performance Analysis and Optimization of Parallel Scientific Applications on CMP Cluster Systems“. In: *ICPPW '08: Proceedings of the 2008 International Conference on Parallel Processing - Workshops*. Washington, DC, USA: IEEE Computer Society, 2008, S. 188–195. ISBN: 978-0-7695-3375-9.

[WWMi93] Alf Wachsmann, Friedrich Wichmann und Fachbereich Mathematikinformatik. *OCCAM-light — A multiparadigm programming language for Transputer networks.* Forschungsbericht der Forschergruppe Effiziente Nutzung massiv paralleler Systeme. 1993.

[Wyl10] Brian J. N. Wylie. „Improved Scalasca toolset support for performance analysis of Cray XT systems“. In: *HPC-Europa2: Science and Supercomputing in Europe research highlights 2009*. Hrsg. von Silvia Monfardini. Casalecchio di Reno (Bologna), Italy: CINECA Consorzio Interuniversitario, 2010, S. 67. ISBN: 978-88-86037-23-5.

[XHG08] Chenchen Xi, Bruno Harbulot und John R. Gurd. „A synchronized block join point for AspectJ“. In: *FOAL '08: Proceedings of the 7th workshop on Foundations of aspect-oriented languages*. New York, NY, USA: ACM, 2008, S. 39–39. ISBN: 978-1-60558-110-1.

[XKJ10] Chang Xu, Steven R. Kirk und Samantha Jenkins. „Tiling for Performance Tuning on Different Models of GPUs“. In: *CoRR* abs/1001.1718 (2010), S. 500 –504.

[Yag03] Karim Yaghmour. *Building Embedded Linux Systems*. O'Reilly Media, Inc., 2003. ISBN: 059600222X.

[YC79] Edward Yourdon und Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Englewood Cliffs, NJ: Prentice Hall, 1979.

[YP05] Michal Young und Mauro Pezze. *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, 2005. ISBN: 0471455938.

[YRC05] Yuan Yu, Tom Rodeheffer und Wei Chen. „RaceTrack: efficient detection of data race conditions via adaptive tracking“. In: *SIGOPS Oper. Syst. Rev.* 39.5 (2005), S. 221–234. ISSN: 0163-5980.

[ZFL10] Kunhua Zhu, Junhui Fu und Yancui Li. „Research the performance testing and performance improvement strategy in web application“. In: *Education Technology and Computer (ICETC), 2010 2nd International Conference on*. 2010.

[Zim05] Dennis Zimmer. *VMware und Microsoft Virtual Server: Virtuelle Server im professionellen Einsatz*. Bonn: Galileo Press, 2005. ISBN: 978-3-89842-701-2.

[Zim80] H. Zimmermann. „OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection“. In: *IEEE Transactions on Communications* 28.4 (1980), S. 425–432.

[Zwe+95] Stuart H. Zweben u. a. „The Effects of Layering and Encapsulation on Software Development Cost and Quality“. In: *IEEE Transactions on Software Engineering* 21 (1995), S. 200–208. ISSN: 0098-5589.

Index

Florian Mangold

Kommunikationsdaten

Name	Florian Mangold
Geburtsdatum	13. April 1978
Geburtsort	Landsberg am Lech
Familienstand	verlobt
Privatadresse	Tassiloweg 28 85737 Ismaning

Studium und Ausbildung

11/2007 – 11/2010	**Promotion**, *Lehrstuhl für Programmierung und Softwaretechnik*, Ludwig-Maximilians-Universität München, Doktorvater: Prof. Dr. Martin Wirsing in Kooperation mit der Siemens AG.
10/1999 – 09/2007	**Diplom Informatik**, *Ludwig-Maximilians-Universität*, München, Prädikat: sehr gut. Schwerpunkte: Informationssysteme, theoretische Informatik, Programmierung und Softwaretechnik Nebenfach Pädagogik, Schwerpunkt Erwachsenenbildung
11/2002	**Vordiplom**, *Ludwig-Maximilians-Universität*, München, Prädikat: sehr gut.
09/1998 – 07/1999	**Fachgebundene Hochschulreife**, *Marian-Bathko-Berufsoberschule*, München.
09/1997 – 07/1998	**Fachhochschulreife**, *Marian-Bathko-Berufsoberschule*, München.
10/1994 – 10/1996	**Staatlich geprüfter Technischer Assistent für Informatik**, *Siemens Technik Akademie*, München.

Berufserfahrung

11/2007 - 11/2010	**Doktorand**, *Siemens AG*, Corporate Technology, München.
10/2005 - 06/2006	**Werkstudent**, *Siemens AG*, Corporate Technology, München.
02/2005 - 06/2005	**Werkstudent**, *Siemens AG*, Siemens Mobile, München.
04/2002 - 09/2004	**Mitarbeiter bei der Erstellung einer eLearning Vorlesung (Medienrezeption und Mediensozialisation) für die virtuelle Hochschule Bayern**, *Ludwig-Maximilians-Universität*, München.
03/2001 - 08/2003	**Werkstudent**, *Siemens AG*, Automatisierungstechnik, München.
03/2000 - 02/2001	**Zivildienst**, *Wildbiologischen Gesellschaft München*, .
11/1997 - 11/1999	**Werkschüler**, *Siemens AG*, Postautomatisierung, München.
11/1996 - 11/1997	**Ingenieur-Assistent**, *Siemens AG*, Automatisierungstechnik, München.

Publikationen

Florian Mangold. *Performanzanalyse mit Hilfe künstlicher Varianz*. Diplomarbeit. Ludwig-Maximilians-Universität München, Geschwister-Scholl-Platz 1, 80539 München, Deutschland, 2007.

Moritz Hammer, Bernhard Kempter, Florian Mangold und Harald Rölle. „Skalierbare Performanzanalyse durch Prolongation“. In: *Software Engineering*. Hrsg. von Korbinian Herrmann und Bernd Brügge. Bd. 121. LNI. GI, 2008. ISBN: 978-3-88579-215-4.

Florian Mangold, Moritz Hammer und Harald Rölle. „Automatable & scalable late cycle performance analysis“. In: *WOSP/SIPEW*. Hrsg. von Alan Adamson, Andre B. Bondi, Carlos Juiz und Mark S. Squillante. ACM, 2009. ISBN: 978-1-60558-563-5.

Schutzrechte: Dynamische Kongruenzanalyse von Laufzeiteigenschaften eines Softwaresystems

Florian Mangold und Moritz Hammer. „[DE] VERFAHREN UND DATENVERARBEITUNGSSYSTEM ZUR RECHNERGESTÜTZTEN PERFORMANZANALYSE EINES DATENVERARBEITUNGSSYSTEMS [EN] METHOD AND DATA PROCESSING SYSTEM FOR COMPUTER-ASSISTED PERFORMANCE ANALYSIS OF A DATA PROCESSING SYSTEM [FR] PROCÉDÉ ET SYSTÈME DE TRAITEMENT DE DONNÉES POUR L'ANALYSE DE PERFORMANCE ASSISTÉE PAR ORDINATEUR D'UN SYSTÈME DE TRAITEMENT DE DONNÉES". European Patent 08736013. Apr. 2008.

Florian Mangold und Moritz Hammer. „[DE] VERFAHREN UND DATENVERARBEITUNGSSYSTEM ZUR RECHNERGESTÜTZTEN PERFORMANZANALYSE EINES DATENVERARBEITUNGSSYSTEMS [EN] METHOD AND DATA PROCESSING SYSTEM FOR COMPUTER-ASSISTED PERFORMANCE ANALYSIS OF A DATA PROCESSING SYSTEM [FR] PROCÉDÉ ET SYSTÈME DE TRAITEMENT DE DONNÉES POUR L'ANALYSE DE PERFORMANCE ASSISTÉE PAR ORDINATEUR D'UN SYSTÈME DE TRAITEMENT DE DONNÉES". European Patent 2008054288. Apr. 2008.

Florian Mangold und Moritz Hammer. „METHOD AND DATA PROCESSING SYSTEM FOR COMPUTER-ASSISTED PERFORMANCE ANALYSIS OF A DATA PROCESSING SYSTEM". International Patent WO/2008/128895. Okt. 2008.

Schutzrechte: Black Box Ressource Dependence Graph

Mortiz Hammer, Kurt Majewski, Florian Mangold, Christioph Moll, Harald Roelle und Rainer Wasgint. „[DE] Verfahren zum rechnergestützten Ermitteln der Abhängigkeiten einer Vielzahl von Modulen eines technischen Systems, insbesondere eines Softwaresystems". Schutzrecht 102007029133. Juni 2007.

Florian Mangold, Moritz Hammer, Kurt Majewski, Christoph Moll, Harald Rölle und Rainer Wasgint. „METHOD FOR THE COMPUTER-AIDED DETERMINATION OF THE DEPENDENCIES OF A PLURALITY OF MODULES OF A TECHNICAL SYSTEM, ESPECIALLY OF A SOFTWARE SYSTEM". International Patent WO/2008/113682. Sep. 2008.

Florian Mangold, Mortiz Hammer, Kurt Majewski, Christioph Moll, Harald Roelle und Rainer Wasgint. „[DE] VERFAHREN ZUM RECHNERGESTÜTZTEN ERMITTELN DER ABHÄNGIGKEITEN EINER VIELZAHL VON MODULEN EINES TECHNISCHEN SYSTEMS, INSBESONDERE EINES SOFTWARESYSTEMS [EN] METHOD FOR THE COMPUTER-AIDED DETERMINATION OF THE DEPENDENCIES OF A PLURALITY OF MODULES OF A TECHNICAL SYSTEM, ESPECIALLY OF A SOFTWARE SYSTEM [FR] PROCÉDÉ DE DÉTERMINATION ASSISTÉE PAR ORDINATEUR DES DÉPENDANCES D'UNE PLURALITÉ DE MODULES D'UN SYSTÈME TECHNIQUE, NOTAMMENT D'UN SYSTÈME LOGICIEL". European Patent 08717395. März 2008.

Florian Mangold, Mortiz Hammer, Kurt Majewski, Christioph Moll, Harald Roelle und Rainer Wasgint. „[DE] VERFAHREN ZUM RECHNERGESTÜTZTEN ERMITTELN DER ABHÄNGIGKEITEN EINER VIELZAHL VON MODULEN EINES TECHNISCHEN SYSTEMS, INSBESONDERE EINES SOFTWARESYSTEMS [EN] METHOD FOR THE COMPUTER-AIDED DETERMINATION OF THE DEPENDENCIES OF A PLURALITY OF MODULES OF A TECHNICAL SYSTEM, ESPECIALLY OF A SOFTWARE SYSTEM [FR] PROCÉDÉ DE DÉTERMINATION ASSISTÉE PAR ORDINATEUR DES DÉPENDANCES

D'UNE PLURALITÉ DE MODULES D'UN SYSTÈME TECHNIQUE, NOTAMMENT D'UN SYSTÈME LOGICIEL". European Patent 2008052641. März 2008.

Florian Mangold, Christoph Moll, Harald Rölle, Kurt Majewski, Moritz Hammer und Rainer Wasgint. „METHOD FOR THE COMPUTER-AIDED DETERMINATION OF THE DEPENDENCIES OF A PLURARITY OF MODULES OF A TECHNICAL SYSTEM, ESPECIALLY OF A SOFTWARE SYSTEM". United States Patent 20100088663. Aug. 2008.

Schutzrechte: Simuliertes Optimieren von Systemteilen

Bernhard Kempter, Florian Mangold und Michael Pönitsch. „METHOD FOR THE COMPUTER-AIDED DETERMINATION OF AN OPTIMIZATION POTENIAL OF A SOFT-WARE SYSTEM". United States Patent 20100095275. Sep. 2007.

Bernhard Kempter, Florian Mangold und Michael Pönitsch. „METHOD FOR THE COMPUTER-AIDED DETERMINATION OF AN OPTIMIZATION POTENTIAL OF A SOFTWARE SYSTEM". International Patent WO/2008/113681. Sep. 2007.

Florian Mangold, Michael Poenitsch und Bernhard Kempter. „[DE] VERFAHREN ZUM RECHNERGESTÜTZTEN BESTIMMEN DES OPTIMIERUNGSPOTENTIALS EINES SOFTWARESYSTEMS [EN] METHOD FOR THE COMPUTER-AIDED DETERMINATION OF AN OPTIMIZATION POTENTIAL OF A SOFTWARE SYSTEM [FR] PROCÉDÉ DE DÉTERMINATION ASSISTÉE PAR ORDINATEUR DU POTENTIEL D'OPTIMISATION D'UN SYSTÈME LOGICIEL". European Patent 2008052638. März 2008.

Schutzrechte: Analyse von Hardware-rahmenbedingungen

Florian Mangold und Harald Roelle. „[DE] VERFAHREN UND VORRICHTUNG ZUM BESTIMMEN VON ANFORDERUNGSPARAMETERN AN MINDESTENS EINE PHYSISCHE HARDWAREEINHEIT [EN] METHOD AND DEVICE FOR DETERMINING REQUIREMENT PARAMETERS OF AT LEAST ONE PHYSICAL HARDWARE UNIT [FR] PROCÉDÉ ET DISPOSITIF POUR DÉTERMINER LES PARAMÈTRES DE CONTRAINTES D'AU MOINS UNE UNITÉ MATÉRIELLE PHYSIQUE". European Patent 2009059342. Juli 2009.

Florian Mangold und Harald Roelle. „METHOD AND DEVICE FOR DETERMINING REQUIREMENT PARAMETERS OF AT LEAST ONE PHYSICAL HARDWARE UNIT". International Patent WO/2010/025994. März 2010.

Schutzrechte: Validierung von Software unter potentiell defekter Hardware

Florian Mangold und Harald Roelle. „[DE] VERFAHREN UND VORRICHTUNG ZUM ANALYSIEREN EINER AUSFÜHRUNG EINES VORGEGEBENEN PROGRAMMABLAUFS AUF EINEM PHYSISCHEN RECHNERSYSTEM [EN] METHOD AND DEVICE FOR ANALYZING AN EXECUTION OF A PREDETERMINED PROGRAM FLOW ON A PHYSICAL COMPUTER SYSTEM [FR] PROCÉDÉ ET DISPOSITIF D'ANALYSE DE L'EXÉCUTION D'UN DÉROULEMENT DE PROGRAMME PRÉDÉFINI SUR UN SYSTÈME INFORMATIQUE PHYSIQUE". European Patent 2009059340. Juli 2009.

Florian Mangold und Harald Roelle. „METHOD AND DEVICE FOR ANALYZING AN EXECUTION OF A PREDETERMINED PROGRAM FLOW ON A PHYSICAL COMPUTER SYSTEM". International Patent WO/2010/025993. März 2010.

Schutzrecht: Auffinden von Race Conditions, Validierung von Synchronisation

Florian Mangold und Harald Roelle. „Framework zum Auffinden von Race Conditions und zur Validierung von Synchronisation mit Hilfe von Virtualisierung". Schutzrecht Anmeldenummer: 10 2009 050 161.4. Okt. 2009.

Schutzrecht: Validierung von Systemen mit asynchroner Hardware

Florian Mangold und Harald Roelle. „Framework zur Validierung von Systemen mit asynchroner Hardware mit Hilfe von Virtualisierung". Schutzrecht Anmeldenummer: 10 2009 049 226.7. Okt. 2009.