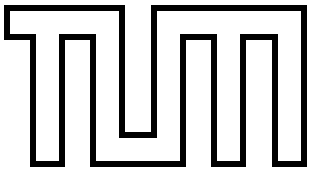


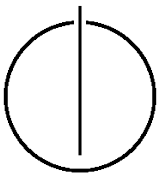
FAKULTÄT FÜR INFORMATIK

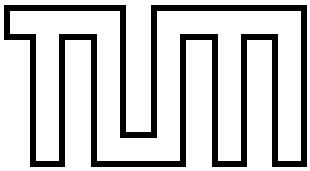
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis Informatik

Statische Performance-Analyse

Thomas Lamperstorfer





FAKULTÄT FÜR INFORMATIK

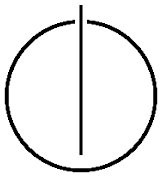
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis Informatik

Statische Performance-Analyse

Static performance analysis

Bearbeiter:	Thomas Lamperstorfer
Aufgabensteller:	Prof. Dr. Dr. h.c. Manfred Broy
Betreuer:	Dr. Florian Deißenböck, TUM
Betreuer:	Jonathan Streit, itestra GmbH
Abgabedatum:	28. April 2011



Ich versichere, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 28. April 2011 Thomas Lamperstorfer

Zusammenfassung

Thema dieser Arbeit war es, die Möglichkeiten der statischen Analyse zur Aufdeckung von Performance-Problemen zu untersuchen. Die Fragestellung lautete: Lassen sich mit Hilfe rein statischer Analysen Performance-technisch signifikante Systemteile in betrieblichen Informationssystemen mit relationalen Datenbanken auffinden? Wie müsste ein solches Analysesystem aufgebaut sein und inwieweit lassen sich damit bekannte Performance-Anti-Pattern nutzen um Optimierungsmaßnahmen vorzuschlagen?

Anhand der Literatur konnte gezeigt werden, dass diese Art der Analyse so noch nicht versucht wurde, jedoch Vorarbeiten zu Teilbereichen der Arbeit wie der Pfadvorhersage vorlagen. Im nächsten Schritt wurden die Anforderungen an ein Performance-Modell erarbeitet, wobei insbesondere Java- und COBOL-Systeme untersucht wurden. Daraus resultierte eine Abstraktionsschicht über dem Quellcode, die nur noch die Performancekritischen Konstrukte wie zum Beispiel Schleifen, Prozeduraufrufe und Datenbankabfragen enthält und daher das „Performance-Modell" genannt wurde. Diese Abstraktion erlaubt es, schnelle und einfache Analysealgorithmen zur Kostenschätzung und zur Erkennung der Anti-Pattern zu entwerfen.

Zur Evaluierung des Konzeptes des Performance-Modells wurden die Rolle und die Auffindbarkeit relevanter Performance-Anti-Pattern im Quellcode untersucht. Wären die problematischen Aspekte nicht im Modell enthalten gewesen, hätte es wahrscheinlich auch keine validen Aussagen über den Ressourcenverbrauch erbringen können. Außerdem wurde eine solche statische Analyse für COBOL-Systeme mit eingebetteten SQL-Datenbankzugriffen implementiert und auf einem in der Industrie verwendetem System evaluiert. Dabei zeigte sich, dass die statische Analyse in der Lage ist, eine kostengünstige Laufzeitabschätzung zu erbringen, die gut mit den dynamischen Messungen korreliert und die zu optimierenden Programmteile auffindet. Somit stellt die statische Performance-Analyse definitiv eine Alternative zur gängigen Praxis der Performance-Tests, Lasttests und Profiling-Verfahren dar, zumal diese aufgrund der benötigten realistischen Testfälle und Testumgebung oft schwer anwendbar sind.

Abstract

The topic of this master thesis was to investigate the possibilities of static source code analysis to discover performance problems. The question to be answered was if performance problems in enterprise information systems connected to a database can be found by means of purely static analysis. How would one build such an analysis tool and to what extend can performance-antipattern help to optimize the performance of the system?

Literature research showed, that this kind of analysis has been investigated in many different settings but not for enterprise information systems. So the next step was to work out the requirements especially for Java and COBOL systems. This resulted in an abstraction layer above the source code called the "performance model". The layer consists of all performance relevant commands like loops, calls and queries and allows to build simple and fast performance cost estimation and antipattern detection algorithms upon it.

The evaluation of the concept of the "performance model" was based on two tests. The first one checked whether the critical performance aspects of the performance antipattern are contained within the model. If this would not have been the case, the model would surely not be able to evaluate the performance of a software system correctly. For the second test the static performance analysis tool described above was implemented and used to estimate the costs of an COBOL software system still running in a large company. The results showed that this approach can identify the expensive and therefore the optimizable parts of the system. Furthermore, the findings correlate with the results of the widely used dynamic analyses like performance tests, software profiling and load tests. As these techniques have some disadvantages like needing realistic test cases, test environments and simulations which can be quite expensive for mainframe systems, static analysis is definitely an alternative worth considering.

Inhaltsverzeichnis

Zusammenfassung vii

Abstract ix

I. Einleitung und Theorie 1

1. Einleitung 3
1.1. Relevanz des Themas . . . 3
1.1.1. Derzeitiger Stand der Technik . . . 3
1.1.2. Motivation der statischen Performance-Analyse . . . 3
1.2. Problemstellung . . . 5
1.3. Beitrag . . . 5
1.4. Zielsetzung dieser Arbeit . . . 5
1.5. Vorgehensweise . . . 6

2. Derzeitiger Stand der Forschung: Statische Performance-Analyse 9
2.1. Statische Performance-Analysen . . . 9
2.2. Werkzeuge zur statischen Performance-Analyse . . . 12
2.3. Pfadvorhersage . . . 13
2.3.1. Datenflussbasierte Pfadvorhersage . . . 13
2.3.2. Statische Pfadvorhersage . . . 14
2.3.3. Pfadhäufigkeitsvorhersage . . . 14
2.4. Abschätzung der Schleifendurchläufe . . . 16
2.5. Fazit . . . 16

3. Derzeitiger Stand der Forschung: Performance-Anti-Pattern 17
3.1. Anti-Pattern . . . 17
3.2. Allgemeine Performance Anti-Pattern . . . 17
3.2.1. The Ramp [39] . . . 18
3.2.2. Tower of Babel [40] . . . 18
3.2.3. The God Class [38] . . . 18
3.2.4. Not Optimizing For The Common Case [37] . . . 19
3.3. Performance Anti-Pattern: Datenbankabfragen . . . 19
3.3.1. Circuitous Treasure Hunt [38] . . . 19
3.3.2. The One Lane Bridge [38] . . . 19
3.3.3. Empty Semi Trucks [40] . . . 19
3.3.4. Application Filters [43, S. 230ff.] . . . 20
3.4. Automatische Erkennung von Pattern . . . 20

4. Beitrag: Performance-Modell **23**
4.1. Definition der Arbeitsgrundlage . . . 23
4.2. Definition des allgemeinen Performance-Modells . . . 23
4.2.1. Kostenberechnungen auf dem Performance-Modell . . . 25
4.2.2. Diskussion des Performance-Modells . . . 26
4.3. Performance-kritische Befehle und Ausdrücke in Programmiersprachen . . . 26
4.3.1. Programmiersprachen allgemein . . . 27
4.3.2. Prozedurale Programmiersprachen . . . 27
4.3.3. Prozedurale Programmiersprachen: COBOL . . . 29
4.3.4. Objektorientierte Programmiersprachen: Allgemein . . . 31
4.3.5. Objektorientierte Programmiersprachen: Java . . . 33
4.4. Bewertung der Kosten für imperative Programmiersprachen . . . 34
4.4.1. Kosten pro Element . . . 35
4.4.2. Statische Kostenbestimmung für SQL-Queries . . . 35
4.4.3. Einfluss der Performance-Anti-Pattern auf die Kostenberechnung . . . 36
4.4.4. Bestimmung der Kosten von Befehlssequenzen oder Blöcken . . . 37

II. Evaluation **39**

5. Evaluation des Performance-Modells anhand der Performance-Anti-Pattern **41**
5.1. Definition des perfekten Performance-Modells . . . 41
5.2. Evaluation des Konzeptes anhand der vorgestellten Performance-Anti-Pattern 42
5.2.1. The Ramp . . . 42
5.2.2. Tower of Babel . . . 43
5.2.3. The God Class . . . 43
5.2.4. Not Optimizing For The Common Case . . . 43
5.2.5. Circuitous Treasure Hunt . . . 44
5.2.6. The One Lane Bridge . . . 44
5.2.7. Empty Semi Trucks . . . 44
5.2.8. Application Filters . . . 45
5.2.9. Fazit . . . 45

6. Fallstudie: COBOL **47**
6.1. Auslesen des COBOL-85-Codes . . . 47
6.1.1. Kombination aus Lexer und Grammatik . . . 47
6.1.2. Island Grammars . . . 48
6.1.3. Lexer-basiertes Parsen mit Pattern-Matching . . . 48
6.2. Das COBOL-Performance-Modell . . . 50
6.2.1. Befüllung des Performance-Modells . . . 52
6.2.2. Kontrollfluss und Kostenanalyse . . . 53
6.3. Performance-Anti-Pattern in COBOL . . . 61
6.3.1. Erkennung von Performance-Anti-Pattern . . . 61
6.3.2. Implementierte Anti-Pattern . . . 62
6.4. Beschreibung der Ausgangsbasis der Evaluation . . . 63
6.4.1. Untersuchtes Informationssystem . . . 63

6.4.2. Optimierungshistorie und Messwerte 63
6.4.3. Konfiguration der Analyse 64
6.5. Evaluation der Praxistauglichkeit der Analyseergebnisse der Fallstudie . . . 64
6.5.1. Frage 1: Ist die Analyse in vernünftiger Zeit durchführbar? 65
6.5.2. Frage 2: Ist die Kostenschätzung der Datenbankzugriffe auf Basis der Tabellengrößen brauchbar? 66
6.5.3. Frage 3: Wie wirkt sich die Wahl der Parameter auf die Analyse aus? 67
6.5.4. Frage 4: Sind die Optimierungsschritte nachvollziehbar? 70
6.5.5. Frage 5: Zeigen die Schätzungen die Einsparungen der Optimierungen? 76
6.5.6. Frage 6: Korreliert die statische Schätzung mit den Messdaten der dynamischen Analyse? 77
6.5.7. Frage 7: Bringt die Performance-Anti-Pattern-Suche brauchbare Ergebnisse? 80
6.5.8. Fazit 81

7. Abschluss **83**
7.1. Zusammenfassung 83
7.2. Ausblick 84

Literaturverzeichnis **85**

Teil I.

Einleitung und Theorie

1. Einleitung

1.1. Relevanz des Themas

1.1.1. Derzeitiger Stand der Technik

Softwaresysteme sind seit vielen Jahren elementare Teile der Wertschöpfungskette von Unternehmen und deren Betriebskosten ein fester Punkt auf der Ausgabenseite der Bilanz. Dabei erhöhen Performance-Defizite in Software-Systemen diese Kosten durch zusätzliche Hardware, nutzungsabhängige Lizenzgebühren und Wartezeiten der Benutzer. Dies fällt aber bei der Inbetriebnahme der Software oft nicht auf, da die Software auf der dafür vorgegeben Hardwareumgebung ja die geforderte Leistung erbringt und das Optimierungspotential nicht bekannt ist.

Da sich die Geschäftsprozesse des Unternehmens jedoch mit dem Markt zusammen weiterentwickeln müssen, muss auch die Software an die neuen Anforderungen angepasst werden. Typische Beispiele hierfür sind erhöhte Datenaufkommen oder die Erweiterung der Funktionalität. Werden diese Veränderungen nicht mit der nötigen Sorgfalt durchgeführt, so ,,altert'' die Software.

Laut David Parnas zeigt sich dieser Alterungsprozess vor allem daran, dass die Software fehleranfälliger wird, sich immer schwerer anpassen lässt sowie Performance-Probleme deutlich werden [30]. Überschreitet das Softwaresystem dabei seine maximale Leistungsgrenze, droht die Wertschöpfungskette abzureißen und das Unternehmen befindet sich plötzlich in einer kritischen Situation. Daher sollte die frühzeitige Erkennung von Performance-Defiziten und deren Behebung ein zentraler Teil der Software-Qualitätssicherung sein.

Eine Performance-Optimierung lohnt sich aber nur dann, wenn dies eine spürbare Auswirkung auf die Gesamtlaufzeit des Systems hat, also wenn ein Hauptkostenverursacher verbessert wird. Welche Systemteile welche Kosten verursachen und wo die kostspieligsten Programme zu finden sind, wird derzeit vor allem mit Hilfe von dynamischen Verfahren, also Performance-Tests, Lasttests und Profiling-Verfahren analysiert, die jedoch deutliche Nachteile haben.

1.1.2. Motivation der statischen Performance-Analyse

Um zu verstehen weshalb die statische Performance-Analyse für die Optimierung nützlich sein kann, müssen zuerst die Nachteile der dynamischen Verfahren erarbeitet werden.

Bei den gerade erwähnten Analysetechniken spricht man von dynamischen Analysen, da zur Ergebniserstellung das Programm ausgeführt werden muss. Dabei wird für jedes

Programm über die verwendeten Ressourcen Buch geführt, wodurch die Hauptkostenverursacher und somit die gesuchten Optimierungskandidaten identifiziert werden.

Der große Vorteil der dynamischen Analysen ist, dass zur Erstellung des Analysewerkzeuges das untersuchte Programm nicht verstanden werden muss. Der Kontroll- sowie der Datenfluss ergeben sich aus der Ausführung des Programmes, so dass nur noch mitgespeichert werden muss, wie oft und wie lange ein Programmteil ausgeführt wurde, wodurch die Hauptkostenverursacher im Regelfall sicher identifiziert werden können. Durch diese Reduktion der Analyseaufgabe auf reine Messung ergeben sich allerdings einige Probleme.

Da die Ausführung des Systems die Messgrundlage darstellt, sind dynamische Analysen grundsätzlich auf Einsatzszenarien beschränkt, in denen geeignete Testfälle vorliegen oder ein spezifisches Performance-Problem reproduziert wird. Dazu müssen allerdings auch die Testumgebung, die Last und die simulierte Zeitspanne repräsentativ sein, wodurch insbesondere im Mainframe-Bereich signifikante Kosten anfallen, so dass eine kontinuierliche Überwachung von Performance-Charakteristiken mit diesen Mitteln sehr teuer ist.

Selbst bei der Messung unter realen Bedingungen, ergeben sich Abweichungen vom regulären Systemverhalten, denn das Aufnehmen und Verwalten der Messdaten verändert natürlich den regulären Programmablauf. Diese können erheblich sein, so liegt zum Beispiel die durchschnittliche Verlangsamung der Analyse mit Intels Profiler „VTune" bei einem Faktor acht [42]. Auch besteht die Möglichkeit, dass aufgrund der Systemgröße der Verwaltungsaufwand für die Zähler so groß wird, dass die Analyse nicht mehr durchgeführt werden kann.

Darüber hinaus erlauben es dynamische Verfahren zwar, problematische Systemteile einzugrenzen, die eigentliche Ursache des Performance-Problems aufzeigen können sie jedoch nicht. Das liegt daran, dass dynamische Analysen meist nur große Mengen an Messdaten produzieren, wie zum Beispiel den Ressourcenverbrauch pro Programmabschnitt. Die nicht triviale Aufgabe, aus diesem Zahlengewirr Ansatzpunkte für eine Optimierung zu finden, muss vom Optimierer geleistet werden. Dabei liegt das Performance-Problem oft in einer bestimmten ungünstigen Aufrufkette [31], was jedoch nicht direkt aus dem Analyseergebnis ablesbar ist.

Die Abschätzung des Ressourcenverbrauchs ohne den Code auszuführen, also per statischer Performance-Analyse, ist ungleich komplexer, da damit selbst die Anzahl der Schleifendurchläufe in vielen Fällen nicht bestimmbar ist. Daher müssen in vielen Fällen, in denen das Wissen aus dem analysierten Quellcode nicht ausreicht, Heuristiken für die Schätzung herangezogen werden. Ein weiterer Nachteil der statischen Analyse ist, dass niemals ausgeführte Programmteile kaum erkennbar sind und deshalb in die Kostenberechnung miteinfließen.

Der große Vorteil der statischen Analysen liegt jedoch darin, dass sie zur Erstellung der Messung nur auf den Quellcode angewiesen sind. Testfälle, Hardwareumgebungen und lange Simulationszeiten entfallen. Dadurch lassen sich kostengünstige, aufwandsarme und vor allem schnelle Analysen erstellen.

Während eine dynamische Analyse nur angibt, wo Ressourcen verbraucht werden und nicht warum, kann eine statische Performance-Analyse die gesuchten teuren Aufrufketten

direkt aufzeigen, da diese ja die Grundlage der Kostenschätzung darstellen.

Wie die Auflistung der Vorteile zeigt, wäre die statische Analyse eine sehr gute Alternative wenn die Genauigkeit der Analyse sichergestellt werden könnte. Grundsätzlich gibt es natürlich Szenarien, in denen mit Hilfe von statisch nicht auslesbaren Konstrukten, wie zum Beispiel dynamischen Sprüngen, die Schätzung keine sinnvolle Aussage mehr treffen kann. Die Frage ist allerdings, wie wichtig diese Fälle in der Praxis sind und ob nicht trotzdem vernünftige Aussagen getroffen werden können. Im Gegensatz zur dynamischen Analyse werden dies natürlich keine absoluten Ergebnisse in CPU-Sekunden sein, was aber auch zur Optimierung meist gar nicht nötig. Was zur Optimierung benötigt wird, ist eine priorisierte Liste der Topverbraucher, so dass der Optimierer seine Aufmerksamkeit in der richtigen Reihenfolge auf die kostspieligsten Programme des Systems richten kann und somit die gewünschte Einsparung schnell und zielsicher erarbeitet.

1.2. Problemstellung

- Lassen sich in der Praxis mit Hilfe rein statischer Analysen Performance-technisch signifikante Systemteile und somit Optimierungskandidaten in betrieblichen Informationssystemen mit relationalen Datenbanken auffinden?
- Wie müsste ein solches Analysesystem aufgebaut sein und inwieweit lassen sich bekannte Performance-Anti-Pattern nutzen um Optimierungsmaßnahmen vorzuschlagen?

1.3. Beitrag

- Erarbeitung der allgemeinen Anforderungen eines Performance-Modells zur Durchführung einer solchen statischen Analyse.
- Genauere Betrachtung der Anforderungen an das Performance-Modell für die Sprachen Java, COBOL und SQL.
- Evaluierung des Konzeptes anhand der Prüfung, ob Performance-Anti-Pattern entsprechend kostspielig geschätzt werden können sowie einer prototypischen Implementierung und Ergebnisanalyse für ein COBOL-System mit eingebetteten SQL-Queries.

1.4. Zielsetzung dieser Arbeit

In dieser Arbeit soll untersucht werden ob der Optimierungsprozess für Softwaresysteme in der Praxis mit Hilfe von statischen Performance-Analysen sowie Performance-Anti-Pattern beschleunigt werden kann. Der Fokus liegt auf klassischen betrieblichen Informationssystemen, weshalb auch SQL-Datenbankenabfragen betrachtet werden. Systeme, die sich noch in der Entwicklungsphase befinden, werden ebenfalls nicht betrachtet, denn die Analysebasis dieser Arbeit soll der Quellcode sein und nicht eine Systembeschreibung.

Zur Abschätzung ob ein Programmteil signifikanten Einfluss auf die Performance des Gesamtsystems hat und somit ein Optimierungskandidat ist, wird jedem Programmteil ein Kostenwert zugeteilt. Dafür wird eine Abstraktion des Quellcodes namens Performance-Modell erstellt, das aus teuren Operationen wie zum Beispiel SQL-Queries, Schleifen und Prozeduraufrufen sowie der Programmstruktur besteht. Da aber nicht immer alle nötigen Informationen direkt im Quellcode enthalten sind, müssen Heuristiken zur Abschätzung unbekannter Größen eingesetzt werden. Durch die Verlagerung der Berechnungen auf die Abstraktion wird die folgende Kostenschätzung erheblich erleichtert und beschleunigt.

Basierend auf diesem Konzept soll ein Analysewerkzeug erstellt werden, dass dem Optimierer erlaubt, anhand der Kostenaufstellung der Programmteile die für eine Optimierung relevanten Programme schnell und einfach herauszufiltern. Entscheidend ist also nicht der Absolutwert der Kosten, sondern vielmehr die Aussage ,,Programm A ist teurer als Programm B".

Eine weitere Möglichkeit den Optimierungsprozess zu beschleunigen besteht darin, dass die Analyse allgemein bekannte Problemmuster, so genannte Anti-Pattern, selbstständig erkennt. Zum jeweiligen Muster finden sich dann in der Literatur nämlich passende Optimierungsvorschläge, so dass der Optimierer diese nicht selbst erarbeiten muss. Denn während gewisse Berechnungen selbst mit der optimalsten Realisierung grundsätzlich große Mengen an Ressourcen benötigen, kann manchmal die Analyse feststellen, dass eine bessere Implementierung möglich wäre. Diese Problematik wird unter dem Begriff des Performance-Anti-Pattern zusammengefasst, gleichbedeutend mit einer bekannten, unnötig teuren, nicht optimalen Lösung für ein wiederkehrendes Problem [38]. Wird nun ein Performance-Anti-Pattern in einem laut Performance-Modell kostspieligen Bereich des Systems gefunden, so wird dieses als Optimierungskandidat ausgegeben.

Bei der Kombination des Performance-Modells und der Anti-Pattern stellt sich die Frage, ob der Fund eines Anti-Pattern eine Auswirkung auf die Kostenberechnung haben soll, es also eine Art ,,Kostenstrafe" für Programmteile mit Anti-Pattern gibt. In dieser Arbeit wurde dieser Weg nicht verfolgt, denn der Anspruch an das Performance-Modell ist, dass solche nicht optimalen Konstruktionen sowieso als entsprechend teuer erkannt werden. Eine künstliche Kostenstrafe würde hierbei nur die Kostenschätzung verzerren. Stattdessen werden die Performance-Anti-Pattern dazu verwendet, unnötig kostspieligen Programmkonstrukten einen Namen zu geben und somit bekannten Optimierungsstrategien zuzuordnen.

1.5. Vorgehensweise

Um diese Ziele zu erreichen wurde das in Abbildung 1.1 dargestellte Vorgehen gewählt. Daraus ergibt sich, dass der nächste Punkte der Gliederung der Arbeit die Literaturrecherche ist und zwar zu den Themen statische Performance-Analyse (Kapitel 2) und Performance-Anti-Pattern (Kapitel 3) sowie deren relevanten Teilgebieten. Daraus werden in Kapitel 4 die Anforderungen an ein Performance-Modell abgeleitet und eine Kostenfunktion beschrieben.

Die Evaluierung des Ansatzes erfolgt einerseits theoretisch anhand der Performance-Anti-Pattern in Kapitel 5. Kapitel 6 enthält die praktische Validierung des Konzeptes. Diese wurde mit Hilfe der Überprüfung der Ergebnisse des implementierten Analysewerkzeugs bewerkstelligt, dass das betriebliche COBOL-Softwaresystem untersuchte. Eine Zusammenfassung sowie Vorschläge für die nächsten Schritte finden sich im letztem Kapitel 7.

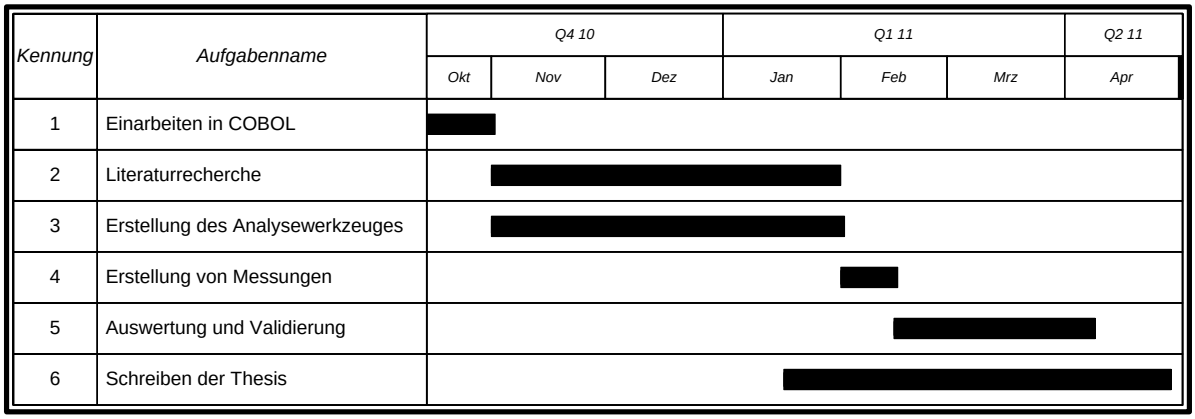


Abbildung 1.1.: Zeitlicher Ablauf der Arbeit

2. Derzeitiger Stand der Forschung: Statische Performance-Analyse

Um eine statische Performance-Analyse durchführen zu können, benötigt man Wissen und Ergebnisse aus einer Vielzahl von Forschungsgebieten. Denn um eine Kostenabschätzung implementieren zu können, müssen viele Teilaspekte des Systems verstanden werden. In diesem Kapitel werden daher neben ähnlichen Forschungsarbeiten und statischen Werkzeugen, Ergebnisse zu zwei Teilgebieten der Problemstellung aufgeführt. Dies ist erstens die Vorhersage ob gewisse Pfade durchlaufen werden sowie zweitens die Abschätzung der Anzahl der Schleifendurchläufe. Beides sind elementare Bestandteile einer statischen Performance-Analyse und sollen daher genauer betrachtet werden.

2.1. Statische Performance-Analysen

Zum Thema dieser Arbeit, der statischen Performance-Analyse auf betrieblichen Informationssystemen, konnte in der Literatur keine Untersuchungen gefunden werden. Jedoch gibt es eine Vielzahl von potentiell relevanten Arbeiten. Einige davon werden nun im Folgenden vorgestellt.

Ein sehr vielversprechender Ansatz wurde in [3] und in der folgenden Veröffentlichung [2] vorgestellt. Als Analysegrundlage wird Java Bytecode verwendet, welcher rein statisch untersucht wird. Dazu wird ein Kontrollflussgraph erstellt, auf dessen Basis nach einigen Vereinfachungen Kostenfunktionen in Abhängigkeit von den Eingabedaten erstellt werden. Das resultierende Gleichungssystem muss anschließend transformiert werden und wird dann per Prolog gelöst. Leider konnte keine Veröffentlichung gefunden werden, in denen die erzielten Ergebnisse der Kostenschätzung vorgestellt und validiert werden, so dass keine Aussage über die Praxistauglichkeit des Verfahrens zur Optimierung von Softwaresystemen gemacht werden kann.

Boogerd und Moonen untersuchten, wie sich die Ergebnisse von Softwareinspektionen mit Hilfe von statischem Profiling priorisieren lassen [6]. Dies erschien ihnen notwendig, da die Ergebnisse einer solchen Untersuchung eine große Anzahl an Warnungen und Fehlalarmen enthalten, die dieses Verfahren ohne Filterung unbrauchbar machen würden. Deshalb wurde eine Priorisierung der Ergebnisse in Abhängigkeit der Ausführungswahrscheinlichkeit des Programmteils in einem zufälligen Durchlauf erstellt. Dynamisches Profiling war hier vor allem aufgrund des Einsatzbereichs (embedded systems) sowie der mangelnden Skalierbarkeit solcher Lösungen ausgeschlossen worden.

Zur Bestimmung der Ausführungswahrscheinlichkeit wird für jede Warnung ein Systemabhängigkeitsgraph erzeugt, der aus Programmstellen (Knoten) sowie Kontroll- und

Datenflüssen (Kanten) besteht und alle die untersuchte Programmstelle betreffenden Anweisungen enthält. Für jeden Pfad werden nun die Wahrscheinlichkeiten berechnet und daraus die Wahrscheinlichkeit für die potentielle Problemstelle errechnet. Dabei werden für die Berechnung der Pfadwahrscheinlichkeiten auch die im nächsten Abschnitt vorgestellten Techniken von Ball und Larus[5] sowie Wu und Larus [49] verwendet. Dabei zeigt sich, dass komplexere Abschätzungen kaum Genauigkeitsgewinne gegenüber simpleren Algorithmen bringen.

In der abschließenden Evaluierung ergibt sich, dass die Einschätzung der Ausführungswahrscheinlichkeit mit der dynamisch berechneten Verteilung korreliert und für die Filterung brauchbar erscheint.

Eine spätere Erweiterung [7] untersucht neben der Wahrscheinlichkeit der Ausführung eines Pfades auch noch dessen Ausführungshäufigkeit. Dabei basiert ein Ansatz auf der Annahme, dass alle Pfade gleich wahrscheinlich sind, ein anderer auf Heuristiken, sowie ein dritter auf Abschätzungen der möglichen Variablenwertemengen im aktuellen Kontext (Value Range Propagation). Auch hier zeigt sich, dass die Algorithmen, die mehr Informationen berücksichtigen, wenn dann nur überschaubare Verbesserungen für das Profiling ermöglichen und dass die Testergebnisse je nach untersuchter Softwareart deutlich schwanken.

Im Gegensatz zu dieser Arbeit wurden von Boogerd und Moonen zwar Grundlagen eines statischen Profils genau untersucht, jedoch die errechneten Wahrscheinlichkeiten und Ausführungsfrequenzen nicht zu einer Kostenabschätzung zusammengeführt. Weiterhin lag der Fokus nicht auf Performance-Optimierung und es wurden keine Datenbanken betrachtet.

Ähnlich wie in dieser Arbeit verwendet Cortellessa et al. Anti-Pattern um die Interpretation der Ergebnisse der Performance-Analyse zu erleichtern. Allerdings liegt der Fokus seiner Arbeit auf der Entwicklungsphase der Software. Daher werden als Analysegrundlage die Anforderungen sowie das Systemmodell verwendet. Über eine regelbasierte Suche werden Anti-Pattern aufgespürt und sogleich mit potentiellen Verletzungen der Anforderungen in Verbindung gebracht [13].

Im Bereich des parallelen, verteilten Rechnens, zum Beispiel auf Supercomputern, spielt das Thema der Performance-Analyse naturgemäß ebenfalls eine wichtige Rolle. So versuchten beispielsweise Cong et al. [12], ein Framework zur automatischen Performance-Optimierung zu erstellen. Dabei werden mit Hilfe dynamischer Analysen und Metriken wie Cache-Misses die besonders kostspieligen Systemteile ausfindig gemacht. Statische Analysen dieser Brennpunkte erzeugen dafür ein Performance-Modell, das mit Hilfe von regelbasierten Suchen nach Anti-Pattern durchsucht wird. All diese Informationen werden in einem Werkzeug verarbeitet, das wiederum nach bestimmten Regeln Optimierungskandidaten sowie mögliche Verbesserungen vorschlägt.

Genau wie in dieser Arbeit untersuchte Chaabane, wie man die Systemleistung mit Hilfe von Anti-Pattern optimieren kann. Da es sich jedoch erst lohnt ein Anti-Pattern zu beheben, wenn die beinhaltende Programmstelle einen signifikanten Teil der Systemressourcen benötigt, kombiniert er die Pattern-Funde mit den Performance-Messungen. Im

Gegensatz zu dieser Arbeit verwendet er zur Messung allerdings dynamische Analysen und beschränkt den untersuchten Bereich der Anti-Pattern auf die Interaktionen zwischen dem Programm und seiner Datenbank [9].

Weiterhin beschreiben Chaabane und Balmas in einem Erfahrungsbericht aus der industriellen Praxis die Performance-Optimierung eines Legacy-Systems im Bankenumfeld [10]. Diese Software wurde in einer proprietären, prozeduralen Sprache geschrieben und war über die Jahre auf über 10 Millionen Zeilen Code angewachsen. Da das Performance-Problem auf die Datenbank zurückgeführt werden konnte, untersuchen sie die Datenbankzugriffe auf potentielle Problemstellen. Dazu wird ein Syntaxbaum aus den Informationen des Quell- sowie des Byte-Codes erstellt, der mit Hilfe eines XML-basierten Verfahrens nach Anti-Pattern durchsucht wird. Zur Einschränkung der Ergebnismenge werden nur Treffer, die besonders häufig ausgeführt werden, ausgegeben. Um die Häufigkeit zu bestimmen wird allerdings, im Gegensatz zu dieser Arbeit, eine dynamische Laufzeitanalyse verwendet.

Was durch die Verwendung von standardisierten Ausführungsumgebungen und Frameworks möglich ist, zeigt [31]. Basierend auf der JavaEE-Technologie, können unabhängig von dem verwendeten Applikationsserver Anwendungen überwacht werden. Somit erhält man auf einfache Art und Weise eine Vielzahl von dynamischen Daten, zum Beispiel die durchlaufenen Pfade. Da allerdings JavaEE auch ein komponentenbasiertes Framework ist, lassen sich aus dem Quellcode zusätzlich noch viele Metainformationen extrahieren. Somit lässt sich zum Beispiel die Länge einer Datenbanktransaktion rekonstruieren, die allgemein eine recht Performance-kritische Größe darstellt. Ebenso können Codeblöcke aufgrund der Metainformation Rollen zugeordnet werden, wodurch eine ganze Reihe von Anti-Pattern durch simples Abprüfen der Erwartungen an diese Rolle gefunden werden kann.

Statische Performance-Analyse wird auch im Bereich der Echtzeit-Systeme untersucht. Hier ist vor allem die obere Schranke der Ausführungszeit (Worst Case Execution Time) interessant, da diese zum Beispiel in sicherheitskritischen Systemen wie Flugzeugen garantiert werden muss. Bei einer solchen Analyse gibt es viele Probleme, die sich nicht per se automatisch lösen lassen. Puschner [34] führt beispielsweise auf:

- Die oberste Schranke der Schleifendurchläufe lässt sich meist nur dann aus dem Code extrahieren, wenn diese explizit im Code angegeben ist.
- Die Verwendung von Rekursionen verhindert ebenfalls in den meisten Fällen eine statische Abschätzung, da die Anzahl der Durchläufe vom Datenzustand abhängt, der nur zur Laufzeit bekannt ist.
- Auch Pointer, „GOTO"-Befehle sowie andere dynamische Sprunganweisungen lassen sich nicht statisch analysieren.

Müssen gewisse Zeitgrenzen garantiert werden, gibt es meist keinen anderen Weg als dass jede Schleife von den Entwicklern umgeschrieben oder mit Zusatzinformationen versehen werden muss oder bestimmte Konstrukte nicht verwendet werden dürfen. Im

Bereich der betrieblichen Informationssysteme ist dieses Vorgehen jedoch undenkbar, so dass die Analyse dieser Arbeit alle nicht exakt auslesbaren Konstrukte abschätzen muss.

Auch im Compilerbau werden Performance-Analysen durchgeführt. Da viele Optimierungen sowohl Vorteile als auch Nachteile haben, gilt es korrekte Einschätzungen für die Codestellen zu treffen. Zum Beispiel kann das sogenannte „Loop Unrolling" die Ausführungsgeschwindigkeit einer Schleife verbessern. Dies erhöht allerdings einmalig die Kosten der Programmausführung, so dass die richtige Abschätzung der Ausführungshäufigkeit durch den Compiler kritisch ist. Im Regelfall verwendet man dazu dynamische Analysen, allerdings gibt es auch Ansätze diese statisch zu vermessen [49] [46]. Da der Compiler allerdings nur vollautomatische Optimierungen durchführen kann, überschneidet sich dieser Bereich der Forschung nur zum Teil mit dieser Arbeit.

Neben den hier vorgestellten Verfahren aus der Literatur gibt es auch noch statische Performance-Analysen, die als kommerzielle oder freie Software verfügbar sind.

2.2. Werkzeuge zur statischen Performance-Analyse

Der Begriff der statischen Performance-Analyse erlaubt eine Vielzahl von möglichen Interpretationen und Anwendungsgebieten, weshalb die Ergebnisse der Recherche nach den jeweiligen Gebieten aufgegliedert wurden.

Unter dem Schlagwort „statische Performance-Analyse" findet man zum Beispiel Software wie Findbugs [1] oder PMD [2]. Diese Werkzeuge sind darauf spezialisiert, Problemmuster im Quellcode zu erkennen und benötigen sie keine Kostenanalyse. Stattdessen werden meist recht sprachnahe, inperformante Befehlsketten gefunden, die sich negativ auf die Performance auswirken. Beispiel hierfür ist die Stringkonkatenation mit „+" in Java, die in bestimmten Sprachversionen sehr ineffizient ist. Auch FxCop von Microsoft sowie Coverity [3] fallen in diese Kategorie, denn auch hier sind die vom Tool vorgeschlagenen Performanceoptimierungsmaßnahmen sehr sprachnah und es gibt keine Kostenschätzung des Programmes.

Daneben gibt es zwar eine Vielzahl von dynamischen Profilern, jedoch konnte zu diesem Suchbegriff keine statischen Performance-Analysen gefunden werden.

Nur im Bereich der eingebetteten Systeme findet man schließlich einige Werkzeuge zur Performance-Messung. Da für diese Systeme oftmals Echtzeit-Kriterien garantiert werden müssen, reichen hier die dynamischen Verfahren zum Nachweis meist nicht aus. Daher werden statische Verfahren verwendet, vor allem um die Laufzeitgrenzen im schlimmsten Fall zu berechnen, die sogenannte Worst Case Execution Time (WCET). Bound-T [4] analysiert zum Beispiel statisch Maschinencode auf verschiedenen Prozessoren

[1] http://findbugs.sourceforge.net/
[2] http://pmd.sourceforge.net
[3] http://www.coverity.com/
[4] http://www.tidorum.fi/bound-t/

und bestimmt jeweils die maximale Ausführungszeit. Das Gleiche gilt für aiT WCET Tool [5] sowie das von der Universität Toulouse entwickelte OTAWA [6]. Das von der TU Wien entwickelte CALC_WCET_167 [7] hingegen analysiert Code auf Basis von C-Quellcode, allerdings nur bis zu einer maximalen Länge von 200 Assembleranweisungen. Ebenfalls auf C-Quellcode arbeitet das Open Source Projekt Chronos [8]. Im Gegensatz zu dieser Arbeit müssen diese Tools aber allesamt für die WCET-Analyse grundsätzlich den gesamten Quellcode vollständig verstehen, also zum Beispiel auch die Anzahl der Schleifendurchläufe kennen oder Rekursion ausschließen.

Da weder eine Beschreibung noch eine Implementierung eines passenden Systems für die Aufgabenstellung gefunden werden konnte, sollen als nächstes relevante Veröffentlichungen zu Teilbereichen des zu bearbeitenden Themas vorgestellt werden.

2.3. Pfadvorhersage

Um die Kosten eines Programmes mit statischer Analyse bestimmen zu können, möchte man Informationen über den eigentlichen Kontrollfluss des Programmes haben. Wird zum Beispiel erkannt, dass ein Pfad in einem Programmteil niemals durchlaufen wird, wäre es falsch diesen in der Kostenschätzung mit zu werten. Damit und mit weiteren Aussagen zu Pfaden beschäftigt sich der Forschungszweig der Branch Prediction oder Pfadvorhersage und ist somit hoch interessant für statische Performance-Analysen.

Bei der Analyse von Pfadentscheidungen im Code unterscheidet man zwischen einer auf Wahrscheinlichkeiten basierenden und einer absoluten Aussage. Letztere sagt voraus, dass genau ein Pfad von mehreren möglichen immer ausgeführt wird. Dies nennt man auch einen „static predictor", mit dem potentiell eine durchschnittliche Trefferquote von ca. 90% möglich sein soll [5]. Dieses Ergebnis ist bemerkenswert, denn für ein Programm, in dem alle Pfade mit gleicher Wahrscheinlichkeit durchlaufen werden, hätte diese Schätzmethode maximal eine Trefferquote von 50%, da ja nur genau ein Pfad aus mehreren vorhergesagt werden kann. Wie [17] allerdings zeigen konnte, gibt es zumindest in C-Programmen meist eine sehr dominante Pfadwahl, weshalb dieses Verfahren durchaus Potential hat.

2.3.1. Datenflussbasierte Pfadvorhersage

Eine mögliche Herangehensweise wurde in [19] untersucht. Hier wurde angenommen, dass für die Eingabedaten keine festen Werte sondern Wahrscheinlichkeitsverteilungen vorliegen. Basierend darauf lassen sich nun Pfad- und Schleifendurchgänge errechnen. Angewandt auf zwei Suchalgorithmen zeigte dieses Verfahren gute Ergebnisse. Allerdings bleibt die Frage ob dieses Vorgehen auch auf großen Softwaresystemen angewendet werden kann. Außerdem werden die Wahrscheinlichkeitsverteilungen der Eingabedaten

[5] http://www.absint.com/ait/
[6] http://www.otawa.fr/
[7] http://www.vmars.tuwien.ac.at/ raimund/calc_wcet/
[8] http://www.comp.nus.edu.sg/ rpembed/chronos/

benötigt, die normalerweise nicht vorliegen. Diese müssten vom Benutzer, von dynamischen Messungen oder anderen Analysen kommen, was die Analysetechnik für diese Arbeit erst einmal ungeeignet erscheinen lässt.

Statt Wahrscheinlichkeitsverteilungen von Eingaben zu betrachten, versuchte Patterson Variablen gewisse Wertemengen zuzuordnen [32]. Durch die Datenanalyse des Quellcodes werden diese Wertemengen weiter eingeschränkt. Damit kann das Verhalten von Verzweigungen und Schleifen ebenfalls eingeschränkt werden. Patterson meldet für seine Technik bessere Ergebnisse als die bisher vorgestellten Techniken, allerdings natürlich verbunden mit einem höheren Implementierungsaufwand. Als Analysegrundlage wurde dabei die Testsuite SPEC-FP-92 [9] gewählt, die 14 stark floating-point-lastige mathematische Berechnungen enthält.

2.3.2. Statische Pfadvorhersage

Um die zeitaufwändigen dynamischen Analysen, die ein Compiler zur Optimierung des Codes benötigt, abzukürzen, entwickelten Ball und Larus Heuristiken zur statischen Verzweigungsvorhersage [5]. Diese basieren auf absoluten Pfadentscheidungen und sollen eine durchschnittliche Genauigkeit von 80% für C und Fortran Quellcode erbringen, wobei Sprünge, die ein dynamisches Ziel haben, explizit ausgeschlossen sind. Dabei lassen sich Pfade, die zum Schleifenkopf zurückspringen, recht einfach vorhersagen, denn die intuitive Annahme, dass Schleifen mehrfach ausgeführt werden trifft für die meisten Programme zu. Die naive Annahme, dass bei Verzweigungen die Wahl immer auf den direkt folgenden Pfad fällt, konnte keine brauchbaren Ergebnisse erbringen. Mit Hilfe von sieben Heuristiken, die in einer bestimmten Reihenfolge auf den Code angewendet werden, können die erwähnte 80% Trefferquote erreicht werden. Ein Beispiel für eine verwendete Heuristik ist dabei die „Opcode-Heuristik". Diese sagt voraus, dass bei einer Fallunterscheidung die Bedingung, die einen Integervergleich mit „$\rangle$ 0" enthält, wahr wird, während ein „$\langle$ 0" als falsch ausgewertet wird. Dies wird damit begründet, dass viele Programme negative Zahlen für Fehlerfälle verwenden.

2.3.3. Pfadhäufigkeitsvorhersage

Auf der Vorarbeit von Ball und Larus aufbauend, entwickelten Wagner und Larus ein Verfahren zur Abschätzung der Häufigkeit der Ausführung von Programmblöcken [46]. Dazu erweiterten sie die statischen Vorhersagen zum Beispiel um die Aussage, dass eine Schleife im Schnitt fünf mal durchlaufen wird. Verglichen wurden nun drei verschiedene Verfahren zur Berechnung der Ausführungshäufigkeit von Blöcken innerhalb einer Funktion in C:

- Loop: Zur Bestimmung der Kosten wird angenommen, dass alle Verzweigungen gleich wahrscheinlich sind und Schleifen genau fünf Mal durchlaufen werden.
- Smart: Zusätzlich zu „Loop" werden den Verzweigungen anhand von Heuristiken Wahrscheinlichkeiten zugeordnet. Sagt eine Heuristik einen Pfad voraus, so ist dessen Wahrscheinlichkeit immer 80% und somit die des anderen Pfades 20%.

[9] http://www.spec.org/cpu92

- Markov: Wie „Smart", nur wird nun auch der Kontrollfluss analysiert. Mit Hilfe der Wahrscheinlichkeiten erhält man ein lineares Gleichungssystem, dessen Lösung jedem Block eine Häufigkeit zuweist. So wird beispielsweise eine Schleife, die in ihrem Inneren zu 20% einen Abbruch-Befehl ausführt, nur noch mit drei statt fünf Durchläufen bewertet.

Interessanterweise konnten die beiden intelligenteren Techniken gegenüber dem Loop-Verfahren keine bemerkenswerten Verbesserungen bringen. Wie in Abbildung 2.1 dargestellt, reicht für die Bestimmung der teuersten Blöcke bereits die einfachste Methode aus, um für die Optimierung brauchbare Ergebnisse zu liefern.

Zur Berechnung, wie oft eine Funktion aufgerufen wird, wurden wiederum mehrere Modelle untersucht. Unter den simplen Verfahren stellte sich ein Verfahren namens „Direct" als Sieger heraus, das die Ausführungshäufigkeit einer Funktion als Summe aller Aufrufe errechnet. Dabei wird jeder Aufruf mit der Häufigkeit des ihn umgebenden Blockes gewichtet. Für den Fall, dass die gerade analysierte Funktion sich direkt selbst, also rekursiv aufruft, wird das Endergebnis noch einmal mit fünf multipliziert. Diese wurde neben den dynamischen Methoden noch mit dem bereits vorgestellten Markov-Verfahren verglichen.

Der Vergleich aus Abbildung 2.1 zeigt, dass selbst mit einfachsten Schätzungen gute Ergebnisse erzielt werden können. Wird eine höhere Genauigkeit benötigt, muss mit komplexeren Algorithmen hantiert werden, die in diesem Fall zusätzliche Probleme wie den Umgang mit Rekursion haben.

In [49] wurde ein ähnliches Verfahren vorgestellt, das genauere Wahrscheinlichkeiten sowie einen schnelleren Algorithmus zur Berechnung verwendet.

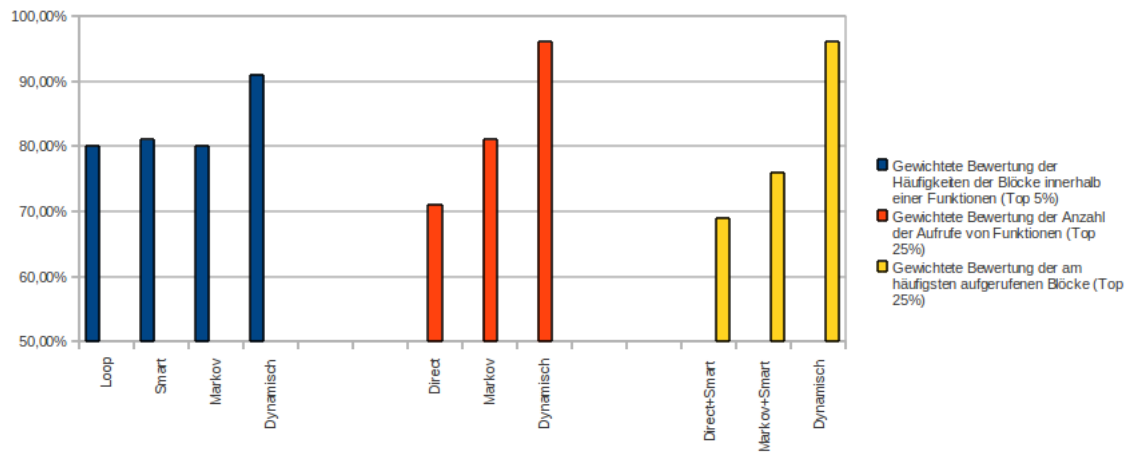


Abbildung 2.1.: Vergleich der Güte der Häufigkeitsanalysen nach [46]. Die Methode zur Errechnung der Zahlenwerte findet sich in [47]

Es gibt auch Versuche, die statische Analyse der Ausführungshäufigkeit und die vorgestellten Heuristiken zusammen mit einem maschinellen Lernverfahren zu verwenden. Während die meisten Verfahren nur geringe Verbesserungen erreichen konnten, meldet ein Verfahren auf den ersten Blick gute Werte [8]. Allerdings wird dazu ein dynamischer Lauf zum Trainieren des Lernverfahrens benötigt. Da aber [17] festgestellt hat, dass bereits

ein dynamischer Durchlauf reicht um das Programmverhalten für Verzweigungen festzulegen, ist der Nutzen dieses Verfahrens gegenüber der rein dynamischen Analyse fraglich.

2.4. Abschätzung der Schleifendurchläufe

Die Abschätzung der maximalen, minimalen oder durchschnittlichen Schleifendurchläufe ist vor allem für Echtzeit-Systeme von großer Bedeutung und natürlich auch für die statische Performance-Analyse. Zur Eingrenzung der Werte finden sich viele Veröffentlichungen, denn die Alternative der Annotation der Durchläufe durch einen Entwickler ist aufwändig und fehleranfällig. Viele Schleifen sind jedoch alleine mit den statischen Informationen nicht abschätzbar, zum Beispiel Schleifen über die Anzahl der Ergebniszeilen von Datenbankabfragen. Als Übersicht über den derzeitigen Stand des Gebiets sei hierzu auf [26] verwiesen. Kurz zusammengefasst lassen sich Zählschleifen meist gut schätzen, während ,,while"-Schleifen, Rekursion und Sprünge wie ein ,,GOTO" nicht in den Griff zu kriegen sind. Möglichkeiten die Trefferquote zu erhöhen bestehen in der Analyse des Daten- und des Kontrollflusses, der Ausführung aller Pfade sowie der Erkennung von Schleifenmustern.

2.5. Fazit

Insgesamt lässt sich sagen, dass trotz der unvollständigen Informationen aus dem Quellcode eine statischen Performance-Analyse machbar erscheint. Untersuchungen wie [46], die 76% der kostspieligsten Aufrufe erkennen konnte, zeigen, dass für die in der Praxis verwendeten Programme oftmals doch gute statische Abschätzungen möglich sind, auch wenn diese natürlich ersteinmal keine exakte Kostenbestimmungen sind.

3. Derzeitiger Stand der Forschung: Performance-Anti-Pattern

Nach der statischen Analyse wird nun der zweite wichtige Baustein dieser Arbeit, nämlich die Performance-Anti-Pattern, genauer untersucht und relevante Exemplare vorgestellt.

3.1. Anti-Pattern

Nachdem Gamma et al mit ihrem Buch zum Thema „Design Pattern" [18] einen wahren Pattern-Hype auslösten, dauerte es nicht lange bis deren Gegenstück, die sogenannten „Anti-Pattern" genauer untersucht wurden. Ein Anti-Pattern soll dabei folgendermaßen definiert sein:

> Anti-Pattern: Genau wie ein Design Pattern ist ein Anti-Pattern eine wiederkehrende Lösung für ein häufig auftretendes Problem. Wie das „anti" schon andeutet, bringt die Verwendung allerdings negative Konsequenzen mit sich. [38]

Der Nutzen dieser Muster liegt dabei neben den vorgegebenen Lösungswegen auch in der Definition und Namensgebung des Problemes, wodurch eine vereinfachte Kommunikation und Diskussion des Sachverhaltes möglich wird. Andererseits ermöglicht die Benennung des Konstrukts auch sofort eine Abschätzung des Optimierungspotentials. Denn für viele Anti-Pattern sind genaue Beschreibungen warum dieses so schadhaft ist, zusammen mit Verbesserungsmöglichkeiten und dem möglichen Einsparungpotential bekannt. [4]

3.2. Allgemeine Performance Anti-Pattern

Anti-Pattern wurden in vielen Bereichen der Informatik dokumentiert, von einzelnen Anweisungen bis hin zur Vorgehensweise bei der Softwareentwicklung. Im Regelfall werden allerdings die Einflüsse auf die Performance nicht genauer aufgezeigt, sondern nur als eines von vielen Problemen dargestellt. Im Folgenden werden einige in der Literatur beschriebene Performance-Anti-Pattern vorgestellt. Kriterium für die Auswahl war hierbei die Anwendbarkeit auf betriebliche Informationssysteme mit relationalen Datenbanken. Nur für eine Sprache gültige Muster werden hier nicht aufgeführt, da sie oftmals zu spezifisch sind und je nach verwendeter Umgebung variieren können.

3.2.1. The Ramp [39]

Der Name dieses Anti-Patterns stammt von der Form seines Datenmenge-Laufzeit-Graphen, das heißt die Laufzeit des Programmes wächst nicht linear sondern schlimmstenfalls exponentiell mit der Menge der Eingabedaten. Es handelt sich damit also um ein Skalierungsproblem. Dieses Anti-Pattern ist im Vorhinein besonders schwer zu finden, da erst eine gewisse kritische Menge an Eingabedaten vorliegen muss, bevor sich die negativen Konsequenzen zeigen. Ein Alltagsbeispiel hierfür ist die Verwendung des Bubble-Sort-Sortieralgorithmus. Für kleine Datenmengen ist seine quadratische Laufzeit nicht erheblich, doch mit steigender Last offenbart sich das Performance-Problem. Ein weiteres Beispiel hierfür ist das „Correlated Subquery", zu Deutsch die korrelierte Unterabfrage. Hierbei wird eine Datenbankabfrage in eine andere so geschachtelt, dass die Innere für jeden Datensatz aus der potentiellen Ergebnismenge der Äußeren einmal ausgeführt werden muss. Wiederum gilt, dass dies für wenige Einträge in der Datenbank kein Problem darstellt, jedoch beim Anwachsen der Datenmenge die Kosten überproportional ansteigen.

Zur Behebung wird der kritische, ineffiziente Algorithmus durch einen effizienteren ausgetauscht. Da die Effizienz je nach Datenmenge variieren kann, besteht auch die Möglichkeit, die Algorithmen dynamisch anhand der Gegebenheiten auszuwählen.

3.2.2. Tower of Babel [40]

Dieses Anti-Pattern beschreibt das Problem, dass unterschiedliche Systemteile unterschiedliche Datenformate benötigen und aus diesem Grund Rechenzeit zur Umwandlung in die verschiedenen Formate vergeudet wird.

Zur Optimierung der Performance wird in diesem Fall vorgeschlagen, entlang der für die Laufzeit kritischen Pfade die Umwandlungen zu minimieren und die Datenformate aufeinander abzustimmen.

3.2.3. The God Class [38]

Die „God Class", die auch unter dem Namen „Blob" bekannt ist, beschreibt ein Anti-Pattern, in dem eine komplexe Klasse viele simple Klassen kontrolliert. Da die gesamte Logik an einer Stelle gebunden ist, werden alle Berechnungen auch nur dort durchgeführt, statt in den jeweils zuständigen Klassen. Das hat zur Folge, dass sich der „Blob" seine Daten erst aus der eigentlich zuständigen Entität auslesen muss. Dieses Holen der Informationen könnte durch besseres Klassendesign vermieden werden und verschwendet somit Ressourcen.

Zur Optimierung eines solchen Programmteils würde man versuchen, durch die Umgestaltung der Klassen die Logik und die dazu nötigen Daten in der gleichen Entität zu kapseln.

3.2.4. Not Optimizing For The Common Case [37]

Der Name dieses Anti-Pattern sagt eigentlich bereits alles. Die Fälle, die ein System am öftesten bearbeiten muss, haben normalerweise den größten Einfluss auf die Laufzeit. Wird hier eine Performance-Steigerung erreicht, so wirkt sich diese deutlich stärker aus als für einen seltenen Fall. Aus diesem Grund sollten die Abläufe daraufhin optimiert sein.

3.3. Performance Anti-Pattern: Datenbankabfragen

3.3.1. Circuitous Treasure Hunt [38]

Dieses Anti-Pattern verwendet eine iterative Suche um das gewünschte Datum aus der Datenbank zu holen und ist daher als auch Schatzsuche bekannt. Das heißt, statt eine einzige, komplexe Datenbankanfrage zu stellen, werden viele davon erzeugt, wobei das Ergebnis der Einen die Eingabe der Nächsten darstellt. Da ein Datenbankzugriff aber recht teuer ist und Datenbanken diese Art der Suche wesentlich effizienter bewältigen, stellt ein solches Vorgehen ein Performance-Problem dar.

Zur Behebung dieses Problems gibt es mehrere Vorgehensweisen. Im Regelfall kann man die Suche in einer komplexen Abfrage bewerkstelligen. Dadurch spart man sich den Aufwand der Datenbankzugriffe, erhöht aber die Last auf der Datenbank. Diese Problematik lässt sich zum Beispiel durch das Verändern des Datenbankschemas abmildern, denn wenn das gesuchte Datum direkt abgefragt werden kann reduziert sich der Aufwand. In vielen Fällen dürfte dies jedoch nicht möglich sein, so dass man stattdessen versuchen könnte, durch eine Verknüpfung der Daten in der Datenbank die Suche abzukürzen.

3.3.2. The One Lane Bridge [38]

Dieses Anti-Pattern, auch bekannt unter dem Namen Flaschenhals, beschreibt Ausführungspfade in einem Programm, in dem ansonsten parallele Berechnungen aufeinander warten müssen. Im Bereich der Datenbanken wäre dies zum Beispiel eine offene Transaktion, die mit Hilfe einer Sperre verhindert, dass andere Prozesse lesend oder schreibend auf die Daten zugreifen.

Die Lösung des Problems besteht in der Minimierung des Flaschenhalses. Dies lässt sich zum Beispiel durch geeignetere Algorithmen sowie spezifischere oder verkürzte Sperren erreichen.

3.3.3. Empty Semi Trucks [40]

Jedesmal wenn in einem Programm eine Datenbankabfrage erfolgt, entstehen erhebliche Fixkosten, die unabhängig von der Größe der Antwort der Datenbank sind. Dies führt zu dem Effekt, dass die Kosten für die Abfrage mehrere Datensätze genauso groß sind wie die für ein Datum. Smith prägt dafür die Analogie des Sattelzugs, der jede Packung Chips

einzeln ausliefert statt alle auf einmal in den Laderaum zu laden.

Wenn man die ideale Antwortgröße einer Abfrage kennt, kann man versuchen, mehrere kleine Abfragen zusammenzufassen. Eine recht bekannte Anwendung hierfür ist der „Multi-Row Fetch", der mehrere Datensätze auf einmal holt, um das Preis-Leistungsverhältnis der Abfrage zu optimieren. Das IBM DB2 Performance-Redbook [1, S. 67ff.] führt ein Beispiel auf, in dem durch einen Multi-Row-Fetch von 100.000 Zeilen auf einmal die Laufzeit halbiert werden konnte. Natürlich tritt dieser Effekt nicht nur bei Datenanfragen auf, sondern zum Beispiel auch bei der Netzwerkkommunikation, die allerdings nicht im Fokus dieser Arbeit steht.

3.3.4. Application Filters [43, S. 230ff.]

Eine der grundlegenden Funktionalitäten einer Datenbank ist das Filtern von Daten, welches in SQL im „WHERE"-Teil des Queries erfolgt. Diese Funktionalität wird oft gebraucht und ist daher hoch optimiert. Ein „Application Filter" liegt vor, wenn eine zu große Datenmenge aus der Datenbank geholt wird und das Aussieben der Daten ineffizienterweise per Hand im Quellcode statt in der Abfrage selbst geschieht.

Zur Optimierung muss die Filterung der Daten aus dem Code genommen und stattdessen in der Datenbank durchgeführt werden. Dieses Anti-Pattern wurde neben einer Reihe von anderen auch in [31] verwendet.

In den zitierten Quellen finden sich noch weitere Performance-Anti-Pattern, die allerdings für diese Arbeit weniger relevant erschienen. Im nächsten Abschnitt soll nun die Frage beantwortet werden, wie diese Muster effizient gefunden werden können.

3.4. Automatische Erkennung von Pattern

Das grundlegende Problem, ein Konzept in einem Programm zu finden und beim Namen zu nennen, wird im Rahmen des Programmverstehens schon lange Zeit untersucht, zum Beispiel unter dem Oberbegriff der „Clichée"-Erkennung. Betrachtet werden dabei sämtliche Abstraktionsstufen, von der Erkennung der Systemarchitektur bis zur Erkennung einzelner Algorithmen. Dabei haben sich mehrere Vorgehensweisen als nützlich herausgestellt, die nun kurz vorgestellt werden sollen.

Eine frühere Arbeit zur Erkennung von Mustern in Software wurde von Paul und Prakash verfasst [33]. Zuerst wurde die Verwendung der Standard UNIX Werkzeuge wie sed, awk, grep aufgrund der mangelnden Möglichkeiten zur Erkennung von Strukturen im Quellcode zur Mustersuche ausgeschlossen. Daher wurde ein annotierter Syntaxbaum aus dem Code erstellt, der die Eingabe für einen nicht-deterministischen finiten Automaten darstellt. Der Automat wiederum wurde aus dem Pattern gewonnen und kann somit feststellen, ob es sich um das gesuchte Muster handelt.

Ein ähnliches Vorgehen zur Erkennung von Design Pattern nutzt [29]. Die Suche arbeitet

dabei auf einem „Abstract Syntax Graph (ASG)", welcher aus einer Grammatik für Java erstellt wird. Ein Vorteil dieser Technik im Vergleich zu einer Suche auf Text ist, dass der Baum oder Graph bereits normalisiert dargestellt wird, also zum Beispiel dass „i++" das Gleiche wie „i+=1" ist. Die gesuchten Pattern werden mit Hilfe von UML ähnlichen Graphen beschrieben, die ebenfalls in ASGs übersetzt werden. Dabei können Teilpattern als Teil einer Patternspezifikation verwendet werden. Denn würde der Algorithmus zum Graphenvergleich jedes Patterns jeweils einzeln mit dem ganzen System vergleichen, wären größere Softwaresysteme aufgrund des großen Ressourcenverbrauchs nicht mehr analysierbar. Daher wird auf der Ebene des Quellcodes nur nach Teilpattern gesucht und deren Ergebnisse mit in den ASG eingehängt. Über mehrere Abstraktionsebenen werden nun immer mächtigere Aussagen getroffen bis schließlich im letzten Schritt die Design-Pattern geprüft werden.

Während Ansätze, die vom Quellcode her stückweise abstrahieren, oft eine große Vielfalt an verschiedenen Varianten des gleichen Pattern benötigen, erlauben Top-Down Vorgehensweisen dieses Problem elegant zu lösen [4]. Dazu wird der Code in eine stark abstrakte, mathematische Notation überführt. Dadurch werden zum Beispiel Variationen bei den Datenstrukturen, der Reihenfolge der Befehle und der Abänderungen im Kontrollfluss wegabstrahiert. Lässt sich das so erhaltene Konstrukt nun auf das gesuchte Pattern abbilden, handelt es sich um einen Treffer. Der Nachteil dieses Verfahrens liegt allerdings in der Laufzeit, denn da keine gemeinsamen Teilmuster gefunden werden können, muss der gesamte Quellcode nach allen bekannten Muster durchsucht werden.

Alias [4] verbindet seinen Algorithmus außerdem noch mit einem interessanten Ansatz aus [25]. Dabei wird aus dem Quellcode ein abstrakter Syntaxbaum erstellt, der mit dem Baum des Pattern verglichen wird. Der Trick dabei liegt in der Konstruktion des Baumes, der mit Hilfe von Heuristiken vereinfacht und normalisiert wird, so dass trotz direktem Vergleich eine Vielzahl an Varianten des Algorithmus erkannt wird. Da die Technik recht einfach anzuwenden ist, hat sie im Vergleich mit den anderen Verfahren eine recht gute Laufzeit.

Projekte zur Erkennung von Pattern unterscheiden sich vor allem auch darin, welche Daten sie berücksichtigen. Während die eben vorgestellte Arbeit auf dem Quellcode arbeitet, nutzt [21] nur die Methodenbeschreibungen aus den C-Headern. Diese werden nach Prolog transferiert, in der die Pattern ebenfalls als Prolog-Regeln abgelegt sind, so dass die Suche komplett mit den Mitteln der Sprache erfolgt. Mit dieser Methode liegt die Erkennungsrate von vier implementierten Pattern zwischen 14% und 50%.

Eine weitere mögliche Herangehensweise wird in [20] beschrieben. Hier werden zwei Methoden aus der Bioinformatik, nämlich die Erkennung von Mustern mit Automaten und mit Bit-Vector-Analyse vorgestellt. Diese Analysen laufen jeweils auf dem ganzen, ungefilterten, zu einem String umgewandelten System. Beide erreichen ähnlich gute Trefferquoten, jedoch bietet letztere deutliche Performance-Vorteile gegenüber ersterer. Allerdings sind die Methoden nur auf kleinere Systeme mit wenigen tausend Zeilen Quellcode anwendbar.

Eine andere Möglichkeit zur Erkennung von Algorithmen sind wissensbasierte Lösungen, zum Beispiel das maschinelle Lernen. Ein Ansatz dazu findet sich in [41]. Basierend auf Entscheidungsbäumen werden hier Programmteile und ihre Komplexität zu

einem Vektor transformiert. Ob dieses nun der gesuchte Algorithmus ist, wird als Klassifizierungsproblem mit Hilfe des Lernverfahrens gelöst. Gezeigt wurde, dass sich drei verschiedene Suchalgorithmen mit hoher Präzision auffinden lassen, allerdings wurde kein ganzes Softwaresystem sondern nur relevante Codeausschnitte untersucht.

Alternativ lässt sich natürlich auch ein simpler Pattern-Matching-Algorithmus auf kleine Codeteile anwenden. Obwohl dieser nicht mit den vorgestellten Techniken konkurrieren kann, stellt [4] fest, dass dies für einfache Probleme oft recht effektiv ist. Die Frage ist nur, wie bei großen Systemen eine vernünftige Laufzeit erreicht werden soll. Möglich wäre dies zum Beispiel durch Vorauswahl geeigneter Codestellen, was im nächsten Abschnitt vorgestellt wird.

Eine andere Möglichkeit potentielle Probleme im Quellcode ausfindig zu machen ist die Verwendung von Metriken. Einzelne, für sich alleine, sind noch keine große Hilfe, da sie oft keine Interpretation erlauben, die auf die Ursache des Problems hinweist. Werden mehrere Metriken zusammen verwendet fehlen weiterhin meist die nötige Informationen, um aus den Zahlenanomalien direkt einen Mangel im Code abzuleiten. [24] Jedoch eignen sich die Metriken sehr gut um Kandidaten für Anti-Pattern zu finden. Dazu wird dieses Muster und seine Auswirkungen erst einmal formal beschrieben. Im nächsten Schritt werden nun geeignete Kombinationen aus Metriken gewählt, die ein Indiz für diese Probleme darstellen. Alle im Quellcode gefunden Treffer werden nun per Hand oder mit Hilfe anderer Techniken analysiert [23].

Die in der Literatur gefundenen Anwendungen dieser Techniken konnten zeigen, dass durch dieses Vorgehen Anti-Pattern wie die „God Class" mit allgemein bekannten Metriken wie Kohäsion und Kopplung von Klassen gefunden werden können [24]. Weitere Probleme können mit Hilfe von speziell dafür entwickelten Metriken aufgespürt werden [28]. Auch eine weitere Verfeinerung der Treffermenge mit maschinellem Lernen wurde untersucht [22].

Welches Verfahren dann letztendlich zum Einsatz kommt, hängt wahrscheinlich auch von den vorhandenen Ergebnissen der restlichen Analyse ab. Auf jeden Fall gibt es eine Vielzahl von gut untersuchten Ansätzen zur Auswahl.

4. Beitrag: Performance-Modell

Ziel dieser Arbeit ist es abzuschätzen, welche Programmteile wie viele Ressourcen und wie viel Rechenzeit benötigen. Da aber nicht jeder Befehl einer Programmiersprache eine messbare Auswirkung auf die Laufzeit hat, liegt es nahe, die Komplexität und das Volumen des zu betrachtenden Quellcodes durch eine Beschränkung auf die Performance-kritischen Elemente zu verringern. Dieses abstrakte Modell des Quellcodes soll Performance-Modell heißen und die folgenden Schritte der Bewertung der Kosten vereinfachen. Wie ein solches Modell aussehen kann, welche Punkte bei der Erstellung zu beachten sind und wie eine darauf basierende Kostenanalyse funktionieren kann, wird in diesem Kapitel genauer erläutert.

4.1. Definition der Arbeitsgrundlage

Bevor das Performance-Modell nun eingeführt wird, muss zuerst die Arbeitsgrundlage festgelegt werden. Im Rahmen dieser Arbeit wird davon ausgegangen, dass der Quellcode die Analysebasis darstellt. Möglich wäre natürlich auch das System erst nach einigen oder allen Kompilierschritten zu betrachten und somit von der Vorarbeit des Compilers zu profitieren. Dies setzt allerdings voraus, dass der Quellcode kompiliert und das Arbeitsergebnis danach auch noch auslesbar ist.

Ob und inwieweit diese Vorverarbeitung nötig oder möglich ist, hängt stark von der verwendeten Programmiersprache ab. Die Technik der Makros und des Präcompilers sind zum Beispiel für die Sprache C so elementar, dass man zur Analyse wahrscheinlich die Ausgabe der ersten Ersetzungsschritte des Präcompilers verwenden wird. Für COBOL hingegen bietet sich der unbearbeitete Quellcode an, da hier entsprechende Vorverarbeitungsschritte fehlen. Allgemein gilt es hier abzuwägen, welche Vorgehensweise die bessere Analysegrundlage bietet.

4.2. Definition des allgemeinen Performance-Modells

Das Performance-Modell sei definiert als:

> Ein Performance-Modell ist eine Abstraktion des eigentlichen Programmes und enthält alle zur Abschätzung der Laufzeit relevanten Informationen. Dies sind die Performance-kritischen Programmelemente, deren Verbindung untereinander, sowie benötigtes Zusatzwissen, wie zum Beispiel die erwartete Anzahl der Durchläufe einer Schleife.
>
> Dieses Modell dient als Eingabe für eine Kostenfunktion, die eine Abschätzung des zu erwartenden Ressourcenverbrauchs für zu untersuchende Modellteile abgibt. Die Struktur des Performance-Modells hängt von den Konzepten der

zu analysierenden Sprache ab und versucht möglichst einfache und schnelle Analysen zu erlauben. Der Grundgedanke dahinter ist vergleichbar mit dem Konzept des abstrakten Syntaxbaumes, das zur Vereinfachung der Programmiersprachenanalyse und Kompilierung verwendet wird.

Da das Performance-Modell notwendigerweise an die Programmierparadigma der zu untersuchenden Sprache angepasst ist, muss die Beschreibung sehr abstrakt bleiben. Schränkt man die betrachteten Sprachen auf die imperativ, prozedural geprägten ein, so lässt sich die Definition verfeinern.

Das Programmierparadigma der prozeduralen Programmierung erlaubt es, eine Befehlssequenz in einem Block zusammenzufassen und diesen unter einem Namen dem restlichem System zur Verfügung zu stellen. Auch Kontrollstrukturen wie eine Schleife werden über Blöcke realisiert. Da ein Befehl auch ein Aufruf eines anderen Blockes sein kann, lassen sich diese Blöcke beliebig schachteln. Wird dieser Mechanismus exzessiv genutzt, was in vielen Programmiersprachen dieser Gattung der Fall ist, liegt es nahe das Performance-Modell auf Basis dieser Blöcke zu definieren. Somit lassen sich die Kosten eines Blockes einmalig schätzen und bei jedem Aufruf wiederverwenden, ohne mehrfach denselben Block bewerten zu müssen.

Ein Performance-Modell für eine imperative, prozedurale Sprache ist analog zur formalen Sprachdefinition aufgebaut und gleicht vom Aufbau her dem des abstrakten Syntaxbaum der Sprache.

Das Performance-Modell P ist also ein Baum, der drei Arten von Elementen (F, I und B) enthält. F sind dabei die sprachfremden Strukturknoten, die zum Beispiel den Wurzelknoten „System" bilden. Sprachinterne Strukturknoten I enthalten die prozeduralen Elemente der Sprache, also benannte und unbenannte Blöcke. B enthält den Befehlsschatz der Sprache, also alle Sprachelemente die direkt zu Rechenoperationen führen. Die Blätter eines Knoten $p \in P$ sind $\in B$, deren Väter $\in I$ und deren Väter wiederum $\in I$ oder $\in F$. Dabei gilt, dass eine Tiefensuche auf P die Befehle in ihrer im Quelltext definierten Reihenfolge auflistet, das heißt, in der Folge wie sie ausgeführt werden würden, wenn man alle Sprünge ignoriert. Grundsätzlich werden Sprünge zu anderen Knoten nicht im Performance-Modell abgebildet, sondern sind eine Eigenschaft des Blattes mit dem Sprungbefehl.

Die sprachfremden Knoten wurden eingeführt, um das System zur Verarbeitung passend aufzuteilen, beispielsweise um einen zentralen Wurzelknoten zu erlauben. Ein Beispiel für ein solches Performance-Modell findet sich in Abbildung 4.1

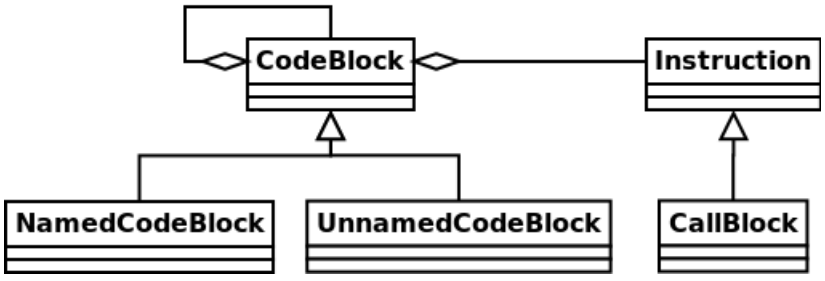


Abbildung 4.1.: Grundgerüst eines prozeduralen Performance-Modells

Ein Beispiel eines Performance-Modells für ein System in einer simplen Sprache findet sich in Bild 4.2. Die Verbindung zwischen dem Aufruf von Programm B durch „CALL B" ist im Bild angedeutet, jedoch nicht Teil der Struktur des Performance-Modells.

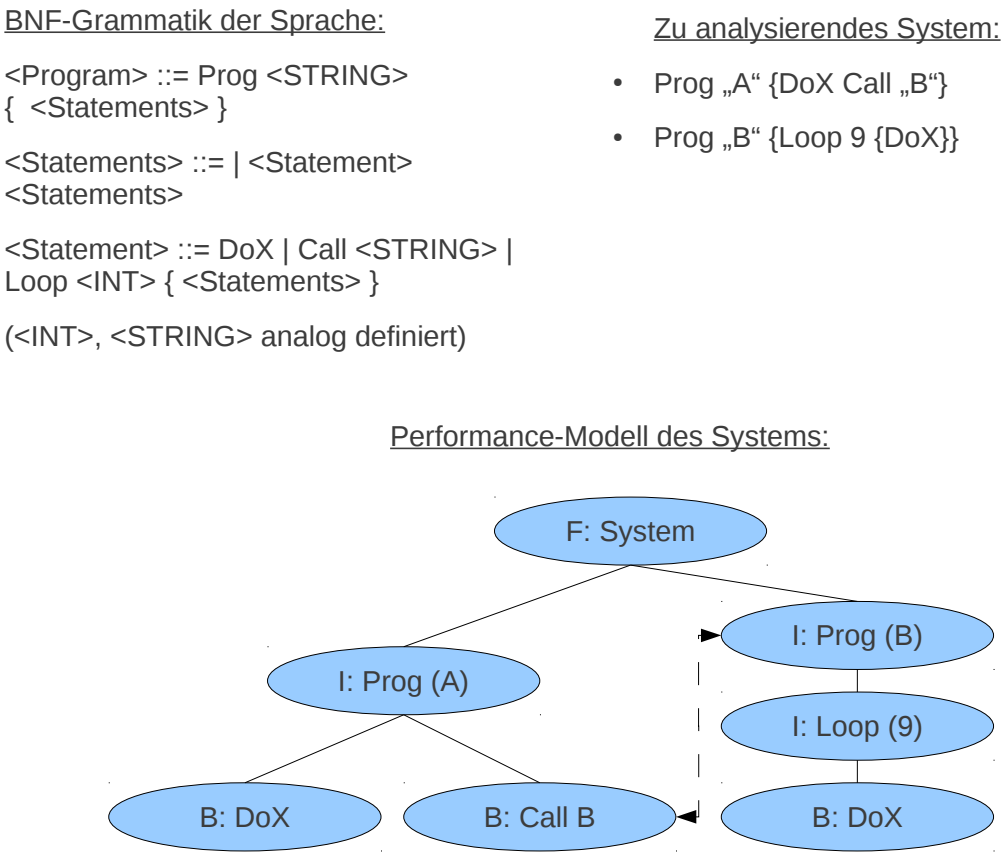


Abbildung 4.2.: Beispiel einer Instanz eines Performance-Modells für eine simple Sprache

Dieses Vorgehen ist natürlich nur dann sinnvoll, wenn die Programmiersprache auch hauptsächlich die ganzen Blöcke aufruft. Sind Sprünge an beliebige Positionen im Inneren eines Blockes die Regel, verschwindet die gewonnene Arbeitsersparnis, da die Befehle jedes Blocks doch wieder mehrfach berechnet werden.

4.2.1. Kostenberechnungen auf dem Performance-Modell

Um nun aus dem Performance-Modell die gewünschte Kostenschätzung zu erhalten, wird ein passender Kostenalgorithmus benötigt. Wie dieser genau aussieht, hängt stark von der Programmiersprache ab, in der das System implementiert wurde. Eine naheliegende Möglichkeit ist es, jeden Arbeitsschritt einzeln zu schätzen und dann eine Ausführung

des Programms zu simulieren, wobei die anfallenden Kosten mitprotokolliert werden. Auf jeden Fall kann dieser Algorithmus auf dem Performance-Modell aufgrund der Abstraktion wesentlich einfacher implementiert werden und die gewünschten Ergebnisse deutlich schneller erarbeiten.

4.2.2. Diskussion des Performance-Modells

Natürlich könnte man sich die Zwischenschicht des Performance-Modelles auch sparen und stattdessen direkt auf dem Quellcode arbeiten. Denn das Einführen einer Abstraktionsschicht hat den Nachteil, dass diese erst einmal erstellt werden muss. Da keine relevanten Informationen verloren gehen dürfen, wird dieses Modell auch eine gewisse Komplexität erreichen. Außerdem besteht auch die Gefahr, relevante Elemente nicht zu erkennen, sowie durch die Ungenauigkeiten im Modell entscheidende Abschätzungsfehler zu begehen.

Andererseits wird durch die Zwischenschicht auch die Wiederverwendung darauf basierender Algorithmen für mehrere Dialekte und Sprachen möglich. Dank der Abstraktion kann die Komplexität, die diese Algorithmen auf dem Modell benötigen, drastisch reduziert werden. So bietet beispielsweise fast jede Sprache verschiedene Varianten an eine Schleife zu programmieren. Eine Berechnungsvorschrift auf dem Quellcode muss alle diese Varianten berücksichtigen. Wird hingegen ein Performance-Modell verwendet, kann diese Vielfalt bereits bei der Erstellung des Modells einmalig berücksichtigt werden und die Algorithmen brauchen nur noch ein Modellelement „Schleife" zu verstehen. Derselbe Effekt tritt auch bei Befehlen im Quelltext auf, die nicht in das Performance-Modell übernommen werden. Auch hier werden nachfolgende Verarbeitungsschritte auf Grund der geringen Sprachvielfalt im Modell vereinfacht. Ein nützlicher Nebeneffekt dabei ist, dass, sobald ein Befehl als unwichtig eingestuft wurde, er auch nicht mehr genau verstanden werden muss. Somit muss bei der Erstellung des Modells unter Umständen nicht mehr der ganze Ausdruck verstanden werden, sondern man kann aufgrund eines gefundenen Schlüsselwortes zum nächsten Befehl übergehen. Weniger Information heißt aber auch weniger Ressourcenverbrauch. Dieser Aspekt ist nicht zu vernachlässigen, da ein Anwendungsfall der statischen Analyse ja große Systeme sind, in denen keine dynamische Analyse mehr möglich ist.

Für die Zwecke dieser Arbeit überwiegen die Vorteile klar, so dass die nächste Aufgabe nun die Einteilung der Sprachelemente in Kategorien ist.

4.3. Performance-kritische Befehle und Ausdrücke in Programmiersprachen

Nicht alle Befehle benötigen gleich viele Ressourcen zur Ausführung und müssen in das Performance-Modell aufgenommen werden. Ziel dieses Abschnittes ist es, Sprachelemente verschiedener Sprachen in Kategorien einzuteilen. Betrachtet werden soll der aktuelle Stand der Technik für betriebliche Informationssysteme, nämlich die Objektorientierung allgemein sowie Java als typischen Vertreter dieses Programmierparadigmas. Andererseits werden laut Studie von DATAMONITOR [35] aus dem Jahre 2008 immer noch 75% der Geschäftsdaten sowie 90% aller Finanztransaktionen mit COBOL verarbeitet. Daher

sollen auch die prozeduralen Programmiersprachen, deren Vertreter heute oftmals aufgrund ihres Alters als Legacy-Sprachen bezeichnet werden, sowie ein typischer Vertreter, nämlich COBOL, analysiert werden.

4.3.1. Programmiersprachen allgemein

Da es eine Unmenge von Programmiersprachen gibt, ist es unmöglich alle Konzepte hier zu behandeln. Allerdings lassen sich diese grob nach grundlegenden Paradigmen der Sprache einteilen. Grundsätzlich unterscheidet man zwischen imperativen und deklarativen Programmiersprachen. Imperative Sprachen bestehen im Endeffekt aus einer Aneinanderreihung von Befehlen, die schrittweise abgearbeitet werden um ein Ergebnis zu erreichen, vergleichbar mit einem Kochrezept. Deklarative Programmiersprachen beschreiben nicht, wie das Ergebnis erreicht wird, sondern nur, wie es aussieht und welche Bedingungen erfüllt sein müssen, also vergleichbar mit einem Rätsel.

Die Erstellung eines Performance-Modells für deklarative Programmiersprachen ist, bis auf das Sonderthema SQL, nicht Teil dieser Arbeit, jedoch sollen die Hauptproblemstellungen kurz angerissen werden. Eine Gattung dieser Sprachen sind die logischen Programmiersprachen, die auf Axiomen und Regeln basieren und versuchen damit die gewünschten Antworten zu finden. Um also die Performance eines solchen Programmes zu bestimmen, könnte man beispielweise versuchen, die Mächtigkeit des beschriebenen Problems abzuschätzen. Zusammen mit den Kenntnissen über den Lösungsalgorithmus ließen sich unter Umständen Ergebnisse gewinnen. Eine andere Variante sind die funktionalen Programmiersprachen, die ein Programm aus einer Menge von mathematischen Funktionen aufbauen. Ein Performance-Modell würde also alle Funktionen enthalten. Eine Verbindung würde genau dann bestehen, wenn eine Funktion zur Ausführung ihrer Berechnung eine andere aufruft. Dabei würde der Analyse zu Gute kommen, dass rein funktionale Sprachen keine Seiteneffekte erlauben, das heißt, dass die Kosten einer Funktion rein von den übergebenen Parametern und nicht vom Programmzustand abhängen.

4.3.2. Prozedurale Programmiersprachen

Wie bereits festgestellt, nutzen imperative Programme Befehle um aus den gegebenen Eingangsdaten und dem Systemzustand schrittweise die gewünschte Ausgabe zu erarbeiten. Prozedurale Sprachen bringen dazu noch das Konzept der Kapselung von Befehlen in benannte Blöcke mit. Dieses Vorgehen bietet eine gute Ausgangslage für eine Analyse, denn was wie wann gemacht wird steht ja zu einem großen Teil explizit im Quellcode. Im Folgenden wurde versucht, grundlegende imperative Konzepte zu erarbeiten, die auch für betriebliche Informationssysteme relevant sind, wobei hier natürlich kein Anspruch auf Vollständigkeit erhoben werden kann.

- Blöcke: Die meisten imperativen Programmiersprachen sind prozedurale Sprachen, das heißt sie erlauben es eine Software in logische Einheiten zu gliedern. Oft werden zum Beispiel Programme in eigene Dateien geschrieben. Diese werden wiederum aus benannten Codeblöcken aufgebaut. Bestimmte Sprachelemente wie Schleifen und Verzweigungen können ebenfalls Blöcke definieren um die

Ausführungsreihenfolge der Befehle geeignet festzulegen. Diese Blöcke formen wie bereits erläutert die statische Struktur des Performance-Modells.

- Einfache Operationen (Arithmetische Operationen, Zuweisung von Werten an Variablen, ...): Grundlegende Operationen werden normalerweise direkt von den Maschinenprogrammen der Hardware unterstützt und werden somit hoch effizient und kostengünstig ausgeführt. Daher werden diese Befehle nicht in das Performance-Modell aufgenommen. Allerdings empfiehlt es sich, die Anzahl der ignorierten Befehle mitzuspeichern, da dies unter Umständen doch einen Einfluss auf die Laufzeit hat.

- Sprünge: Kaum ein Programm wird von oben nach unten jeden Befehl der Reihe nach abarbeiten. Eine Vielzahl von Sprüngen erlaubt es, die Ausführungsreihenfolge zu verändern. Beispiele hierfür sind das oft geächtete „GOTO", Fehlerbehandlungen mit „Exceptions", Fallunterscheidungen wie „IF" sowie Schleifenkontrollelemente wie „BREAK" und „CONTINUE". Diese Elemente können Teil des Performance-Modells sein, je nachdem welche Art von Algorithmen, wie zum Beispiel Pfadvorhersage, verwendet wird. Zusätzlich ist zu beachten, dass mit Sprüngen auch Schleifen gebildet werden können, welche unter Umständen im Performance-Modell als solche behandelt werden müssen.

- Aufrufe: Aufrufe sind wie Sprünge, nur wird am Ende der Ausführung des Sprungziels wieder zurückgesprungen und mit dem auf den Aufruf folgenden Befehl weitergemacht. Aufrufe von Programmen und Programmteilen sind ein grundlegendes Element des Performance-Modells und werden daher in dieses übernommen.

- Schleifen und Rekursionen: Beide bilden einen wichtigen Teil des Performance-Modells und müssen aufgenommen werden. Im Gegensatz zu Rekursionen kann die Durchlaufzahl für Schleifen oftmals sogar direkt aus dem Quellecode abgelesen werden. Ansonsten muss diese geschätzt werden.

- Belegung von Speicher: Jedes Programm läuft im Speicher des Rechners und kann diesen zum Speichern seines Zustandes sowie seiner Berechnungen nutzen. Je nach Sprache kann über verschiedene Mechanismen Speicher angefordert und freigegeben werden. Hier gilt es abzuschätzen, ab welcher Menge von angeforderten Speicher spürbare Auswirkungen auf die Performance auftreten und dies dann entsprechend im Performance-Modell zu berücksichtigen.

- Parallele Ausführung: Werden Teile eines Programmes parallel ausgeführt, ergeben sich neue Probleme für das Performance-Modell. Zur Abschätzung der Laufzeit muss nun nicht nur betrachtet werden, welcher Prozess oder Thread welche Programmteile ausführt. Zusätzlich müssen auch die Auswirkungen von Synchronisierungselementen wie Sperren von Ressourcen (zum Beispiel Semaphore) berücksichtigt werden. Diese Problemstellung wurde in dieser Arbeit nicht untersucht.

- Zugriffe auf externe APIs (Eingabe und Ausgabe, Betriebssystemaufrufe, Datenbankzugriffe, Netzwerkkommunikation, CICS,...): Wird in einem Programm eine

fremde API aufgerufen, werden Befehle ausgeführt, die meist nicht analysiert werden können und sollen. Da solche Aufrufe oftmals aber teuer sind, müssen diese entsprechend abgeschätzt ins Performance-Modell übernommen werden.

Nachdem nun die Bausteine des Performance-Modells bekannt sind, gilt es diese miteinander zu verbinden und mit den nötigen Informationen zu versehen:

- Sprünge und Aufrufziele: Jeder im Performance-Modell enthaltene Sprung oder Aufruf muss mit einer Verbindung zu seinen Sprungzielen versehen werden. Bei statischen Sprüngen, also Zielen die bereits fest im Quellcode stehen, ist dies meist recht einfach. Wird das Ziel jedoch dynamisch, also erst zur Laufzeit bestimmt, muss der Sprung entweder aus dem Modell entfernt werden oder mit Hilfe von Schätzungen einem Ziel zugewiesen werden.

- Schleifen: Die erwartete Anzahl von Durchläufen kann für einen Teil der Schleifen bereits aus dem Quellcode bestimmt werden. Für die verbleibenden sowie für die rekursiven Aufrufketten muss nun eine Abschätzung getroffen werden, zum Beispiel mit den in Kapitel 2 vorgestellten Methoden. Alternativ kann, wie in [7] vorgeschlagen, bei unbekannten Schleifen ein fester Wert angenommen werden, zum Beispiel fünf Durchläufe.

- Kontrollfluss: Jedes System benötigt mindestens einen Einstiegspunkt sowie eine natürliche Abarbeitungsreihenfolge der Befehle. Oftmals wird ein Programm als Startprogramm gewählt und dessen Befehle dann von der ersten bis zur letzten Zeile abgearbeitet. Dies ist jedoch absolut abhängig von der gewählten Programmiersprache und muss im Performance-Modell entsprechend berücksichtigt werden.

- Datenfluss: Wurde beim Einlesen des Quelltextes auch eine Datenanalyse durchgeführt, können diese Informationen für weitere Schlüsse genutzt werden. Ein Beispiel hierfür ist ein Sprung zu einem Block, dessen Name aus einer Variablen gelesen wird. Ohne Informationen über diese Variable ist der Sprung nicht analysierbar. Wurde der Variable aber bei der Initialisierung bereits ein fester Blockname zugewiesen, kann die Analyse dies nun berücksichtigen.

Nachdem die einzelnen Elemente im Performance-Modell verbunden wurden, kann die Kostenanalyse beginnen. Allerdings enthalten die Programmiersprachen oft weitere, sprachspezifische Besonderheiten. Für C sind dies zum Beispiel Makros, ein recht mächtiger Textersetzungspräprozessor, in dem ganze Funktionalitäten abgebildet werden. Ein weiteres Problem ergibt sich bei der Analyse von Pointern. Ein Pointer kann zum Beispiel auf eine Funktion zeigen und so können Funktionen als Parameter an andere Funktionen übergeben werden. Für das Performance-Modell ist solch ein Konstrukt bis auf Sonderfälle nicht mehr analysierbar.

4.3.3. Prozedurale Programmiersprachen: COBOL

COBOL ist eine recht alte Sprache und wurde bereits 1959 entwickelt. Der letzte Standard wurde 2002 veröffentlicht und erlaubt nun sogar objektorientierte Programmierung.

Für diese Arbeit wurde allerdings die Annahme getroffen, dass die meisten COBOL-Programme bereits vor dieser Zeit geschrieben wurden und daher wird der vorletzte Standard, COBOL-85, also eine prozedurale Sprache untersucht. Dieser COBOL-Standard wurde hauptsächlich mit Hilfe des Buches [48] erlernt, das somit die Grundlage für die COBOL-spezifischen Aussagen des folgenden Abschnitts darstellt.

Für die Bausteine des Performance-Modells ergeben sich folgende COBOL-spezifische Anpassungen:

- Programmstruktur: Ein COBOL-Programm ist eine streng gegliederte Textdatei. Die IDENTIFICATION DIVISION enthält den eindeutigen Programmnamen. Die ENVIRONMENT DIVISION enthält die nötigen Informationen zur Interaktion mit der Umgebung des Programmes, also zum Beispiel mit den Datenbanken und dem Dateisystem. Die DATA DIVISION definiert, wie der Name schon sagt, die Beschaffenheit der im Programm verwendeten Daten. Der letzte Abschnitt, die PROCEDURE DIVISION beinhaltet den eigentlichen Befehlsteil der wiederum in SECTIONS und PARAGRAPHS gegliedert wird, welche nichts anderes als benannte Befehlsblöcke darstellen.

- Keine lokalen Variablen: COBOL kennt keine lokalen Variablen und somit existiert nur ein Datenbereich pro Programm. Daraus folgt, dass die SECTIONS und PARAGRAPHS keine Parameter übergeben bekommen.

- Speicherallokation: In COBOL wird der Speicher beim Aufruf eines Programmes einmalig allokiert, eine nachträglich Änderung von dessen Größe ist nicht möglich.

- Sprungziele: Eine Besonderheit von COBOL ist es, dass für Aufrufe angegeben werden kann, dass ein ganzer Bereich von Blöcken ausgeführt werden soll. Dabei wird nur der erste und der letzte Block angegeben und alle Blöcke die dazwischen liegen werden ebenfalls ausgeführt. Somit bestimmt die textuelle Position eines Blockes in der Datei die Ausführungsabfolge.

Für die Verbindungen zwischen den Bausteinen des Performance-Modells gibt es ebenfalls einige Besonderheiten zu betrachten:

- Kontrollfluss: Ein COBOL-Programm wird immer von oben nach unten ausgeführt, wobei der Einstiegspunkt immer der erste Befehl der PROCEDURE DIVISION ist. Wird das Programm nicht explizit terminiert, endet es mit dem Befehl in der letzten Zeile des Quelltextes.

- Dynamische Sprünge und Aufrufe: COBOL erlaubt den Namen des aufzurufenden Programmes in einer Variable zu übergeben, welche ohne Datenflussanalyse im Regelfall nicht analysiert werden kann. COBOL ermöglicht es außerdem, statische Sprungziele per ALTER-Befehl zu verändern, wodurch das Auslesen extrem erschwert wird.

- Unterscheidung von internen und externen Aufrufen: COBOL kennt zwei verschiedene Arten von Aufrufen, je nachdem ob das Aufrufziel ein Programm ist oder ein programminterner Block. Externe Programme können nur per eindeutigem Programmnamen aufgerufen werden, wobei das Programm dann immer nur komplett

von oben nach unten ausgeführt wird. Interne Aufrufe hingegen dürfen nur Namen von SECTIONS und PARAGRAPHS enthalten, wodurch die Rekonstruktion des Kontrollflusses enorm erleichtert wird.

- Rekursion und zyklische Aufrufketten: Die Sprache COBOL sieht keine Vorrichtung vor die es nativ ermöglichen würde rekursive Programme zu schreiben, da der Datenzustand bei einem erneuten Aufruf nicht zwischengespeichert wird. Trotzdem ist es möglich solche Algorithmen zu schreiben, beispielsweise in Code zur Fehlerbehandlung.

Abschließend bleibt zu sagen, dass COBOL-Programme sehr strikt voneinander getrennt sind. Da jeder externe Programmaufruf das gesamte aufgerufene Programm ausführt (und nicht etwa nur Sprungziele innerhalb), erscheint es sinnvoll, die Analyse in eine Programm- und eine Systemphase zu trennen.

Ein Beispiel für ein Performance-Modell für COBOL-Programm findet sich in Abbildung 4.3.

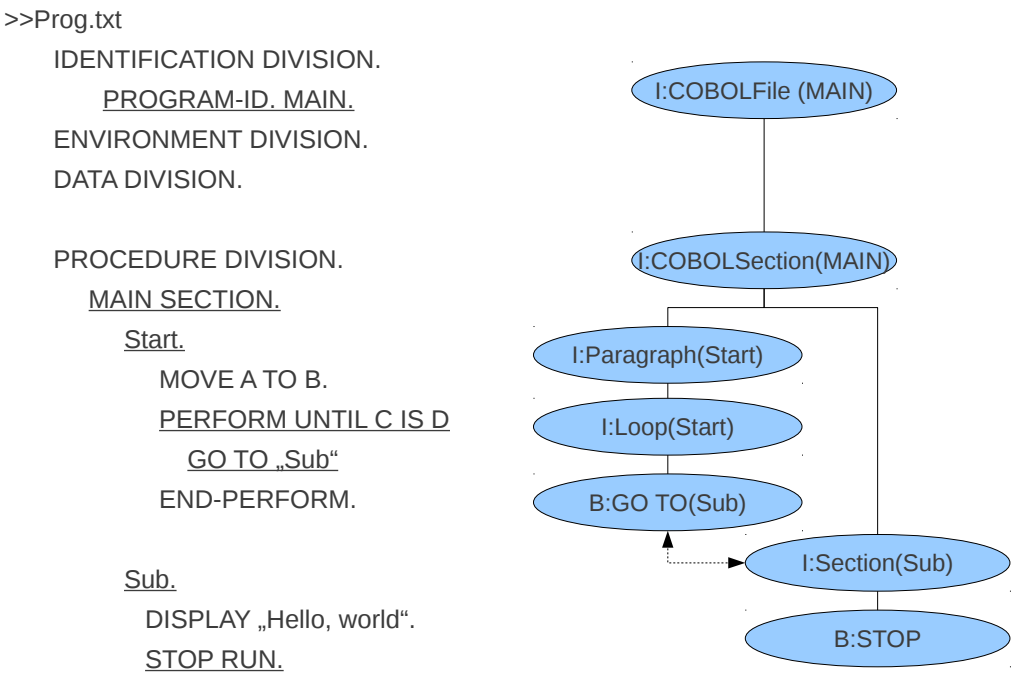


Abbildung 4.3.: COBOL-Quellcode und das zugehörige Performance-Modell

4.3.4. Objektorientierte Programmiersprachen: Allgemein

Für objektorientierte Programmiersprachen gibt es, neben den zumeist imperativen Wurzeln, eine Reihe weiterer Konzepte, die im Performance-Modell berücksichtigt werden

müssen. Eine der ersten erfolgreichen objektorientierten Programmiersprachen war Smalltalk. Deren fünf grundlegende Konzepte fasst Bruce Eckel in [16] folgendermaßen zusammen:

1. Alles ist ein Objekt.
2. Ein Programm ist eine Menge an Objekten, die sich gegenseitig Nachrichten schicken, in denen steht was getan werden muss.
3. Jedes Objekt hat einen eigenen Speicherbereich, der aus anderen Objekten besteht.
4. Jedes Objekt hat einen Typ.
5. Alle Objekte eines bestimmten Typs können die gleichen Nachrichten empfangen.

Daraus folgt für das Performance-Modell, dass der grundlegende Block aus dem das System besteht, die Objekte und deren Methoden sind. Allerdings erlauben die vorgestellten Konzepte recht mächtige Beziehungen zwischen Objekten, so dass die Herstellung der Verbindungen zwischen den einzelnen Elementen des Performance-Modells deutlich schwieriger wird. Grundlegende Probleme ergeben sich in folgenden Bereichen:

- Statische oder dynamische Typisierung: Statische Typisierung heißt, dass bereits bei der Kompilierung des Programmes die Typen der einzelnen Variablen bekannt sind und dementsprechend Typinformationen im Quellcode vorhanden sein müssen. Das Gegenteil davon ist die dynamische Typisierung, bei der erst zur Laufzeit der Typ des Objektes bekannt wird und Typinformationen nicht im Quellcode vorhanden sind. Deshalb kann bei letzterer auch der Compiler kaum mehr Überprüfungen auf Typkorrektheit durchführen. Das heißt aber auch, dass bei einer dynamischen Typisierung die Analyse wesentlich schwieriger wird. Denn wenn eine Methode auf einer Objektvariable ohne Typangabe aufgerufen wird, kann jedes Objekt des Systems, das eine Methode mit einem solchen Namen besitzt, gemeint sein. Natürlich muss auch bei statischer Typisierung nicht der genaue Typ bekannt sein, denn die Typbezeichnung kann ja ein Supertyp des tatsächlich verwendeten Objektes sein, aber die Aufgabe hier den exakten Typ zu finden erscheint wesentlich einfacher. Nur der exakte Typ erlaubt eine eindeutige Zuordnung eines Methodenaufruf zum tatsächlich ausgeführtem Quellcode und somit eine genaue Abschätzung der entstehenden Kosten.
- Initialisierung: Da ein Objekt einen eigenen Speicherbereich hat, muss dieser bei der Erzeugung einer Instanz angelegt und initialisiert werden. Dies passiert im Regelfall implizit, so dass sich der Entwickler nicht darum kümmern muss. Wie groß der Aufwand dafür ist, findet sich nun aber nicht mehr explizit im Code, so dass das Performance-Modell dies gegebenenfalls selbst erkennen und entsprechend werten muss. Enthält zum Beispiel ein Objekt ein hochdimensionales und damit entsprechend großes Zahlenfeld, kann jede Instanziierung signifikanten Einfluss auf die Performance des Systems haben.
- Vererbung und Typsystem: Das Typsystem erlaubt weiterhin, Hierarchien von Typen zu modellieren. Hierdurch ergeben sich Besonderheiten im Kontrollfluss, da der

Subtyp im Regelfall spezielle Zugriffsmöglichkeiten auf die Implementierung des Supertypen hat. Ein Beispiel wäre die Erzeugung eines Subtyps, die einen impliziten Aufruf des Konstruktors des Supertyps zur Folge hat. Diese Mechanismen müssen im Performance-Modell ebenfalls abbildbar sein.

- Polymorphismus: Die fünfte Eigenschaft von objektorientierten Programmiersprachen lautete: Alle Objekte eines bestimmten Typs können die gleichen Nachrichten empfangen. Daraus folgt aber, dass beim Senden einer Nachricht an einen allgemeinen Typ erst zur Ausführungszeit bekannt ist, welches konkrete Objekt und somit welche konkrete Implementierung eigentlich ausgeführt wird. Dies nennt sich auch „Late Binding", da der Compiler nicht mehr wissen kann, welche Codestelle ausgeführt wird, da dies ja erst zur Laufzeit bestimmt wird.

  Aus Sicht der Softwareentwicklung ist diese Abstraktion sehr erstrebenswert und findet sich deshalb auch in vielen Design Pattern. Für die Analyse bedeutet dies aber, dass der Typ des Objektes nicht mehr direkt auslesbar ist und somit zusätzlicher Aufwand wie Daten- und Kontrollflussanalyse sowie Typinformationen nötig werden um eine größere Anzahl von Fällen abzudecken. Der schlimmste Fall wäre eine „perfekte" Abstraktion, bei der zum Beispiel ein recht generischer Quellcode ausgeführt wird, dessen Objekte mit dem Fabrik-Pattern in Abhängigkeit der Eingabe instanziiert werden.

Insgesamt zeigt sich, dass die mächtigen Konzepte der objektorientierten Programmierung zusätzliche Anforderungen an die Analyse stellen. Da es möglich ist, Code abstrakt für einen Typ zu schreiben und nicht nur für ein konkretes Objekt, wird der Aufruf von seiner Implementierung entkoppelt. Zur Erstellung des Performance-Modells empfiehlt es sich daher, die Typinformation, falls sie denn vorhanden sind, aufzunehmen, um somit einer größeren Anzahl von Aufrufen ein Ziel zuordnen zu können. Denn sobald bekannt ist welches konkrete Objekt verwendet wird, lassen sich auch die ausgeführten Codestellen zuordnen. Diese Folgerungen werden natürlich nicht immer möglich sein. Ob der genaue Kontrollfluss nötig ist um Vorhersagen zu treffen, kann allerdings pauschal nicht beantwortet werden, denn solange verschiedene Objekte dieselbe Nachricht in ähnlicher Zeit abarbeiten, hat diese Ungenauigkeit keinen Einfluss auf die Performance-Messung.

4.3.5. Objektorientierte Programmiersprachen: Java

Nachdem allgemein objektorientierte Programmiersprachen betrachtet wurden, soll nun konkret auf einen Vertreter dieser Art eingegangen werden. Java bricht dabei mit dem ersten Konzept von Smalltalk und erlaubt primitive Datentypen, die keine Objekte darstellen. Dies hat allerdings keinen Einfluss auf das in den bisherigen Abschnitten Festgestellte, welches genauso für Java gilt.
Java besitzt eine statische Typisierung. Die Sprachdefinition sieht weiterhin vor, dass alle Blöcke von geschweiften Klammern umschlossen werden und nach jedem Befehl ein Strichpunkt erscheint. Dies erleichtert das Auslesen des Quellcodes. Allerdings enthält diese Sprache noch weitere, für das Performance-Modell relevante Konzepte, die im Folgenden vorgestellt werden sollen.

- Kontrollfluss: Ein Java-Programm hat genau einen definierten Einstiegspunkt in Form einer Methode mit einer bestimmten Signatur. Diese Methode wird befehlsweise von oben nach unten ausgeführt und am Ende dieser terminiert das Programm.

- Reflection: Dieser Mechanismus erlaubt es das Programm während der Ausführung abzuändern. So können zum Beispiel neue Klassen, die im ursprünglichen System nicht enthalten waren, aus Dateien geladen und verwendet werden. Auch Methodenaufrufe lassen sich per Reflection ausführen, ohne dass diese im Quellcode so auftauchen. Offensichtlich ist solch dynamischer Code mit statischer Analyse nur schwer fassbar.

- Annotationen: Diese Technik erlaubt Metainformationen in den Quellcode aufzunehmen. Es gibt aber nicht nur wenige, von der Sprache vorgegebene Annotationen, sondern diese können auch vom Entwickler erstellt werden. Da diese unter anderem auch per Reflection ausgelesen werden können, ermöglichen Annotation beliebige Seiteneffekte. Hier gilt es die verwendeten Annotationen zu verstehen und gegebenenfalls in das Performance-Modell aufzunehmen.

- Generics: Generics erweitern die Mächtigkeit des Typsystems und erlauben Typdefinitionen genauer zu fassen. Dadurch ist es möglich, generische Programme zu schreiben, die für einen beliebigen aber festen Typ aus einer vorgegebenen Gruppe von Typen funktionieren. Die Festlegung auf einen konkreten Typ erfolgt bei der Instanzierung des Objektes. Da Java statisch typisiert ist, wird der Compiler beziehungsweise die Laufzeitumgebung diese Einschränkung auf den gewählten Typ genau kontrollieren.

  Für die statische Analyse bedeutet dieser Mechanismus, neben der neuen Syntax, kaum zusätzlichen Aufwand. Kann der Typ in einer Programmversion ohne Generics statisch abgeleitet werden, so lässt sich dieser auch mit Generics bestimmen.

- Methoden überladen: Java erlaubt es, in einem Objekt mehrere Methoden mit dem gleichen Namen zu haben, solange sich die Liste der Parameter unterscheidet. Für das Performance-Modell bedeutet dies zusätzlichen Aufwand zur Bestimmung der aufgerufenen Codestelle.

Insgesamt bleibt festzustellen, dass Java neue Konzepte einführt, die das Erstellen eines Performance-Modells deutlich erschweren können. Je mehr der Ablauf des Programmes durch dynamischen oder annotierten und somit nicht direkt im Quellcode enthaltenen Code verändert wird, desto schwieriger wird es Aussagen zu treffen. Werden diese Konzepte jedoch mit Bedacht eingesetzt, erscheint eine Vorhersage mit dem Performance-Modell, vor allem aufgrund der strengen, statischen Typisierung durchaus machbar.

4.4. Bewertung der Kosten für imperative Programmiersprachen

Nachdem mit dem Performance-Modell die zu analysierenden Informationen verringert und vereinfacht wurden, wird nun in diesem Abschnitt das Problem der Bewertung der Kostenschätzung einzelner Programmteile sowie die Kostensummation angegangen.

Eine naheliegende Herangehensweise dafür ist es, zuerst die einzelnen Konstrukte abzuschätzen und diese danach schrittweise aufzurechnen. Alternativ lässt sich natürlich auch der Kontrollfluss simulieren und beim Durchlaufen des Modells die Kosten mit zu summieren, was jedoch aufgrund der Vielzahl der möglichen Pfade kaum machbar sein dürfte. Auf jeden Fall reicht eine reine Summation aufgrund der Verzweigungen, Sprünge, Aufrufe und Schleifen hier nicht aus. Ziel dieses Schrittes ist es, die teuersten Programme und somit die Optimierungskandidaten richtig zu benennen. Dass darüber hinaus Aussagen möglich sind, wie zum Beispiel ob ein Programm mit halb so vielen errechneten Kostenpunkten die halbe Laufzeit hat, ist im Regelfall nicht zu erwarten.

4.4.1. Kosten pro Element

Der erste Schritt ist also die Bewertung der Kosten pro Programmkonstrukt. Diese Zuordnungsfunktion der Kosten lässt sich schwer universell definieren. Eine Möglichkeit ist über die Anzahl der nötigen Assembler-Operationen zur Ausführung des Befehls zu gehen. Es bietet sich an die Kosten als Zahl zu erfassen und diesen Punktestand als Kosteneinheit zu betrachten. Ein Beispiel für eine solche Kostenzuordnung, die aus initialen Experimenten entstanden ist, wäre folgende:

- Blockdefinitionen: 0
- Schleifen, Verzweigungen und Aufrufe: 1
- Externe API-Aufrufe sowie SQL-Queries: 500 pro eingebettetem externem API-Aufruf. Bei SQL-Befehlen wird kommen noch die Kosten für die Ausführung des Queries hinzu.
- Speicherzugriffe: 1
- Speicherallokation: 1 pro kb
- Ignorierter Befehl: 1
- Restliche Konstrukte: 1

Aus der Menge der externen API-Aufrufe erscheinen die Datenbanken am wichtigsten, da sie meist ein entscheidender Faktor für die Laufzeit von betrieblichen Informationssystemen sind. Daher sollen die Kosten eines Datenbankzugriffe nun genauer analysiert werden.

4.4.2. Statische Kostenbestimmung für SQL-Queries

Intuitiv ist es klar, dass die Datenbankabfrage eines Datums aus einer Tabelle mit nur einem Eintrag wesentlich billiger ist als die Abfrage eines Datums aus einer Tabelle mit einer Million Einträgen. Oftmals befindet sich das gesuchte Datum aber nicht in genau einer Tabelle sondern in mehreren. In diesem Fall entscheidet das Können des Entwicklers, der die Datensätze miteinander kombinieren muss, wie viele Daten durchsucht werden müssen. Wie bei den Anti-Pattern bereits aufgeführt, gibt es mehrere solche Stolperfallen bei Datenbankabfragen. Deshalb kann die Optimierung von Datenbankzugriffen ein wichtiger Schritt zur Laufzeitreduktion sein.

Moderne Datenbanksysteme besitzen bereits Diagnosewerkzeuge, die anhand der Historie der gestellten Anfragen Performance-Probleme erkennen und sogar selbständig Verbesserungen vorschlagen. Die auch heute noch relevanten Grundlagen vieler dieser Query-Optimierungsmaßnahmen [11] basieren auf einer Arbeit aus dem Jahre 1979 mit dem Titel „Access Path Selection in a Relational Database Management System" von Selinger [36]. Hierin werden die Kosten für ein Query aus Abschätzungen für die CPU- und Eingabe/Ausgabe-Last zusammengesetzt. Diese Zahlenwerte wiederum errechnen sich aus Statistiken über die Größe von Tabellen und Indizes sowie der Selektivität der Bedingungen in der Abfrage, die zusammen die erwartete Ergebnismenge nähern. Insgesamt erhält man:

> Kosten = (Größe der betroffenen Tabellen) + (Gewichtungsfaktor CPU zu I/O)
> * (Größe der Ergebnismenge des Queries)

Für diese Arbeit sind dabei vor allem die Abschätzungen für den Fall bei dem keine Statistiken zur Verfügung stehen relevant. So wird zum Beispiel für die Bedingung auf den Tabellen A und B: „where A.spalte = B.spalte" geschätzt, dass ein Zehntel der Datensätze damit selektiert wird. „where A.spalte greater than B.spalte" hingegen würde nur ein Drittel der gesamten Daten zurückgeben. Leider wurden diese Annahmen nicht durch Zahlenmaterial in der Veröffentlichung gestützt und geben somit nur die subjektive Einschätzung des Autors wieder. Zudem wurden noch Joins und geschachtelte Queries untersucht und Berechnungsvorschriften für die Kosten erarbeitet.

Mit dieser Arbeit wurde das Zeitalter der dynamischen Analyse zur Query-Optimierung eingeläutet, so dass keine weiteren statischen Abschätzungen gefunden werden konnten. Heutige Datenbanksysteme, wie zum Beispiel Oracle, benutzen die gesammelten Statistiken um Optimierungen vorzuschlagen [14].

Eine mögliche Kostenvorschrift zur Berechnung von Datenbankzugriffen wäre also eine Abschätzung der Ergebnismenge anhand der Datenbankgrößen sowie der gerade beschriebenen Selektivitätsheuristik.

4.4.3. Einfluss der Performance-Anti-Pattern auf die Kostenberechnung

Bevor nun die Kosten der einzelnen Konstrukte zusammengefasst werden, sei, wie bereits in der Einleitung geschehen, darauf hingewiesen, dass gefundene Performance-Anti-Pattern nicht gesondert in die Kostenrechnung eingehen. Das heißt, es gibt keine künstliche „Performance-Strafe" für Programmteile die ein solches enthalten. Die Idee dahinter ist, dass das Performance-Modell sowieso die Kosten für eine solche nicht-optimale Konstruktion höher bewerten sollte, als die einer optimaleren Lösung für das Problem. Ob dies wirklich so ist wird im nächsten Kapitel evaluiert, jedoch wird in dieser Arbeit dem Grundsatz gefolgt, diese nicht extra zu bestrafen. Denn ein Anti-Pattern in einem nicht Performance-kritischen Bereich zu optimieren lohnt sich im Regelfall nicht, so dass eine künstliche Verzerrung der Kosten wohl eher kontraproduktiv wäre.

Die künstlichen Strafen wären nur dann sinnvoll, wenn man davon ausgeht, dass das Anti-Pattern nicht im Modell als kostspielige Konstruktion auftaucht. Ein Beispiel hierfür wäre die Ankündigung im Kommentar, dass der folgende Bereich sehr langsam ist. Auch die Beschreibungen oder Namen der Blöcke könnten herangezogen werden um bei Wörtern wie „bubble sort", „simple", „slow" die Kosten zu erhöhen, bei „binary search",

„hash" und weiteren Hinweisen auf effiziente Algorithmen die Kostenschätzung zu verringern. Dies wurde jedoch im Rahmen dieser Arbeit nicht weiter untersucht.

4.4.4. Bestimmung der Kosten von Befehlssequenzen oder Blöcken

Nachdem nun die Kosten für jedes Konstrukt feststehen, gilt es nun zu bestimmen, ob und wenn ja wie oft dieses in die Wertung für die untersuchte Befehlssequenz, im Folgendem allgemein Block genannt, eingeht. Wie bereits festgestellt, lassen sich die Kosten aufgrund von Sprüngen, Verzweigungen und Schleifen kaum als Summe der Kosten der Befehle errechnen. Eine mögliche Lösung dafür wäre, die in Kapitel 2 gezeigten Verfahren zur Pfadvorhersage anzuwenden, welche eine Vielzahl von verschiedenen Strategien zur Gewichtung der einzelnen Kosten ermöglichen.

So könnten beispielsweise Sprünge im Block ignoriert werden oder dazu führen, dass nachfolgende Befehle geringer gewichtet in die Summe eingehen. Spielraum gibt es auch bei Verzweigungen, denn ein „if" könnte zum Beispiel ignoriert werden. Allerdings könnte man sich auch zur Analyse auf einen Zweig festlegen und davon ausgehen, dass genau dieser gewählt wird. Alternativ kann man auch annehmen, dass beide Blöcke zu 50% aufgerufen werden oder dies sogar mit Heuristiken noch genauer abschätzen.

Grundsätzlich gilt jedoch, dass ein Block, der einen anderen Programmteil aufruft, dessen Kosten zu seinen eigenen addieren muss. Da beliebig viele Aufrufe an beliebige andere Blöcke enthalten sein können, kann die Reihenfolge der Auswertung problematisch sein. Hier bieten sich ein Verfahren an, das zuerst Programmteile ohne Abhängigkeiten zu anderen bewertet und dann iterativ mit den nun abschätzbaren Blöcken weitermacht. Allerdings muss dieser Algorithmus auch mit zyklischen Abhängigkeiten umgehen können.

Wichtig ist auch der Einstiegspunkt an dem die Berechnungen beginnen. Falls es nur einen Einstiegspunkt in das Programm gibt, lässt sich bereits für jeden Block einer Software die Ausführungshäufigkeit abschätzen. Ist dies nicht der Fall muss darauf geachtet werden, dass verschiedene Programmabläufe von einzelnen Einstiegspunkten sich nicht gegenseitig die Kostenschätzung beeinflussen. Sind zum Beispiel zwei Programme A und B gegeben, wobei A B aufruft und jedes ist ein Einstiegspunkt, wird A je nach Betrachtung einmal oder gar nicht ausgeführt.

Nachdem das Verfahren alle Blöcke mit Kosten annotiert hat, können die Kosten dem Benutzer präsentiert werden.

Inwieweit dies dann mit den dynamischen Messungen übereinstimmt, lässt sich schwer abschätzen. Wie dieses Kapitel gezeigt hat, müssen zur Erstellung des Performance-Modells sowie zur Pattern-Erkennung eine Vielzahl von Entscheidungen und Annahmen getroffen werden. Deren Auswirkungen sind kaum abschätzbar und werden erst am Ende bei der Auswertung des Ergebnisses spürbar. Aus diesem Grund wurde im Rahmen dieser Arbeit ein Analysewerkzeug für COBOL-Code entwickelt, das im nächsten Kapitel erläutert und evaluiert wird.

Teil II.

Evaluation

5. Evaluation des Performance-Modells anhand der Performance-Anti-Pattern

Zur Evaluation des aufgestellten Konzeptes des Performance-Modells wurde dieses zwei Prüfungen unterzogen. Zum einem die praktische Überprüfung anhand einer Implementierung eines Performance-Modells für ein COBOL-System, für das eine Optimierungshistorie mit Messdaten zum Vergleich vorlag. Zum anderem die theoretische Validierung, die prüft, ob das Modell überhaupt die bekannten Performance-Anti-Pattern teurer bewerten kann, als eine nachträglich optimierte Version des jeweiligen Codes. Wäre dies nicht der Fall, würde sich die Frage stellen, ob das Performance-Modell wirklich eine geeingete Repräsenation zur Kostenschätzung darstellt, da ja offensichtlich einige performancekritische Aspekte nicht darin enthalten sind.

5.1. Definition des perfekten Performance-Modells

Damit festgestellt werden kann, ob eine Performance-Modell ein Performance-Anti-Pattern kospielieger als eine optimierte Version desselben Algorithmuses einschätzen kann, muss zu erst geklärt werden über welche Informationen denn ein mögliches, perfektes Modell verfügen würde. Das hierzu betrachtete, perfekte Performance-Modell würde folgende Fähigkeiten besitzen:

- Schätzungen für Variablenbelegungen: Vergleichbar mit den im Abschnitt 2.3.1 vorgestellten Verfahren, würde das perfekte Performance-Modell die Wertmenge sowie die zugehörige Wahrscheinlichkeitsverteilung der Variablen auf Basis des Quellcodes schätzen. Auf Basis dieser Informationen können nunmehr Fallunterscheidungen und Schleifendurchläufe sehr genau bestimmt werden.
- Wissen über die Datenbank: Um die Wertemengen der Variablen richtig abschätzen zu können, enthält das perfekte Performance-Modell Informationen über die Datenbanktabellen und der Anzahl und Beschaffenheit der Einträge. Es kann somit die Rückgabemenge der Queries richtig einschätzen.
- Keine Schätzung von Dead-Code: Das perfekte Performance-Modell erkennt selbstständig welche Codeteile nicht ausgeführt werden und berücksichtigt diese nicht bei der Kostenschätzung.
- Abschätzung dynamische Kosten SQL: Die Kostenschätzung im perfekten Modell kann jeder Datenbankabfrage den richtigen Kostenwert zuordnen.
- Simulation des Datenflusses: Im perfekten Performance-Modell sind die Wertemengen der Variablen bekannt und somit auch, welche Datenmenge bei welchem Aufruf

mitübergeben wird. So kann jeder Block für jeden einzelnen Aufruf seperat bewertet werden und veränderte Kosten durch andere Parameterkonstellationen fließen in die Kostenschätzung mit ein.

Nachdem das perfekte Performance-Modell vorgestellt wurde, kann nun die Evaluation des Konzeptes anhand der Performance-Anti-Pattern beginnen.

5.2. Evaluation des Konzeptes anhand der vorgestellten Performance-Anti-Pattern

Dieses Kapitel dient zur theoretischen Evaluierung des Konzeptes des Performance-Modells. Dazu soll die Frage beantwortet werden, ob denn das Modell alle performancekritischen Aspekte eines betrieblichen Informationssystems überhaupt erkennen kann und somit eine reale Chance besteht, eine vernünftige Kostenschätzung zu produzieren.

Da Performance-Anti-Pattern bekanntermaßen Konstruktionen mit negativen Auswirkungen auf die Performance beschreiben, sollen diese die Testbasis für die Beantwortung dieser Frage darstellen. Eine für betriebliche Informationssysteme relevante Sammlung dieser Muster wurde bereits in Kapitel 2 erarbeitet und wird im Folgenden abgearbeitet. Zur Erinnerung beginnt jeder Abschnitt mit der Kurzbeschreibung des Anti-Pattern gefolgt von der eigentlichen Evaluation, welche anhand der Prüfung erfolgt, ob das Performance-Modell denn eine optimierte Version des Anti-Pattern als kostengünstiger erkennen kann.

5.2.1. The Ramp

Kurzbeschreibung: Ineffiziente Algorithmen für große Datenmengen.

Dieses Anti-Pattern beschreibt die Beobachtung, dass verschiedene Algorithmen, die die gleiche Aufgabe lösen, je nach Größe der Eingabemenge unterschiedlich effizient sind. Klassisches Beispiel hierfür sind die Sortieralgorithmen. Während es kaum einen Unterschied macht wie man ein paar Elemente sortiert, sind für große Datenmengen nur noch wenige Algorithmen geeignet. Um diese Tatsache auszudrücken, werden in der Komplexitätstheorie die Landau-Symbole verwendet, die den wachsenden Ressourcenverbrauch eines Algorithmuses in Abhängigkeit der Eingabemenge beschreiben.

Das Verbrauchsverhalten eines Algorithmus beim Anwachsen der Eingabemenge kann im Performance-Modell oftmals noch erkannt werden, vor allem wenn man die Ausführung simuliert. Nicht mehr bestimmbar ist jedoch in vielen Fällen die Größe der Datenmengen mit denen ein Programmteil arbeitet. Da das Performance-Modell keine Modellierung der vom System zu verarbeitenden Eingabedaten vorsieht, kann kaum abgeschätzt werden, mit welchen Eingabedaten ein Alogrithmus arbeitet. Damit kann die Kostenschätzung aber auch die Kosten nur grob nähern. Im Regelfall können die Schleifen, die zur Verarbeitung nötig sind, daher nicht ausgelesen werden und die Schleifendurchlaufzahl wird daher mit einem kleinen, vordefinierten Wert belegt. In diesem Fall werden

alle Algorithmen ungefähr die gleichen Kosten haben, wobei für komplexere Vorgehensweisen höhere Kosten zu erwarten sind. Liegt dieses Anti-Pattern aber nun wirklich vor, so liegt die Schätzung deutlich daneben und ein wichtiger Optimierungskandidat kann nicht gefunden werden.

Das Problem kann abgemildert werden, in dem zum Beispiel Rückgabemengen von Datenbankabfragen richtig eingeschätzt und propagiert werden, oder Informationen über Strukturgrößen oder zu erwartenden Datenmengen vom Benutzer vorab angegeben werden. Alternativ könnte man auch über eine Kostenstrafe oder eine Warnung pro bekanntem, ineffizientem Algorithmus überlegen.

Insgesamt bleibt festzuhalten, dass das Performance-Modell seine Kostenschätzung nicht an die Größe der zu erwartenden Datenmenge anpassen kann, da diese zumeist nicht bekannt ist. Das heißt, dass Verarbeitungsschritte mit großen Datenmengen im Regelfall zu billig eingeschätzt werden, vor allem wenn ineffiziente Algorithmen verwendet werden.

5.2.2. Tower of Babel

Kurzbeschreibung: Mehrfache Umwandlungen der Daten aufgrund verschiedener Datenformate.

Das Anti-Pattern „Tower of Babel" wird vom Performance-Modell mit Sicherheit erkannt. Da jede Umwandlungen der Daten in ein anderes Format Kopieroperationen benötigt, oftmals sogar innerhalb von Schleifen über die Datenelemente, entstehen hier auf jeden Fall Kosten. Die optimierte Version würde diese Transformation nicht mehr enthalten und somit günstiger eingeschätzt werden.

5.2.3. The God Class

Kurzbeschreibung: Statt die Logik in den Klassen zu definieren in denen auch die zugehörigen Daten liegen, wird diese in der „God Class" gebündelt. Dadurch wird eine Vielzahl von zusätzlichen Methodenaufrufen zum Daten Lesen und Schreiben nötig, um die Verarbeitung der Daten durchzuführen. Da der Zugriff auf eine lokale Variable im Regelfall billiger ist als ein Methodenaufruf, ergeben sich Performance-Nachteile.

Das perfekte Performance-Modell wird dieses Anti-Pattern sicherlich erkennen, da ein Zugriff auf eine Variable nur eine Operation darstellt, jedoch ein Methodenaufruf sowie die Wertrückgabe mindestens zwei.

5.2.4. Not Optimizing For The Common Case

Kurzbeschreibung: Dieses Performance-Anti-Pattern beschreibt keinen konkreten Algorithmus, sondern nur die Tatsache, dass die Verarbeitungsfolge, die am häufigsten aufgerufen wird, hochoptimiert sein sollte.

Die Bewertung des Performance-Modells anhand dieser abstrakten Beschreibung lässt keine eindeutige Antwort zu, denn da die Art der Optimierung nicht genannt wird,

kann kein Vergleich mit einer optimierten Version des Problems zur Evaluation herangezogen werden. Auch die naheliegende Frage, ob überhaupt die am häufigsten eintretende Verarbeitungsfolge erkannt wird, lässt sich nicht pauschal beantworten. Bestimmt sich die Folge beispielsweise rein aus den Eingangsdaten kann nicht davon ausgegangen werden.

5.2.5. Circuitous Treasure Hunt

Kurzbeschreibung: Das gesuchte Datum wird über mehrere Datenbankabfragen, die jeweils nur Zwischenergebnisse darstellen, erarbeitet, statt einmalig in einer komplexen Anfrage aus der Datenbank gelesen zu werden.

Eine Optimierung dieses Anti-Patterns würde also mehrere Queries, die vielleicht sogar in einer Schleife enthalten sind, entfernen und stattdessen ein komplexes Query einfügen. Da das perfekte Performance-Modell die Kosten der Queries richtig abschätzt, werden die Kosten der einen, komplexen Abfrage richtigerweise als wesentlich billger als mehrere, simple Queries eingeschätzt. Daher wird die Kostenschätzung für die optimierte Version wie erwartet deutlich geringer ausfallen.

5.2.6. The One Lane Bridge

Kurzbeschreibung: Die Laufzeit von ansonsten parallel ausgeführten Berechnungen wird aufgrund von ungünstigen Sperren auf gemeinsam verwendeten Ressourcen signifikant erhöht.

Dieses Performance-Anti-Pattern tritt in betrieblichen Informationssystemen vor allem im Zusammenhang mit Sperren auf der Datenbank auf. Damit das Performance-Modell diese Problematik erkennen könnte, müsste es die Berechnungen simulieren oder Informationen über das Zeitverhalten des Systems ableiten können, um die Auswirkungen der Sperren abschätzen zu können. Dazu müssten zusätzliche Heuristiken sowie Ressourcenmodelle implementiert werden um die teuren Wartezeiten auf die gemeinsamen Ressourcen mit einzukalkulieren. Da ein Performance-Modell jedoch nur die kostspielen Operationen enthält, könnten hier nur die Sperren im Quellcode entsprechend teuer gewichtet werden aber nicht die verursachte Wartezeit, womit sich das Anti-Pattern nicht entdecken lässt. Das Performance-Modell berücksichtigt also den Aspekt der Wartezeit auf gemeinsam verwendete Ressourcen nicht. Es stellt sich allerdings die Frage, ob dies überhaupt mit statischer Analyse erkannt werden kann. Selbst mit Hilfe dynamischer Verfahren lassen sich diese Wartezeit schwer vorhersagen, da minimal veränderte Bedingungen, zum Beispiel durch die Messung selbst, das Problem stark beeinflussen können.

5.2.7. Empty Semi Trucks

Kurzbeschreibung: Beim Informationsaustausch über Systemgrenzen hinweg, zum Beispiel bei der Kommunikation mit der Datenbank, treten hohe Transportkosten auf. Diese bestehen aus einem signifikanten Anteil an Fixkosten für den Verbindungsaufbau sowie einem variablen Kostenanteil abhängig von der Menge der transportierten Daten. Das Anti-Pattern beschreibt dabei den Fall, dass viele Verbindungen aufgebaut werden in

denen jeweils nur geringe Mengen an Daten transportiert werden, statt diese in nur einer Verbindung alle gemeinsam zu holen.

Aus der Beschreibung des Patterns folgt, dass die kritischen Punkte der bei der Erkennung zum einem die Häufigkeit der Datenbankzugriffe ist und zum anderen die korrekte Kostenschätzung der Queries. Da Datenbankabfragen richtig kostengeschätzt werden, werden die Kosten auf jeden Fall korrekt wiedergegeben. Die Häufigkeit ist der Queries ist schwerer einzuschätzen, da aber sowohl die Anzahl der Einträge der ganzen Tabelle, die Anzahl der mit der Abfrage selektierten Datensätze der Datenbank sowie eine Schätzung für Variablen in Schleifen im perfekten Performance-Modell enthalten ist, kann davon ausgegangen werden, dass diese korrekt erkannt wird. Damit zeigt sich, dass dieser Aspekt der Performance im Modell berücksichtigt wird.

5.2.8. Application Filters

Kurzbeschreibung: Statt nur die benötigten Datensätze aus der Datenbank zu holen, werden zu viele Datensätze beschafft, die anschließend ausgefiltert werden müssen.

Da spezifischere Filterangaben in einer Datenbankabfrage deren Kosten kaum beeinflussen, wird das Performance-Modell die Kosten des optimierten Queries nicht wesentlich teurer einschätzen als die ursprügnliche. Da allerdings bei Vorliegen des Pattern ein zusätzlicher Filtermechanismus nötigt ist, der in der optimierten Version entfällt, wird dessen Kostenschätzung sicherlich dafür sorgen, dass die optimierte Version als kostengünstiger eingestuft wird.

5.2.9. Fazit

Wie in Tabelle 5.1 dargestellt, kann der Ansatz des Performance-Modells grundsätzlich die meisten kritischen Aspekte der Performance in betrieblichen Informationssystemen erkennen.

Probleme ergeben sich beim ,,Flaschenhals" und beim Anti-Pattern ,,Ramp". Der ,,Fla-

Theoretische Erkennbarkeit von Performance-Anti-Pattern		
Pattern-Name	Erkennbar	Kritischer Punkt
Ramp	Nein	Abschätzung der Datenmenge
Tower of Babel	Ja	
The God Class	Ja	
Not Optimizing For The Common Case	?	Erkennung der Optimierung
Circuitous Treasure Hunt	Ja	Schätzung Schleifen u. Queries
The One Lane Bridge	Nein	Erkennbarkeit der Wartezeiten
Empty Semi Trucks	Ja	Schätzung Schleifen u. Queries
Application Filters	Ja	

Tabelle 5.1.: Zusammenstellung der Ergebnisse der Analyse, ob ein perfektes Performance-Modell relevante Performance-Anti-Pattern erkennen kann.

schenhals" ist dabei ein allgemein recht schwer greifbares Problem, so dass das Konzept deswegen nicht abgewertet werden soll. Die fehlende Erkennung der ineffizienten Algorithmen bei großen Datenmengen hingegen stellt ein ernstzunehmendes Problem dar. Hier stellt sich die Frage, inwieweit Informationen über Datenmengen überhaupt statisch verfügbar sind und ob diese notfalls nicht vom Analysten eingefordert werden müssen.

Da allerdings ein Großteil der Performance-kritischen Aspekte erkannt wurde und die nicht Erkannten grundsätzliche Probleme der statischen Analyse betreffen, erscheint das Konzept des Performance-Modells grundsätzlich geeignet zur statischen Performance-Analyse.

6. Fallstudie: COBOL

Nachdem in der Theorie das Performance-Modell auf eine Vielzahl von Arten erstellt werden kann und aufgrund der vielen Annahmen recht abstrakt blieb, war ein wichtiger Teil dieser Arbeit die Realisierung und Evaluierung der Praxistauglichkeit des Ansatzes für die Programmiersprache COBOL. In diesem Kapitel werden daher die einzelnen Schritte, angefangen vom Parsen bis zur Analyse der Ergebnisse, beschrieben.

6.1. Auslesen des COBOL-85-Codes

Der erste Schritt zur Erstellung des Performance-Modells ist das Auslesen des Quellcodes. Da COBOL eine Sprache mit einer Vielzahl an Versionen und herstellerspezifischen Dialekten ist, muss die implementierte Lösung möglichst robust sein, um mit vielen Systemen umgehen zu können. Letztendlich wurde aufgrund der höheren Robustheit eine Kombination aus einem Lexer und einer handgeschrieben Verarbeitung gewählt.

6.1.1. Kombination aus Lexer und Grammatik

Eine naheliegende Vorgehensweise COBOL-Code zu analysieren, ist die Verwendung eines lexikalischen Scanners (Lexer) zusammen mit einem Parsergenerator. Ein Lexer liest dabei den Quellcode ein und teilt die einzelnen Wörter mit Hilfe eines vorher festgelegten Regelsatzes in Kategorien ein. Auf dieser Ebene definiert man sich nun im Parsergenerator eine Grammatik, die festlegt was gültige Kombinationen dieser Wörter sind und was in welchen Fall zu tun ist. Diese Grammatik muss vollständig sein, das heißt jedes Element des Quellcodes muss vollständig abgebildet sein. Der große Vorteil dieser Technik liegt darin, dass eine Grammatik die Regeln auf einem hohen Abstraktionsniveau beschreibt, weshalb sich diese sehr elegant und schnell aufschreiben lassen. Eine Implementierung per Hand hingegen würde auf den einzelnen Worten des Lexers arbeiten und typischerweise von der Art „IF (token(n) == a AND token(n+1) == b AND ...) DO ..." sein, was wesentlich fehleranfälliger und aufwändiger ist.

Der Parsergenerator hingegen generiert aus der Grammatik das Programm, das die Eingabe wie definiert verarbeitet. Die Erstellung der Konfiguration für den Scanner sowie der Grammatik ist eine recht komplexe Aufgabe und wäre für diese Arbeit zuviel Aufwand gewesen. Allerdings gibt es bereits fertige Scanner, zum Beispiel vom Compentence Center Software Maintenance der TU München [1] , sowie fertige COBOL-Grammatiken wie die der VU University Amsterdam [2]. Allerdings gibt es in COBOL mehrere offizielle Sprachstandards, Metasprachen sowie verschiedene Dialekte der COBOL-Produkthersteller. Da eine Grammatik per Definition vollständig sein muss, also alle verwendeten Befehle verstehen

[1] http://conqat.org/index.php/CCSM_Scanners
[2] http://www.cs.vu.nl/grammarware/vs-cobol-ii/

muss, benötigt jede dieser Versionen eine eigene Grammatik. Wie groß die Problematik ist, zeigt sich daran, dass es sogar eigene Veröffentlichungen gibt, wie man eine Grammatik für einen COBOL Dialekt erarbeiten kann, zum Beispiel von Brand [45]. Da die COBOL-Systeme, mit denen diese Arbeit evaluiert werden sollte, nicht von Anfang an feststanden, war die Verwendung einer Grammatik und somit die Festlegung auf einen Dialekt einer Sprachversion nicht möglich, weshalb dieser Ansatz nicht gewählt wurde.

6.1.2. Island Grammars

Eine Möglichkeit den offensichtlichen Widerspruch zwischen einem unbekannten COBOL-Dialekt und einer vollständigen Sprachbeschreibung in Form einer Grammatik aufzuheben, ist die Verwendung einer „Island Grammar".

> An island grammar is a grammar that consists of detailed productions describing certain constructs of interest (the islands) and liberal productions that catch the remainder (the water) [27]

Das heißt, dass eine solche Grammatik auch nur die Bereiche (Inseln) spezifizieren muss, die von Interesse sind. Je nach eingesetzter Parsertechnik, müssen auch noch die Start- und bzw. oder Endpunkte des zu ignorierenden Teils (Wasser) definiert werden.

Doch selbst mit dieser Technik wäre eine Eigenentwicklung eines COBOL-Parser zu aufwändig gewesen. Allerdings gibt es ein vielversprechendes Open-Source Projekt namens „The Koopa Cobol Parser Generator"[3], das diese Art der Grammatik unterstützt und daher genauer untersucht wurde.

Das Projekt unterstützt derzeit neben einer recht umfangreichen COBOL-Grammatik auch die Erweiterungen für SQL und CICS und verarbeitet den Quellcode standardmäßig zu einem mit der Grammatikregel annotierten, Wort-basierten Ausgabestrom. Daraus lässt sich ein Syntaxbaum erstellen, der wiederum von eigenen Parsern auf dem Baum analysiert werden kann. Hierfür enthält das Projekt bereits mehrere Lösungen, zum Beispiel eine Anbindung an das Werkzeug ANTLR[4]. ANTLR benötigt allerdings wieder eine vollständige Grammatik, so dass der Parsergenerator eigentlich nur noch einen aufwändigen Vorfilter darstellt. Da kein Übersetzungsmechanismus der KOOPA-Grammatik in die von ANTLR mitgeliefert wird, müssen beide Grammatiken separat gepflegt werden. Da bereits beim ersten Test eines COBOL-Systems dieses nicht geparst werden konnte und Anpassungen der Grammatik nötig gewesen wären, wurde auch dieser Weg nicht weiter verfolgt.

6.1.3. Lexer-basiertes Parsen mit Pattern-Matching

Da keine befriedigende Lösung zum Auslesen von COBOL-Code gefunden werden konnte, wurde ein erster Prototyp in Java erstellt, der die Ausgabe des Lexers per Hand in das Performance-Modell überführt. Eine solche manuelle Vorgehensweise wurde auch in der Literatur vorgeschlagen und verwendet, zum Beispiel in [15].

[3] http://koopa.sourceforge.net/
[4] http://www.antlr.org/

Programmstruktur		
Name	Auslesevorschrift	Beschreibung
Programm-Id	PROGRAM-ID. [STRING]	Die Programm-Id ist [STRING]
Beginn Prozeduren	PROCEDURE DIVISON.	Beginn der Befehle
Section	. SECTION [STRING].	Beginn der Section [STRING]
Paragraph	. [STRING].	Beginn des Paragraphs [STRING]
COPY	COPY [STRING]	Einfügen der Datei [STRING]

Tabelle 6.1.: Vorschriften zum Auslesen der Struktur aus der Lexerausgabe

Befehle		
Name	Auslesevorschrift	Beschreibung
CALL	CALL [STRING]	Aufruf der Programm-Id [STRING]
CALL	CALL [VARIABLE]	Aufruf der Programm-Id im Feld
EXIT	EXIT.	Ende des Programmes
MOVE	MOVE [WERT] TO [VARIABLE]	Kopieren des Wertes
EXEC SQL	EXEC SQL [QUERY] END-EXEC	SQL-Aufruf mit [QUERY]

Tabelle 6.2.: Vorschriften zum Parsen der Befehle aus der Lexerausgabe

Zum Auslesen des Quellcodes wurde der bereits oben genannte Lexer aus dem Scanner-Paket des Compentence Center Software Maintenance (CCSM) der TU München verwendet. Dieser basiert auf JFlex[5] und enthält bereits die entsprechende Konfiguration für COBOL-Code. Da die Sprachspezifikation Befehle meist mit einem eindeutigen Schlüsselwort beginnen lässt, konnten mit recht einfachen Mitteln schnell große Teile der nötigen Informationen entnommen werden. Einige Beispiele finden sich in Tabelle 6.1 und Tabelle 6.2.

Mit Hilfe der Vorschriften aus Tabelle 6.1 lässt sich die gesamte Struktur des Anweisungsteils des COBOL-Codes auslesen. Es müssen keinerlei Zustände oder vorhergehende Befehle mitgespeichert werden, denn die verwendeten Schlüsselwörter sind global eindeutig. Auch die Befehle lassen sich recht einfach auslesen. Die „CALL"-Regeln gelten analog für „PERFORM"- und „GO"- Befehle, decken also alle Aufrufe von Programmen sowie Programmteilen ab. Für eine Minimalversion des Performance-Modelles fehlen somit nur noch die Schleifen.

Leider lassen sich die Schleifen nicht so leicht aus dem Quellcode ablesen. Allerdings bietet das ConQAT-Projekt [6], das ebenfalls an der TU München entwickelt wird, eine Erweiterung für das Java-eigene Pattern-Matching, so dass dieses auch auf der Ausgabe des Scanners arbeiten kann. Somit werden die verschiedenen Varianten der Schleifen per regulärer Ausdrücke erkannt und die entsprechende Verarbeitung angestossen.

Nachdem nun der Quellcode verarbeitet werden kann, muss nun genauer definiert werden in was dieser denn übersetzt werden soll, also wie das gewünschte Performance-Modell denn eigentlich implementiert werden soll.

[5] http://jflex.de/
[6] http://conqat.org/

6.2. Das COBOL-Performance-Modell

Das Ergebnis des gerade beschriebenen Auslesens des Cobol-Quellcodes, ist das COBOL-Performance-Modell, dass in Abbildung 6.1 dargestellt ist.

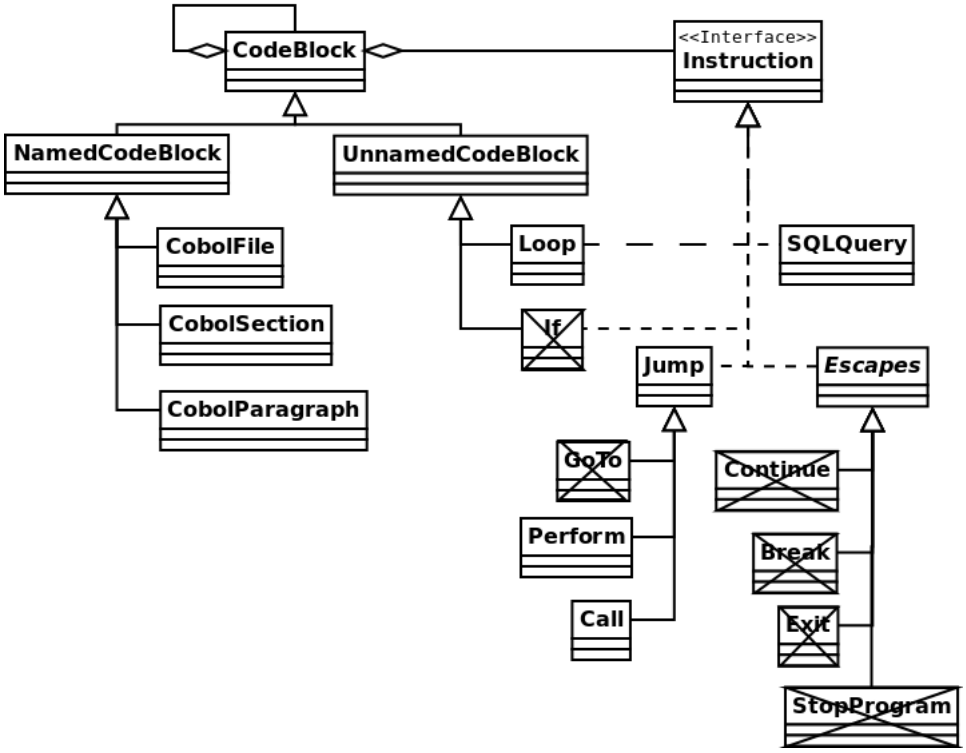


Abbildung 6.1.: Minimales Datenmodell des COBOL-Performance-Modelles

Dieses orientiert sich am Aufbau der Sprache COBOL und enthält zwei Arten von Knoten. Dies sind die Codeblöcke und Performance-kritische Anweisungen, wobei Codeblöcke wiederum andere Codeblöcke oder Anweisungen enthalten können. Aus der Definition der Sprache folgt, dass das COBOL-Performance-Modell immer aus einem Codeblock „Programm" (im Bild als CobolFile dargestellt) als Wurzel besteht, dessen Kinder „Sections" sind, deren Kinder wieder „Paragraphs" sind. Falls letztere nicht explizit im Quellcode stehen, werden diese beim Auslesen zusätzlich erzeugt, so dass diese Struktur grundsätzlich im Performance-Modell vorausgesetzt werden kann. Das Programm wird im Modell als „CobolFile" bezeichnet um auszudrücken, dass in COBOL ein Programm immer für sich abgeschlossen in einer eigenen Datei definiert ist.

Ein Paragraph hat laut Sprachdefinition nur noch Anweisungen als Kindknoten. Es gibt jedoch auch Anweisungen die namenlose Blöcke enthalten, nämlich die Kontrollstrukturen. Ein Beispiel hierfür ist der Befehl „IF ⟨Bedingung⟩ ⟨Statements⟩ END-IF" oder Schleifen wie „PERFORM UNTIL ⟨Bedingung⟩ ⟨Statements⟩ END-PERFORM". Von diesen wurde nur der Schleifenblock implementiert und die restlichen Blöcke werden ignoriert und stattdessen deren Befehle einfach zu dem übergeordneten Block hinzugefügt.

Aus der Menge der in COBOL definierten Anweisungen wurden die im Bild abgebildeten Aufrufe „Call" für Programmaufrufe sowie „Perform" für programminterne Aufrufe

aufgenommen. „GOTO" wurde zwar ursprünglich ins Modell aufgenommen und ausgelesen, allerdings stellt sich bei der Modellierung des Kontrollflusses heraus, dass diese nicht sinnvoll interpretiert werden kann. Daher wurde es nachträglich wieder aus dem Modell entfernt.

Weiterhin wurde das „SQLStatement" eingebaut, was dem COBOL-sprachfremden Zusatz für SQL (Syntax: „EXEC SQL ⟨SQL⟩ END-EXEC") entspricht.

Alle anderen Befehle werden nicht ins Performance-Modell übernommen. Bemerkenswert ist dies vorallem bei den Escape-Befehlen, also zum Beispiel „CONTINUE", „BREAK", „EXIT" und „STOP PROGRAM", die ebenfalls nach der Modellierung des Kontrollflusses als überflüssig erkannt und somit wieder aus dem Modell entfernt wurden.

Allerdings werden auch Informationen aus Befehlen gezogen, die nicht im Performance-Modell erscheinen. So sind zum Beispiel initiale Wertbelegungen für Variablen sowie Wertzuweisungen von statischen, hardcodierten Werten im akutellem Block verfügbar. Dieses Vorgehen erlaubt oftmals Informationen, die eigentlich im Quellcode stehen und vom menschlichen Betrachter mühelos abgeleitet werden können, in das Performance-Modell aufzunehmen. Dies findet man beispielsweise beim Iterieren über Strukturen deren Größe als Konstante systemweit festgelegt ist.

Eine Besonderheit von COBOL ist, dass es Schleifen ohne eigenen Schleifenkörper gibt, die stattdessen über externe Blöcke (in dem Fall über Sections und Paragraphs) iterieren. Dies wird nicht direkt ins Modell aufgenommen, sondern stattdessen in eine normale Schleife umgewandelt und mit entsprechenden Aufrufbefehlen im Schleifenrumpf modelliert. Allgemein gilt, dass die verschiedenen Schleifentypen alle in ein Schleifenobjekt umgewandelt werden, um die Analyse möglichst einfach zu halten.

Natürlich ließe sich das Modell noch um weitere Konstrukte erweitern und damit möglicherweise die Genauigkeit der Analyse erhöhen. Der hier vorgestellte Ansatz implementiert nur eine einfache Version um die Anzahl der nötigen Annahmen zu minimieren.

Was bisher im Performance-Modell noch fehlt sind die Verknüpfungen zwischen den Elementen. Diese orientieren sich an der im Quelltext definierten Ordnung. Das heißt, dass ein Befehl, der im Quelltext innerhalb eines Blockes niedergeschrieben wurde, natürlich auch im Performance-Modell diesem Block zugeordnet wird und sich somit ein Baum ergibt. Ein Aufruf eines anderen Blocks, zum Beispiel mit dem „PERFORM"-Befehl, wird als Querverbindung im Strukturbaum dargestellt, ist jedoch eigentlich nicht Teil der Struktur. Ein Beispiel für eine Instanz des vorgestellten Datenmodells findet sich in Abbildung 6.2.

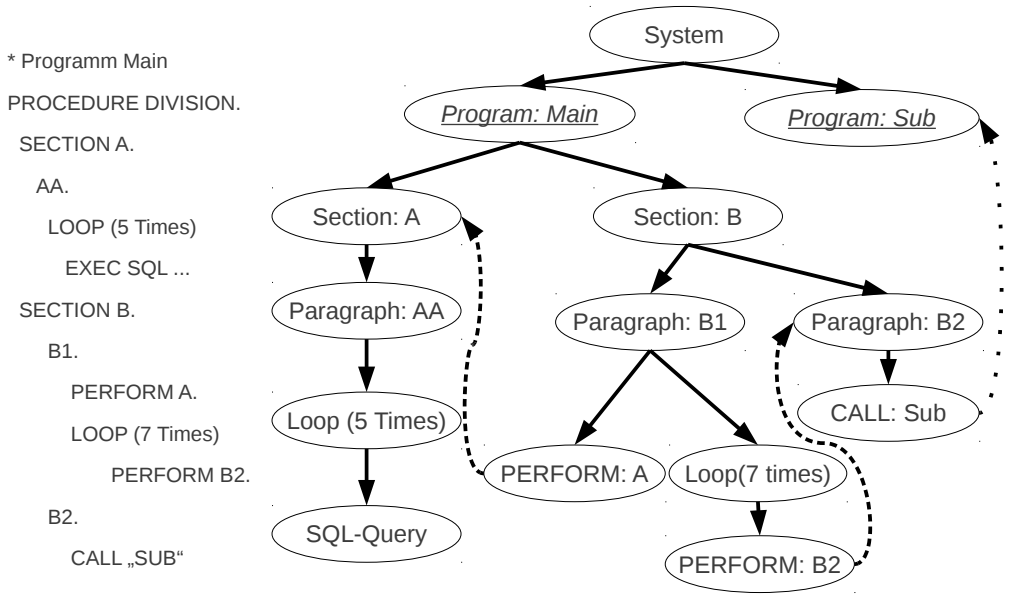


Abbildung 6.2.: COBOL-Code und die zugehörige Instanz des Performance-Modell. Ein durchgezogener Pfeil deutet eine Beinhaltet-Beziehung im Quellcode an, gestrichelte und gepunktete Pfeile eine Aufruf-Beziehung.

Der oberste Knoten, „System", ist dabei ein künstlicher Knoten um eine gemeinsame Wurzel des Baumes zu erhalten. Die Reihenfolge der Elemente im Baum wird dabei so gewählt, dass eine Tiefensuche, die immer zuerst das linke Kind wählen würde, genau dem Lesen des Quellcodes von oben nach unten entspricht. Natürlich sind nicht alle im Bild dargestellten Informationen direkt aus dem Quellcode auslesbar, weshalb ein Analyseschritt diese Informationen erarbeiten muss.

6.2.1. Befüllung des Performance-Modells

Nachdem eine Instanz des in Abbildung 6.1 gezeigten Performance-Modells erstellt wurde, gilt es nun die zur Kostenanalyse nötigen Daten zu ergänzen. Dazu sind folgende Schritte nötig:

- Bestimmung der Anzahl der Schleifendurchläufe:
  - Zählschleifen mit festen Grenzen: Hier kann der Wert direkt ausgelesen bzw. ausgerechnet werden.
  - Zählschleifen mit bekannten Grenzen: Nach der Ersetzung der bekannten Variablen erhält man denselben Fall wie bei festen Grenzen. Die Grenzen können dabei bekannt sein, weil sie einen festen Vorwert besitzen oder im aktuellen Kontext einen festen Wert zugewiesen bekommen haben.

  - Zählschleifen mit variablen Grenzen und andere Schleifen: Schleifen deren Schleifenbedingung nicht anhand der im Quelltext gegebenen Informationen abschätzbar ist, erhalten eine konfigurierbare, feste Schleifendurchzahl. [7] verwendet dafür zum Beispiel die Abschätzung von fünf Durchläufen, die er aus der Wahrscheinlichkeit, dass in einer Schleife ein Rücksprung zum Schleifenkopf erfolgt erhält (Wahrscheinlichkeit von 0.88, wobei $0.88^5 \approx 0.52$, also 5 Durchläufe). Diese Abschätzung findet sich auch in [46].

- PERFORM A THROUGH B: Die THROUGH Anweisung führt neben A und B alle Sections und Paragraphs die zwischen diesen liegen aus. Dazwischen bezieht sich in diesem Fall auf die Position im Quelltext. Da das Performance-Modell die Reihenfolge der Blöcke erhält, können die fehlenden Aufrufe recht leicht rekonstruiert werden und der THROUGH Befehl wird mit den entsprechenden einfachen Aufrufen ersetzt.

- Aufrufe: Die Sprunganweisungen, die bisher rein textuelle Sprunginformationen enthielten, werden nun mit einer Referenz auf die aufgerufenen Blöcke versehen.

Nachdem das Performance-Modell mit den Ergebnissen dieser Schritte angereichert wurde, ist es soweit vollständig, um eine Kostenanalyse auf dieser Grundlage auszuführen.

6.2.2. Kontrollfluss und Kostenanalyse

Für die Kostenanalyse wurde ein sehr einfaches und naheliegendes Verfahren implementiert, nämlich das Schätzen der Kosten anhand einer Simulation des Kontrollflusses. Dabei wird für jeden Befehl der in der Simulation ausgeführt wird, die Kostensumme um dessen Ausführungskosten erhöht. Da ja Sprünge wie „GOTO" und „CONTINUE" nicht im Performance-Modell enthalten sind, wird jeder Block immer vollständig ausgeführt und somit kann die Zählung sehr effizient geschehen, denn jeder solcher Block muss nur noch einmal geschätzt werden.

Damit also dieses Verfahren angewendet werden kann, muss zum einen der Kontrollfluss modelliert werden und zum anderen müssen die Kosten der einzelnen Befehle bekannt sein.

Kostenschätzung für COBOL-Entitäten

Die Ergebnisse des ersten Schrittes, also der Kostenbestimmung pro COBOL-Entität, finden sich in Tabelle 6.3 aufgelistet. „Zeile ignorierter Code" soll hierbei eine Abschätzung für die Kosten der Befehle sein, die nicht ins Modell aufgenommen werden. Dies liegt nahe, da COBOL eigentlich nur einen Befehl pro Zeile erlaubt und diese großteils ohne Seiteneffekte sind. Genauere Angaben der Kosten der nicht betrachteten Befehle wären auch schwer möglich gewesen ohne diese auszulesen, was unnötigen Aufwand verursacht hätte.

In der Tabelle findet sich bereits der Posten für die Datenbankabfrage, dessen Wert für die Analyse konfigurierbar bleiben wird und standardmäßig mit 500 angegeben wird. Dies basiert auf der Vorstellung, dass die Kommunikation mit einer entfernten Datenbank

Kostenaufstellung der COBOL-Befehle		
Befehl	Kosten	zusätzliche Kosten
Zeile ignorierter Code	1	
Section/Paragraph	0	
PERFORM	0	Kosten des Aufrufblocks
CALL	0	Kosten des Programmes
EXEC SQL	500	Kostenaufschlag für Tabellengrößen
Loop	0	Kosten des Blocks * Schätzung Schleifendurchlauf

Tabelle 6.3.: Kostenaufstellung der COBOL-Befehle

grundsätzlich eine teure Operation darstellt und somit der Aufruf grundsätzlich gewisse feste Kosten verursacht. Natürlich ist die Abschätzung damit noch nicht vollständig, denn was die Datenbankabfrage genau macht wurde noch nicht berücksichtigt. Diese Schätzung wird im folgenden Abschnitt nachgereicht.

Kostenschätzung für Datenbankabfragen

Bisher wurde nur die Kommunikation mit der Datenbank als grundsätzlich teuer bewertet. Diese Abschätzung alleine würde allerdings bedeuten, dass die Datenbank jeden Befehl instantan verarbeiten kann.

Diese Annahme wird allerdings für viele betriebliche Informationssysteme nicht gelten, da solche Systeme oftmals große Datenmengen verarbeiten und somit die Datenbankzugriffe entsprechend Berechnungszeit benötigen. Daher wurde, angelehnt an Selinger [36], versucht, die zusätzlichen Kosten für die Ausführung eines SQL-Queries aus der Menge der „Queries" über die Anzahl der Einträge pro Datenbanktabelle zu schätzen. Für Tabellen deren Größe nicht bekannt ist, wird ein konfigurierbarer Wert angenommen. Die Funktion „tables(q)" soll dabei alle in einer Abfrage verwendeten Tabellen zurückgeben.

$$\forall Query\ q\ \in\ Queries : Cost(q) = [\prod_{t\ \in\ tables(q)}(\lceil size(t)/faktor)\rceil] \\ /[10 * \sum_{query\_count('=')}] \\ /[5 * \sum_{query\_count('>|<|<=|>=|like')}] \\ /[(0.9) * \sum_{query\_count('<>|!=')}]$$

Durch diese Formel wird im Endeffekt eine Schätzung der Kosten per Funktion „Cost" durchgeführt, die auf der Größe der erwarteten Rückgabemenge basiert. Wie in [36] vorgeschlagen, wurde dabei davon ausgegangen, dass sich die Anzahl der nun selektierten Tabelleneinträge auf ein je nach Vergleichsoperator in dem Bedingungsteil der Abfrage reduziert. Die Anzahl dieser Operatoren soll dabei von der Methode „query_count" angegeben werden, die die Anzahl der entsprechenden Selektionen in der Abfrage zählt. Die Zahl der Einträge in den Tabellen wird jedoch nicht direkt verwendet, sondern vorher mit einem festen Faktor verringert. Dieser Faktor ist ebenfalls konfigurierbar und wurde initial mit 1000 gewählt. Dies drückt die Erwartung aus, dass Queries auf Tabellen bis zu einer bestimmten Größe keinen nennenswerten Mehraufwand bedeuten.

Natürlich lassen sich Datenbankabfragen auch ineinander schachteln. Für den Fall, dass die beiden Abfragen voneinander abhängen, spricht man von einer korrelierten Unterab-

frage. Aufgrund der Abhängigkeit der Unterabfrage, die einen Parameter der Hauptabfrage enthält, muss für jeden Wert des Parameters die Unterabfrage ausgeführt werden. Somit erhält man die Multiplikation der Kosten. Dies kann zwar zum Teil von Datenbanksystemen selbstständig optimiert werden, allgemein kann jedoch nicht davon ausgegangen werden.

$$\forall\_Abfragen\_Mit\_Korrelierter\_Unterabfrage\ q \in Queries : Cost(q) = Cost(Hauptabfrage(q)) * Cost(Unterabfrage(q))$$

Für den Fall, dass keine Abhängigkeit zwischen den beiden Abfragen besteht, werden beide auch nur einmal ausgeführt und es ergibt sich eine Addition der Kosten.

Damit sind nun die Kosten für alle Befehle bekannt und der nächste Schritt, die Bewertung von Blöcken von Befehlen, kann in Angriff genommen werden.

Kontrollfluss-Modell

Um die Kosten eines Programmes bestimmen zu können, müssen Annahmen über den Kontrollfluss gemacht werden. Die einfachste Annahme, dass jeder Befehl genau einmal ausgeführt wird, erscheint nicht zielführend. Daher wurde der Kontrollfluss der Sprache COBOL genauer untersucht und daraus ein Bewertungsverfahren abgeleitet.

Wie bereits bei der Erstellung der Anforderungen an das COBOL-Performance-Modell festgestellt wurde, trennt COBOL strikt zwischen den einzelnen Programmen. Bei einem Aufruf eines anderen Programmes ist es nicht vorgesehen, nur vorgegebene Teile dessen auszuführen, also Programmblöcke anzuspringen, sondern das Programm wird beginnend mit der ersten Anweisung ausgeführt. Da der „Exit"-Befehl, der die Ausführung eines, von einem anderen Programm aus aufgerufenen Programmes frühzeitig beendet, nicht im Modell enthalten ist, folgt daraus, dass ein Programm sowie dessen aufgerufenes (Unter-)Programm immer vollständig ausgeführt werden.

Da für die Kostenschätzung nicht relevant ist, dass beim Aufruf eines anderen Programmes der COBOL-Kontrollfluss erst diese abarbeitet und dann zurückspringt, kann der modellierte Kontrollfluss soweit vereinfacht werden, dass ein Programm immer vollständig ausgeführt wird und danach erst alle aufgerufenen Programme verarbeitet werden. Aus diesem Grund wurde entschieden, dass die Analyse in zwei Phasen durchgeführt wird. Erst wird jedes Programm für sich alleine analysiert und alle Aufrufe an andere Programme nur vermerkt aber nicht ausgeführt. Danach werden diese Aufrufe und somit das Gesamtsystem auf der Programmebene untersucht. Dadurch ist es einerseits möglich die erste Phase zu parallelisieren und andererseits können die Zwischenergebnisse nach dieser Phase verworfen werden und somit Speicher freigegeben werden.

Programminternes Kontrollfluss-Modell

Für die Modellierung des Kontrollflusses im Programm wurde angenommen, dass der Kontrollfluss eines COBOL-Programmes einen Block mit Befehlen samt Unterblöcken immer komplett abarbeitet. Das heißt, Sprünge aus dem Block wie „GOTO", oder Sprünge aus Schleifen wie „BREAK" und „CONTINUE" werden wie bereits angekündigt nicht berücksichtigt. Für die Sprünge in den Schleifen könnten diese Sprünge durch eine angepasste Durchlaufzahl der Schleife entsprechend berücksichtigt werden und somit der

Fehler begrenzt werden. Der Verzicht auf das Auslesen der GOTO-Befehle verändert hingegen definitiv das Programmverhalten.

Die Annahme, dass dies eine gültige Vereinfachung darstellt, basiert auf dem Grundgedanken, dass „GOTO" für den normalen Programmablauf vermieden [7],[44] und nur in Ausnahmefällen wie zum Beispiel zur Fehlerbehandlung verwendet werden soll. Somit würde sich auch das Problem von alleine lösen, dass die Fehlerbehandlung real selten aufgerufen wird, jedoch im Quellcode viele Sprünge in diese codiert sind und diese dementsprechend fälschlicherweise sehr teuer bewertet werden würden. Zusätzlich kann nun auf das Auslesen weiterer, komplexer COBOL-Eigenheiten wie dem „ALTER"-Befehl verzichtet werden, der das Ziel eines GOTO-Befehles ändern kann. Dieses Vorgehen hat natürlich den Nachteil, dass ein komplett auf „GOTO" basierendes Programm nicht mehr sinnvoll schätzbar ist und daher die Analyse nicht mehr auf jedem System verwendbar ist.

Somit bleiben nur noch die beiden Aufrufe PERFORM für programminterne Ziele sowie CALL für andere Programme. Da bei diesen nach der Abarbeitung des aufgerufenen Blockes oder Programmes der Kontrollfluss wieder zurückspringt und regulär den Block arbeitet, bleibt die Annahme, dass die Blöcke immer komplett abgearbeitet werden, gültig.

Die zweite Annahme die zur Erstellung des Kontrollfluss-Modelles nötig ist, ist die Angabe des Startpunktes. Die Ausführung eines COBOL-Programmes beginnt grundsätzlich mit dem ersten Befehl im Befehlsabschnitt, weshalb es naheliegt, den zugehörigen benannten Block, also den ersten Paragraph der ersten Section, als Startpunkt zu definieren. Davon wurde jedoch abgesehen und stattdessen davon ausgegangen, dass jeder Paragraph grundsätzlich einmal aufgerufen wird. Dies wäre aufgrund des regulären COBOL-Kontrollflusses der Fall, wenn das Programm keinen Sprung und keine Terminationsbefehle enthielte. Natürlich ließe sich der Kontrollfluss auch noch intelligenter modellieren um somit zum Beispiel niemals aufgerufene Blöcke aus der Kostenrechnung aus zuschließen. Dies wurde allerdings aus Gründen der Robustheit der Analyse nicht implementiert, da zum Beispiel fehlende Copy-Programme oder nicht auslesbare Sprünge möglicherweise ganze Blöcke fälschlicherweise aus der Kostenberechnung ausschließen würden. Auch CICS, eine Transaktionssteuerung von IBM, die laut einem White-Paper von Fujitsu[8] aus dem Jahre 2002 noch in COBOL/CICS-Systemen mit mehr als einer Milliarde Zeilen Code im Einsatz war, könnte durch die zusätzlichen, dynamischen Sprünge zu Problemen führen.

Als letztes muss noch festgelegt werden, in welcher Reihenfolge die Blöcke oder Befehle ausgeführt werden. Da das Performance-Modell so konstruiert wurde, dass im Quellcode das linke Kind vor dem rechtem Kind steht, wird auch der Kontrollfluss dementsprechend immer zuerst das linke Kind durchlaufen. Enthält ein Block einen Aufruf oder einen inneren Block, so wird der COBOL-Kontrollfluss sofort diesen Block abarbeiten und dann erst den nächsten Befehl des ursprünglichen Blockes. Daher gilt auch für den Kontrollfluss, dass der Durchlauf einer Tiefensuche gleicht.

Zur Veranschaulichung wird dies nochmals anhand der Abbildung 6.3 erklärt.

[7] vgl: Edsger W. Dijkstra: „go to statement considered harmful"
[8] http://www.fujitsu.com/downloads/AU/Migrating_Legacy_CICS_Applications.pdf

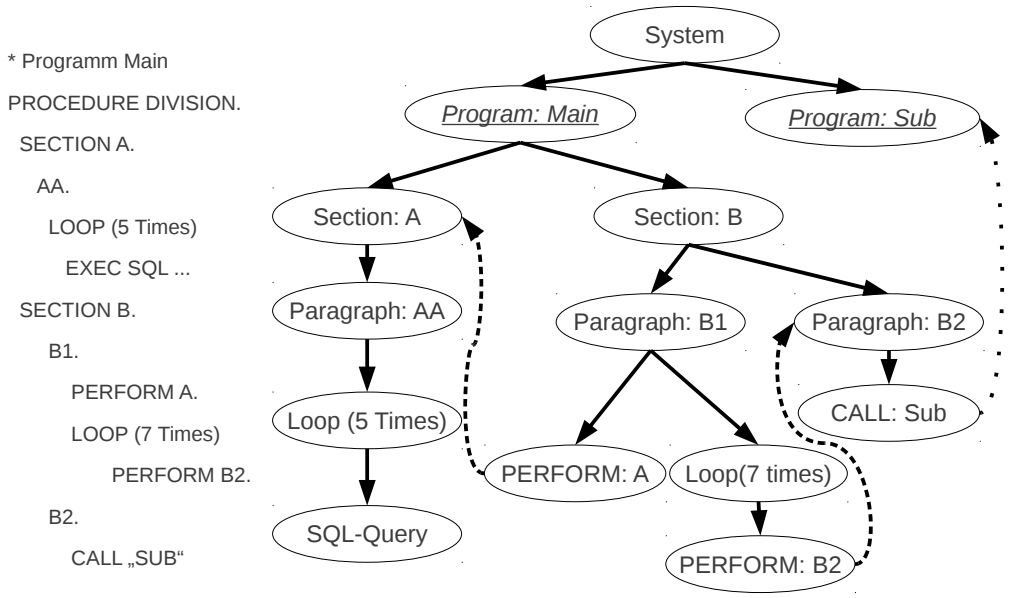


Abbildung 6.3.: COBOL-Code und das zugehörige Performance-Modell.

Für dieses Beispiel ergibt sich folgender Kontrollfluss:

- Jeder Paragraph eines Programmes wird von diesem einmal aufgerufen. Daraus folgt, dass im Performance-Modell der Kontrollfluss und somit die Tiefensuche im Knoten des Programmes beginnt.
- Ein Knoten wird ausgeführt, in dem alle ausgehenden Kanten durchlaufen werden. Auch programminterne Aufrufe mittels PERFORM-Befehl zählen dazu. Programmexterne Aufrufe mittels CALL (gepunktete Linie) werden vermerkt aber nicht ausgeführt. Die Reihenfolge des Durchlaufs gleicht dabei der einer Tiefensuche, wobei immer das linke Kind vor dem rechten ausgeführt wird.
- Im Falle einer Schleife wird der Kindblock mehrfach ausgeführt, die genaue Anzahl steht in der Schleifendurchlaufzahl der Schleife.

Vergleicht man die Reihenfolge der Abarbeitung der Knoten mit dem COBOL-Quelltext, so ergibt sich wie erwartet, dass dieser von oben nach unten durchlaufen wird.

Kostenanalyse auf Programmebene

Um die Kosten für ein Programm zu berechnen muss bekannt sein, welche Befehle wie oft ausgeführt werden. Da der Kontrollfluss ja bereits bekannt ist, reduziert sich diese

Kostenanalyse auf ein geschicktes Ablaufen aller Pfade sowie der Addition der jeweiligen Kosten. Geschickt bedeutet in diesem Fall, dass zum Beispiel die Kosten eines Blockes immer gleich sind. Daher kann beim Erreichen eines bereits berechneten Blockes dessen Kostenergebnis hinzuaddiert werden, ohne den Block noch einmal neuzuschätzen.

Somit ergibt sich direkt aus dem Aufbau von COBOL-Programmen für die Kosten eines Programmes die folgende Kostenfunktion:

- $\forall Program\ p \in Programs : Cost(p) = \sum_{Section\ s \in Sections(p)} Cost(s)$
- $\forall Section\ s \in Sections : Cost(s) = \sum_{Paragraph\ p \in Paragraphs(s)} Cost(p)$
- $\forall Paragraph\ pa \in Paragraphs : Cost(pa) = \sum_{Block\ b \in Blocks(p)} Cost(b)$
- $\forall Block\ b \in Blocks : Cost(b) = \sum_{Block\ innerblock \in Blocks(b)} Cost(innerblock)$

Im Endeffekt hätte auch die letzte Formel ausgereicht um die Kostenfunktion darzustellen, denn sowohl ein Programm, als auch eine Section und ein Paragraph sind Blöcke. Aus Anschaulichkeitsgründen wurde aber die vollständige COBOL-Struktur aufgeführt. Die Funktionen Sections(p) gibt dabei alle Sections eines Programms aus. Paragraphs(s) und Blocks(p) funktionieren analog. „Programs" enthalt dabei alle Programme aus denen das System besteht.

Die Verwendung der Summe über alle Sections und Paragraphs folgt direkt aus der Annahme, dass der Kontrollfluss eines Programmes jeden benannten Block grundsätzlich einmal ausführt.

Als letzter Schritt zur Kostenfunktion fehlt noch die Berechnung der Kosten eines Blockes. Hierfür wird folgende Berechnung verwendet:

- $\forall Block\ b : Cost(b) = Multiplikator(b) * [UnparsedLOC(b) + Cost(Befehle\ bb \in b) + (\sum_{Block\ i \in Called\_Blocks(b)} Cost(i)) + (\sum_{Block\ bi \in Blocks(b)} Cost(bi))]$
  - UnparsedLOC(Block b): Die Anzahl der Zeilen des Blockes b, die zwar COBOL-Code enthalten, jedoch komplett ignoriert wurden.
  - Called_Blocks(Block b): Die Liste aller Blöcke, die beim Ausführen dieses Blocks (ohne innere Blöcke zu betrachten) ebenfalls ausgeführt werden, also alle Sprungziele der direkt in diesem Block enthaltenen „PERFORM"-Befehle. GOTO-Befehle werden nicht verarbeitet.
  - Blocks(Block b): Die Liste aller direkt in diesem Block definierten inneren Blöcke, also aller direkten Kinder im Performance-Modell.
  - Multiplikator(Block b): Der Multiplikator gibt die Gewichtung eines Blockes an. So werden zum Beispiel Schleifenblöcke mit der Anzahl ihrer erwarteten Durchläufe multipliziert. Auch Wahrscheinlichkeitsangaben, wie zum Beispiel 0.5 für einen „IF"-Block, sind möglich.

Mit Hilfe dieser beiden Formeln und der Kostenübersicht für Befehle lassen sich nun über den Kontrollfluss, also per Tiefensuche, alle Knoten mit Kosten annotieren. Der letzte annotierte Knoten ist das Programm selbst und so kann nun dazu übergegangen werden, die Beziehungen zwischen Programmen zu betrachten.

Kostenanalyse auf Systemebene

Ein COBOL-System besteht aus einer Menge von COBOL-Programmen, dessen Ausführung durch einen externen Aufruf angestoßen wird. Jedes Programm, das über einen solchen externen Aufruf gestartet werden kann, soll Startprogramm heißen. Für jedes solche Startprogramm lassen sich nun die bei der Ausführung entstehenden Gesamtkosten (WeightedCosts) berechnen, die nicht nur die eigenen Programmkosten, sondern auch die Gesamtkosten aller aufgerufenen Programme enthalten.

- $\forall Programm \quad s \in Startprogramme : WeightedCosts(s) = Cost(s) * \sum_{Programm\ sub \in Called(s)} WeightedCosts(Programm(sub)) * Multiplikator(sub, s)$
  - Called(Programm b): Alle Programme, die von diesem Programm aus aufgerufen werden, also alle Ziele der „CALL"-Befehle.
  - Multiplikator(Programm sub, Programm haupt): Der Multiplikator gibt an, wie oft ein Programm „sub" in einem Programm „haupt" beim Durchlauf anhand des simulierten Kontrollflusses aufgerufen wird. Erfolgt zum Beispiel der Unterprogrammaufruf in einer Schleife, so berücksichtigt der Multiplikator die Anzahl der Schleifendurchläufe.

Welches Programm welche weiteren wie oft, also mit welchem Multiplikator aufruft, wird dabei bereits in der erste Phase vermerkt. Die Berechnung dafür erfolgte wiederum anhand des vorgestellten Kontrollfluss-Modelles. Bildlich gesprochen werden alle Pfade, PERFORM-Aufrufe eingeschlossen, vom Programmknoten (Program: Name) zu einem CALL-Befehl zwischengespeichert, so wie dies in Abbildung 6.4 dargestellt ist.

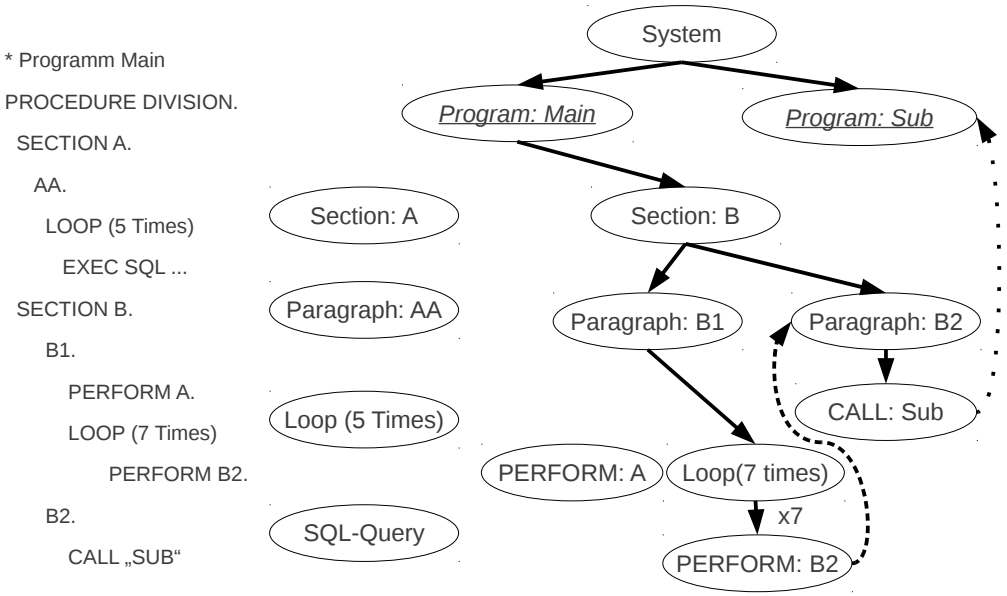


Abbildung 6.4.: COBOL-Code und das zugehörige Performance-Modell in dem nur die Aufrufpfade an externe Programme dargestellt sind.

Wie oft ein Programm aufgerufen wird, errechnet sich dabei als Summe über die Anzahl der Aufrufe auf jedem möglichen Pfad vom Programmknoten zum Aufruf. Dabei ist darauf zu achten, dass der Kontrollfluss bei Schleifen ja wieder zurückspringt und daher ein Pfad mehrere Aufrufe verursachen kann. Im Beispiel in Abbildung 6.4 ergibt sich also, dass das Programm „Main" das Programm „Sub" acht mal aufruft, der Multiplikator also acht ist, und zwar sieben mal über den Pfad über B1, ein Mal über den Pfad über B2.

Ergebnisse der Kostenfunktion

Insgesamt ergeben sich damit drei Ergebnisse der Kostenberechnungen.

- Kosten pro isoliertem Programm sowie den enthaltenen Blöcken.
- Auflistung der Programm-externen Aufrufe eines Progammes mit Angabe der Häufigkeit.
- Kosten pro Startprogramm samt Berechnungskette der aufgerufenen Programme.

Bevor diese allerdings ausgegeben werden, muss noch auf ein wichtiges Randthema eingegangen werden. Im Beispiel aus den Abbildungen 6.3 und 6.4 zeigt sich deutlich, dass die bisher verwendeten Vorschriften bei Kreisen im Kontrollfluss innerhalb des Programmes nicht funktionieren, da die Tiefensuche hier nicht mehr terminieren würde. Auch die Zyklen im Aufrufgraph der Programme untereinander können zu Problemen führen. Dies ist beides in COBOL nicht vorgesehen, da COBOL keine lokalen Variablen kennt und jedes Programm nur einmal im Speicher vorhanden sein kann, wodurch rekursive Algorithmen kaum umsetzbar sind. Jedoch kann es zur Fehlerbehandlungen durchaus sinnvoll sein solche Konstrukte zu verwenden. Aus diesem Grund muss das Vorgehen entsprechend angepasst werden.

Umgang mit Rekursion und zyklischen Aufrufketten

Die in den vorherigen Abschnitten definierten Kostenfunktionen benötigen zur Errechnung der Kosten jeweils die Kosten der inneren Blöcke sowie die Kosten aller aufgerufenen Blöcke. Das heißt zum Beispiel, dass die Kostenfunktion für einen Block der sich selber aufruft nicht terminieren kann.

Um solche Endlosschleifen zu erkennen, muss die Kostenfunktion die Liste der besuchten Knoten mitprotokollieren. Der Berechnungsalgorithmus für jeden Block wird daher so angepasst, dass die Kanten, die zu einer Rekursion führen, ignoriert werden. Stattdessen erhält der erste Knoten der Rekursion eine Kostenstrafe, die am Ende der Berechnungen zu seinen Kosten hinzugefügt wird. Auch andere Vorgehensweisen wären denkbar, wie zum Beispiel alle Knoten einer Rekursionskette teurer zu bewerten. Jede Lösung schätzt in bestimmten Fälle besser und in anderen schlechter, so dass bei der initalen Version geblieben wurde.

Da einige weitere Algorithmen der Analyse Probleme mit rekursiven Systemen haben, wurde ein Parameter „Rekursionslimit" eingeführt, der die Obergrenze der Berechnungsschritte angibt nach welcher diese abgebrochen wird. Dies betrifft den Algorithmus der

alle möglichen Pfade für bestimmte interessante Elemente wie Queries erarbeitet und die maximale Schleifentiefe festlegt. Da Rekursionen in COBOL recht selten sind, erscheint es günstiger in den wenigen Fällen dieses Limit zu erreichen als in allen anderen Fällen eine Liste der bereits besuchten Elemente mitzuschreiben. Diese Optimierung wäre so nicht für die Kostenalgorithmen möglich gewesen, da hier die Kostenrechnung massiv verfälscht werden würde.

Damit ist die Termination der Kostenrechnung gesichert, die Ergebnisse können nun ausgegeben werden und der nächste Arbeitsschritt, die Performance-Anti-Pattern, kann in Angriff genommen werden.

6.3. Performance-Anti-Pattern in COBOL

Die zweite große Fragestellung dieser Arbeit war die Verwendbarkeit von Performance-Anti-Pattern zur Optimierung. Dazu wurde die Erkennung von vier solchen Mustern implementiert, was in diesem Abschnitt beschrieben wird.

6.3.1. Erkennung von Performance-Anti-Pattern

Da die statische Analyse möglichst performant arbeiten soll, muss die Erkennung der Anti-Pattern möglichst effizient ablaufen. Dazu wurde ein dreistufiges Verfahren, angelehnt an [23], verwendet.

Der erste Schritt basiert auf Metriken, die während der Erstellung und der Kostenschätzung des Performance-Modells mitgespeichert werden. Nur wenn ein Anti-Pattern aufgrund dieser Daten überhaupt vorkommen kann, wird die nächste Phase eingeleitet. Beispiel hierfür sind Sortieralgorithmen. Diese benötigen im Allgemeinen mindestens zwei ineinander geschachtelte Schleifen zur Ausführung. Da bei der Kostenrechnung alle Pfade durch das Modell bereits durchlaufen wurden, ist die maximale Schleifentiefe eines Programmes bekannt. Liegt diese bei unter zwei muss dieses Programm nicht nach ineffizienten Sortierverfahren untersucht werden.

Der nächste Schritt betrachtet nun das Performance-Modell selbst. Nur wenn hier die nötigen Charakteristika vorliegen, wird in die nächste Phase gewechselt. Ein Beispiel hierfür wäre das Anti-Pattern der Schatzsuche. Findet sich kein Pfad von einer Schleife zu einem SQL-Query so kann die Suche abbrechen.

Zu guter Letzt kann die Anti-Pattern-Suche auf den Ergebnissen des Lexers, also auf den einzelnen Wörtern im Quelltext, suchen, um nicht im Performance-Modell enthaltene Information zur Verifikation des Vorliegen einer Anti-Pattern-Instanz zu nutzen. Dies geschieht wiederum mit Hilfe der schon zur Erstellung des Performance-Modells genutzten regulären Ausdrücke. Dieser Schritt ist nötig, denn das Performance-Modell kann oft „nur" feststellen, dass ein Algorithmus teuer ist. Zur Namensklassifizierung muss aber, neben den Informationen über die Laufzeit, oftmals auch eine gewisse Grobstruktur des Algorithmuses nachgewiesen werden, die nicht notwendigerweise im Performance-Modell enthalten ist. Ein Beispiel hierfür wäre, dass die Information benötigt wird, ob der Laufindex der Schleife auch in der Fallunterscheidung verwendet wird.

Natürlich müssen nicht immer alle Schritte zur Pattern-Erkennung durchgeführt werden, vor allem der letzte Schritt sollte in einem Performance-Modell, das alle relevanten Informationen enthält, oftmals entfallen.

6.3.2. Implementierte Anti-Pattern

Aufgrund der Erfahrungsberichte der itestra-Mitarbeiter wurden eine Reihe von Anti-Pattern abgeleitet, die für diese Arbeit implementiert werden sollen. Dies waren die Erkennung von korrelierten Unterabfragen, Bubble Sort, Schatzsuchen sowie die Einzeldatenverarbeitung. Bubble Sort stellt natürlich an sich noch kein Anti-Pattern dar, wird dieser Algorithmus allerdings zum Sortieren von größeren Datenmengen verwendet, handelt es sich um eine Instanz des „Ramp"-Anti-Patterns und soll daher auch in der Analyse erkannt werden.

- Korrelierte Unterabfragen (vgl. 3.2.1): Die Erkennung von korrelierten Unterabfragen nutzt ausschließlich das Performance-Modell als Suchraum. Hierzu werden alle darin enthaltenen SQL-Befehl per regulärer Ausdrücke zuerst nach solchen mit Unterabfragen durchsucht. Damit das Anti-Pattern vorliegt, muss die Unterabfrage eine Referenz einer Tabelle aufweisen, die so in der eigentlichen Hauptabfrage enthalten ist.

- Bubble Sort (vgl. 3.2.1): Ein Bubble Sort kann nur dann vorliegen, wenn die Schleifentiefe eines Programmes größer als eins ist. Ist dies der Fall, werden alle Kandidaten für das Anti-Pattern, also alle geschachtelten Schleifen im Performance-Modell vermerkt. Diese werden nun in einem dritten Schritt nach dem charakteristischen Muster: „IF ...MOVE ...MOVE ...MOVE" durchsucht, wobei die MOVE-Befehle den Inhalt zweier Variablen tauschen müssen. Liegt dies in der richtigen Reihenfolge und zwar ohne bestimmte Befehle wie „CONTINUE" vor, so wird der Pattern-Fund vermerkt.

- Schatzsuche (vgl. 3.3.1): Eine Schatzsuche liegt dann vor, wenn das eigentlich gesuchte Datum nicht per Datenbankabfrage direkt abgefragt wird, sondern in mehreren Schleifendurchläufen im eigentlichen Programmcode erarbeitet wird, welche jeweils selbst einen Datenbankzugriff beinhalten. Daher wird für die Erkennung zuerst nach Schleifen mit Datenbankzugriffen im Quellcode gesucht. Enthalten diese Queries nun eine Bedingung, die aus einer Variable ausgelesen wird und überschreiben diese mit dem Rückgabewert der Datenbank, so liegt dieses Anti-Pattern vor. Enthält die Abfrage hingegen eine Bedingung, die aus einer Variablen gelesen wird, und einen Rückgabewert, der in eine andere Variable geschrieben wird, so könnte es sich ebenfalls um eine Pattern-Instanz handeln. In diesem Fall wird ein Hinweis ausgegeben, denn der Variable der Bedingung könnte ja im Code jeweils der Rückgabewert zugewiesen werden.

- Einzelverarbeitung (vgl. 3.3.3): Dieser Begriff beschreibt ein Programm, das jeden Datensatz einzeln aus der Datenbank holt, verarbeitet und zurückschreibt. Die Problematik liegt dabei darin, dass viele Datenbankabfragen die kaum Daten enthalten

verwendet werden. Viel effizienter wäre es, eine größere Menge von Datensätzen auf einmal zu holen, diese dann einzeln zu verarbeiten und gemeinsam wieder zurückzuschreiben.

Wie bereits festgestellt, sind diese Performance-Anti-Pattern jedoch nur dann relevant, wenn sie auch in Performance-technisch bedeutsamen Programmen enthalten sind. Daher wird die Liste der Funde nach den Kosten der Programme, in denen sie enthalten sind, sortiert. Dies soll sicherstellen, dass die Optimierung der genannten Anti-Pattern auch wirklich nur in den für die Performance kritischen Systemteilen vorgenommen wird.

Hiermit ist nun die Beschreibung der Implementierung des Analysewerkzeuges beendet und es folgt die Evaluation des Werkzeuges.

6.4. Beschreibung der Ausgangsbasis der Evaluation

Zur Evaluation des Analysewerkzeuges benötigt man ein System das analysiert werden kann, sowie Zusatzwissen um die Analyseergebnisse einschätzen zu können. Das COBOL-System zur Evaluierung wurde der itestra GmbH für diese Arbeit zur Verfügung gestellt, unter der Bedingung dass weder der Name des Inhabers, der Name des Systems, die Namen der Programme noch der Quellcode veröffentlicht werden. Auch absolute Zahlen betreffend der Systemperformance und der Größen der Datenbanktabellen sind geheim zu halten.

6.4.1. Untersuchtes Informationssystem

Das untersuchte System ist ein COBOL-Informationssystem, das in einer Mainframe-Umgebung läuft und das seit vielen Jahren im Einsatz ist. Um die großen Mengen an Daten zu verarbeiten, werden diese in eine relationale Datenbank geschrieben, die per SQL gesteuert wird, wobei die Anzahl der Einträge der Datenbanktabellen vorliegt.

Der Aufruf des Systems selbst erfolgt über eine IBM Transaktionssteuerung namens CICS. Da CICS neben Aufrufen von außen auch dynamische Sprünge im Quellcode erlaubt, ist die Erweiterung erst einmal als kritisch zu sehen. Insgesamt stehen 400.000 LOC, also Zeilen Quellcode, zur Verfügung. Darin enthalten ist der Großteil der Funktionalität, jedoch fehlen kleinere Teile wie zum Beispiel Copybooks.

6.4.2. Optimierungshistorie und Messwerte

Über dieses System ist weiterhin bekannt, dass eine Optimierung stattgefunden hat. Zum Vergleich liegen daher die optimierten Dateien auch in ihrer ursprünglichen Form vor. Außerdem weiß man, dass die Optimierung auf Basis der Aussage erfolgte, dass die CICS-Transaktion A und B die teuersten seien. Den Optimierern lagen für das System keine Kostenmessungen vor. Welche Maßnahmen sie zur Optimierung durchgeführt haben, findet sich in einer ebenfalls verfügbaren Beschreibung.

Nach der Optimierung wurde die System-Performance anhand zweier dynamischer

Standardkonfiguration der Analyse-Parameter	
Parameter	Wert
Kosten eines Queries	500
Durchlaufzahl unparsbarer Schleifen	5
Rekursion innerhalb eines Programmes	5
Rekursion bei Programmaufrufen	5
Rekursionslimit	5000
Tabellengrößen unbekannter Tabellen	1
Faktor durch den Tabellengrößen geteilt werden	1000

Tabelle 6.4.: Standardkonfiguration der Analyse

Messungen analysiert, welche ebenfalls zur Verfügung gestellt wurden. Diese Messungen enthalten neben den Informationen über die Kosten auch die Aufrufzahlen für jedes Programm. Das heißt, dass die Programmaufrufe per CICS direkt in diesen Aufrufzahlen und nicht in den Kosten erscheinen. Es ist somit nicht nötig, die dynamischen CICS-Sprünge mit in die Analyse aufzunehmen, da diese bereits über die Aufrufzahlen abgegolten sind.

Was zählt ist die Information, dass das COBOL-System acht CICS-Transaktionen enthält, sowie 25 weitere mögliche Einstiegspunkte existieren. Im Endeffekt handelt es sich also um 33 eng verwobene COBOL-Systeme, wobei die jeweilige Aufrufhäufigkeit pro Subsystem bekannt ist.

6.4.3. Konfiguration der Analyse

Wie bereits bei der Beschreibung des Analysewerkzeuges dargestellt, gibt es einige Annahmen, die anhand eines Parameters in die Kostenschätzung einfließen. Für die folgenden Ergebnisse sah diese Konfiguration wie in Tabelle 6.4 dargestellt aus.

Außerdem musste das Werkzeug mit Zusatzwissen über den Präcompiler ausgestattet werden, um das System analysieren zu können. In diesem Fall gab es eine Reihe von Sprüngen zu Blöcken, die nicht im Modell auftauchten. Alle diese Aufrufe hatten die Form: „PERFORM CALL-PROGRAMMNAME". Auch aufgrund der Kommentare dieser Aufrufe wurde klar, dass dies in „CALL PROGRAMMNAME" übersetzt werden musste, falls dieser Block nicht im Programm vorhanden war.

Auf Basis dieser Einstellungen wurde das Analysewerkzeug nun auf den Quellcode des Systems angewendet.

6.5. Evaluation der Praxistauglichkeit der Analyseergebnisse der Fallstudie

Nachdem das Analysewerkzeug nun lauffähig ist und der zu analysierende Code bereit steht, kann final der Frage nachgegangen werden, ob dieses denn in der Praxis brauchbar ist, um den Optimierungsprozess zu beschleunigen.

Zur Beantwortung diese Frage wurde, unter anderem aus dem vorhandenen Vorwissen über das System, eine Liste mit Teilfragen erarbeitet, die die Praxistauglichkeit des Ansatzes prüfen sollen.

1. Ist die Analyse in vernünftiger Zeit durchführbar?
2. Ist die Kostenschätzung der Datenbankzugriffe auf Basis der Tabellengrößen brauchbar?
3. Wie wirkt sich die Wahl der Parameter auf die Analyse aus?
4. Sind die Optimierungsschritte nachvollziehbar?
5. Zeigen die Schätzungen die Einsparungen der Optimierungen?
6. Korreliert die statische Schätzung mit den Messdaten der dynamischen Analyse?
7. Bringt die Performance-Anti-Pattern-Suche brauchbare Ergebnisse?

Durch die Beantwortung dieser Fragen soll die Praxistauglichkeit des Werkzeugs beleuchtet werden.

6.5.1. Frage 1: Ist die Analyse in vernünftiger Zeit durchführbar?

Damit eine Analyse in der Praxis eingesetzt werden kann, muss sie innerhalb eines vernünftigen Zeitrahmens fertig werden. Weiterhin stellt sich die Frage, wie sich diese Laufzeit verändert, wenn die zu verarbeitende Menge an Quellcode wächst und wo gegebenenfalls die zu erwartende Obergrenze der maximal verarbeitbaren Eingabe liegt.

Dazu wurde die Laufzeit auf einem AMD 64x2 Dual Core mit 3 Ghz sowie 3 GB Arbeitsspeicher gemessen. Die Zeitbestimmung erfolgte anhand der Java-internen Funktion: „System.currentTimeMillies()". Festgehalten werden die Zeitpunkte vor und nach der Analyse aller Programme sowie nach der Analyse des Gesamtsystems, also genau die zwei beschriebenen Phasen der Analyse. Als Eingabe dient dabei das Zielsystem oder ausgewählte Teile dessen. Andere Zielsysteme standen leider nicht zur Verfügung, so dass vor allem die Werte für Phase II, also die Kostenberechnung für das Gesamtsystem, nur als grobe Richtwerte anzusehen sind. Weiterhin wurde die Laufzeit von einzelnen Programmen gemessen. Dazu wurde das längste Programm mit Programmen verglichen, die je ein halb bzw. ein viertel Mal soviele LOC haben.

Zur Größenmessung der Eingabe wurde die Metrik LOC verwendet, was die Anzahl der nichtleeren, nichtkommentierten Zeilen im Quellcode angibt. Um Messfehler zu verringern wurden die Messungen mehrfach durchgeführt und der Mittelwert verwendet. Weiterhin wurden alle Ergebnisse ausgegeben sowie einige Zwischenergebnisse mitgeschrieben (ca. fünf Megabyte an Daten), um realitätsnahe Analysebedingungen zu simulieren.

Dabei wurden die in Tabelle 6.5 und 6.6 dargestellten Messergebnisse festgestellt.

Laufzeitmessung auf Systemebene in Sekunden					
.	Anz. Programme	ca. LOC	Phase I	Phase II	Gesamt
halbes System	202	200.000	37,8	0,2	38,0
dreiviertel System	303.000	300	48,6	0,2	48,8
ganzes System	404	400.000	68,4	0,3	68,7

Tabelle 6.5.: Laufzeitmessung auf Systemebene

Laufzeitmessung auf Programmebene in Sekunden			
LOC	Laufzeit	Wachstum der Laufzeit	Kosten pro kLOC
3400	0,4		0,13
7400	1,3	2,94	0,17
16300	3,5	2,80	0,22

Tabelle 6.6.: Laufzeitmessung auf Programmebene. kLOC steht dabei für 1000 LOC.

Aufgrund dieser Ergebnisse zeigt sich, dass die Laufzeit mit 43 Sekunden für ein 400.000 LOC großes System durchaus im Bereich des Vertretbaren liegen. Weiterhin darf aufgrund von Tabelle 6.5 angenommen werden, dass die Laufzeit für größere Gesamtsysteme ungefähr linear wächst. Dies folgt einerseits aus den Tests mit dem halben System in verschiedenen Partitionierungen sowie aus der Verteilung der Zeit zwischen Phase I und Phase II. Da Phase II, in der das System analysiert wird, nur minimalen Einfluss auf die Laufzeit hat, ist davon auszugehen, dass Phase I die Laufzeit bestimmt, welche wie gezeigt linear mit der Zahl der Programme anwächst.

Tabelle 6.6 zeigt die Kosten pro 1000 Zeilen Code. Obwohl nur drei zufällige, besonders lange Programme des Systems analysiert wurden, legen die Daten nahe, dass die Laufzeit mit der Länge der Programme nicht mehr linear, sondern exponentiell wächst. Da mit der Länge des Quellcodes auch zu erwarten ist, dass sich die Anzahl und Länge der zu analysierenden Pfade steigert und somit der Analyseaufwand wächst, verwundert dies nicht.

Insgesamt ergibt sich, dass die Analyse gut skaliert, zumal offensichtliche Optimierungsmöglichkeiten wie Phase I zu parallelisieren noch nicht ausgeschöpft wurden.

6.5.2. Frage 2: Ist die Kostenschätzung der Datenbankzugriffe auf Basis der Tabellengrößen brauchbar?

Eine der gewagtesten Abschätzungen des Performance-Modells war die Kostenschätzung der Datenbankabfragen anhand der Tabellengrößen.

Für die Berechnung der Kosten wurde dazu die in Abschnitt 6.2.2 definierte Schätzung verwendet, wobei der Faktor, durch den die Anzahl der Datensätze geteilt wird, einmal mit 1.000 und einmal mit 100.000 gewählt wurde. Auf Basis der Einstiegspunkte der aus den beiden dynamischen Messungen bekannten Programme, wurde die Schätzung mit und ohne Kostenschätzung für die Queries verglichen.

Ergebnisse für Faktor 1.000: Während die meisten Programme (21/30) ein Wachstum

von unter 5 Prozent verbuchten, explodierten vier davon mit einem Wachstum von über einer Millionen Prozent, wodurch sich diese Programme nun deutlich an die Spitze der Kostenreihenfolge stellten.

Ergebnisse für Faktor 100.000: Die Auswirkungen auf die Kostenschätzung sind in den meisten Fällen im Promillebereich, nur bei den vier vorher erwähnten Programmen wurde noch ein Wachstum von über 10.000% erreicht.

Interpretation: Aufgrund der stark unterschiedlichen Größen der Datenbanktabellen, die zwischen einer und über 100 Millionen Einträgen liegt, wirken sich die meisten Abfragen kaum aus. Einige große Queries zerstören dagegen die Kostenschätzung, da die Kosten der Abfrage alleine bereits teurer als das teuerste Programm ist. Zudem widerspricht die dadurch entstehende Reihenfolge der Topverbraucher sowohl den dynamischen Messungen als auch den Erkenntnissen aus der Optimierung.

Die Problematik zeigt sich auch an den Ergebnissen für die verschiedenen Faktoren. Dreht man den Faktor nach oben verringert sich der sowieso kleine Einfluss der meisten Queries ins Unkenntliche, während die großen Queries langsam vernünftige Werte annehmen. Da nur ein System zur Analyse verfügbar ist, für das allerdings keinerlei Angaben über die Kosten der Queries existieren, kann auf dieser Datenbasis keine sinnvolle Abschätzung gefunden werden. Der vorgestellte Weg hat sich auf jeden Fall als nicht brauchbar erwiesen.

Da viele derzeit verwendete Datenbanken jedoch eigene Analysewerkzeuge zur Kostenbestimmung von Queries sowie detailierte Statistiken anbieten, scheint dies ein möglicher Ausweg zu sein. Natürlich müsste zur Durchführung einer solchen Analyse dann zuerst eine entsprechende Anbindung an die verwendete Version der Datenbank implementiert werden.

Für die folgenden Messungen wurde die erweiterte Kostenschätzung für Datenbankzugriffe daher ausgeschaltet und jede Datenbankinteraktion mit 500 Kostenpunkten bewertet.

6.5.3. Frage 3: Wie wirkt sich die Wahl der Parameter auf die Analyse aus?

Zur Befüllung des Performance-Modells sowie zur Kostenberechnung müssen, wie beschrieben, Annahmen in Form eines Wertes eines Parameters getroffen werden und zwar für Rekursionen, unbekannten Schleifen sowie die Kosten eines Queries. Da die Wahl dieser Parameter nicht auf nachprüfbaren Berechnungen basiert, soll im folgenden Abschnitt untersucht werden, inwieweit die Wahl der Parameter die Ergebnisse beeinflusst.

Zur Beantwortung dieser Frage müsste eigentlich eine Sensitivitätsanalyse durchgeführt werden. Da allerdings die Parameter nur additiv bzw. multiplikativ in die Schätzung eingehen und die Kosten mit den Parametern monoton anwachsen, wurde aufgrund des hohen Aufwandes von diesem Vorgehen abgesehen.

Die Methodik zur Beantwortung dieser Frage basiert stattdessen auf den statisch errechneten Kostenrangfolgen für die bekannten Einstiegspunkte. Werden durch die Variation des Parameters die Rangfolgen der Programme nur geringfügig verändert, so gilt die Analyse als stabil und der Parameter hat keinen großen Einfluss. Zu erwarten wäre, dass der Einfluss dieser Parameter recht gering ist, da ansonsten die Aussagekraft der Analyse stark verringert werden würde, da sie dann anhand der Parameter steuerbar wäre.
Im Folgenden werden die Parameter untersucht, wobei alle gerade nicht untersuchten Pa-

rameter jeweils auf ihren Standartwert gelassen werden, welcher in Tabelle 6.4 dargestellt ist.

Parameter: Rekursion

Da in COBOL Rekursionen eigentlich nicht vorgesehen sind, sollte der Parameter der angibt, wie oft eine gefundene Rekursionskette innerhalb oder außerhalb eines Programmes durchlaufen wird und somit deren Kosten um den zugeteilten Faktor erhöht, keinerlei Auswirkung auf das Analyseergebnis haben.

Dazu wurde die Messung einmal mit der Wertbelegung 1 und einmal mit 500 durchgeführt und die Ergebnisse mit denen der Referenzmessung verglichen.

Das Ergebnis dieser Untersuchung zeigte, dass nur ein Programm von 25 davon betroffen war. Für den Wert 1 zeigte sich eine Verringerung der Kosten um 9%, was keinerlei Auswirkungen auf die Rangfolge hat. Für den Parameter 500 liegt das Wachstum des einen Programmes bei einem Faktor 13, welches zur Folge hat, dass das betroffene Programm mit einem anderen die Rangnummer tauscht, wobei beide nach wie vor im Mittelfeld rangieren.

Anhand des geringen Einflusses dieses Parameters zeigt sich, dass die Festlegung dieses Parameters mit dem Wert fünf als unproblematisch eingestuft werden kann und im Folgenden nicht weiter beachtet werden muss.

Parameter: Querykosten und Schleifendurchlaufzahl

Der Parameter der Querykosten weist jedem Query unabhängig vom Inhalt einen Kostenwert zu. Da die Kostenschätzung einer Datenbankabfrage anhand der Tabellengrößen wie bereits dargelegt keine brauchbare Lösung darstellt, kostet jedes Query mit Ausnahme der korrelierten Unterabfragen 500 Punkte. Da auch dieser Parameter frei gewählt wurde, soll nun untersucht werden ob dessen Veränderung spürbare Auswirkungen auf die Aussagekraft der Analyse hat.

Der Parameter der Schleifenzahl legt die Anzahl der Schleifendurchläufe für den Fall fest, dass dieser Wert nicht direkt aus dem Quellcode folgt. Diese Schätzung betrifft ca. 40% aller Schleifen im Programm. Knapp 60% sind auslesbar und haben eine durchschnittliche Durchlaufzahl etwa zehn Durchläufen.

Als Methodik zur Bestimmung der Auswirkung der Werte des Parameters wird der Vergleich der Kosten gewählt. Zuerst werden beide Parameter unabhängig voneinander variiert und anschließend beide miteinander. Dabei sollten möglichst wenige Schnittpunkte im Graphen der Kostenrangfolgen der Programme auftauchen, da an diesen Stellen die Reihenfolge verändert wird.

Die Abbildungen 6.5, 6.6 und 6.7 zeigen die Ergebnisse für die Kostenschätzung der acht CICS Einstiegspunkte. Da sich für die restlichen Einstiegspunkte ein ähnliches Bild abgezeichnet hat, wurden die Graphen nicht in die Arbeit mitaufgenommen. Bemerkenswert ist jedoch, dass das teuerste Programm, die Transaktion A, egal unter welcher Parameterwahl die unangefochtene Nummer eins bleibt.

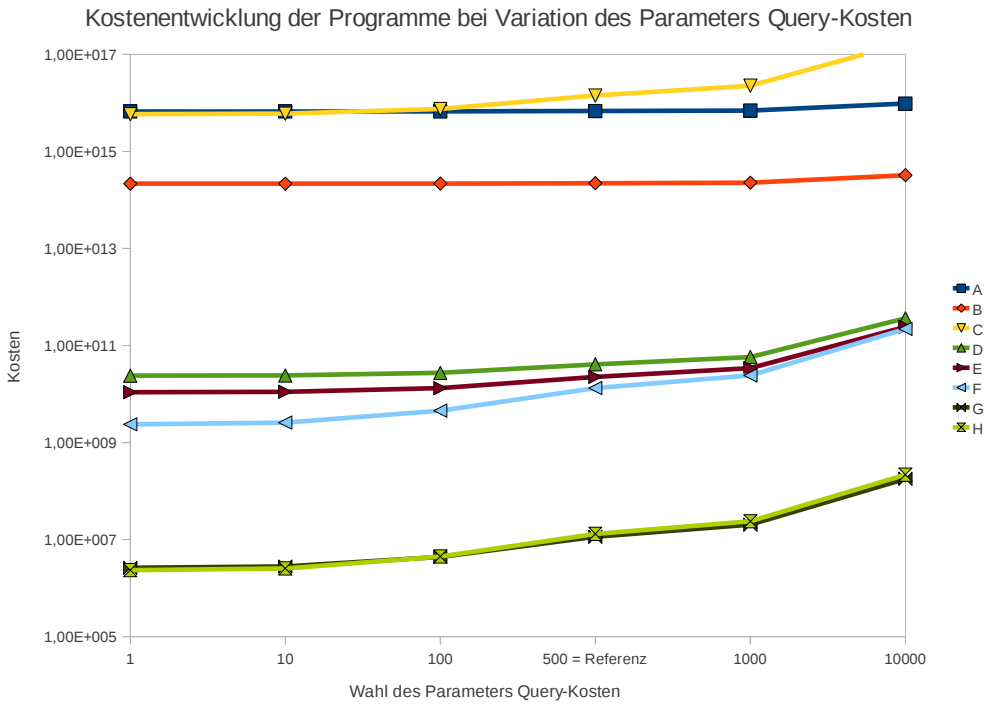


Abbildung 6.5.: Kostentwicklung der CICS Einstiegspunkte bei Variation des Parameters Query-Kosten

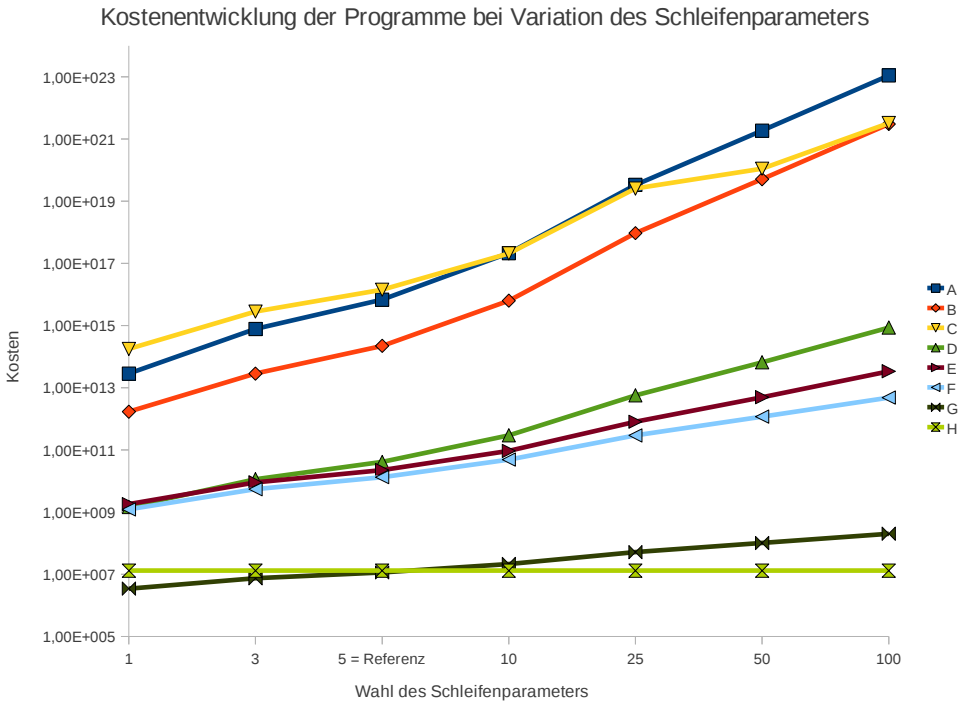


Abbildung 6.6.: Kostentwicklung der CICS Einstiegspunkte bei Variation des Parameters Schleifendurchlaufzahl

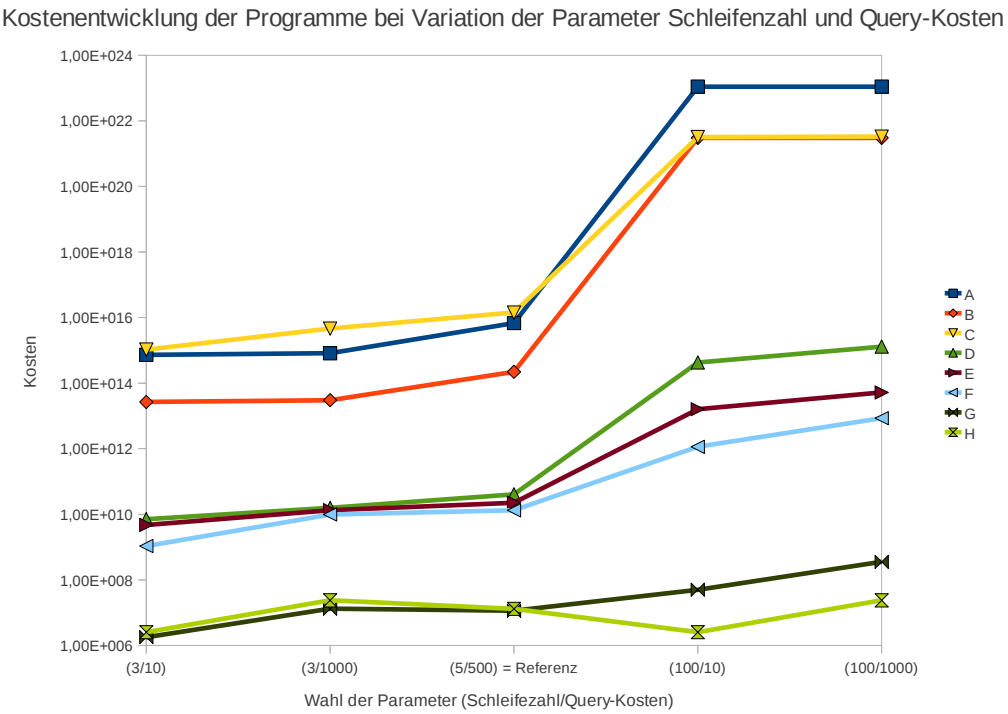


Abbildung 6.7.: Kostentwicklung der CICS Einstiegspunkte bei gleichzeitiger Variation der Parameter Schleifendurchlaufzahl und Query-Kosten

Die Graphiken zeigen, dass der Einfluss der Parameter nur gering ist. Zwar tauschen immer wieder Programme ihren Rang, allerdings kommt es niemals zu massiven Verschiebungen und die teuersten bzw. billigsten Programme bleiben unter sich.

Insgesamt ergibt sich somit, dass die Parameterwahl zwar geringen Einfluss auf die Reihenfolge der Programme hat, dies aber zu gering ist um die Aussage der Analyse zu beeinträchtigen.

6.5.4. Frage 4: Sind die Optimierungsschritte nachvollziehbar?

Da der Codestand vor der Optimierung und die Optimierungshistorie bekannt ist, stellt sich die Frage, ob die Optimierungen anhand der Analysen nachvollziehbar sind. Dazu wird untersucht, wie viele Programme man analysieren muss, bis man die später tatsächlich Optimierten findet, wenn man als Reihenfolge die Kostenschätzung der statischen Analyse verwendet. Da das Projekt erfolgreich war, zur Optimierung nur drei Dateien verändert wurden und nicht mehr Informationen zur Verfügung stehen, wird im Folgenden davon ausgegangen, dass der optimierte Systemteil jeweils einer der teuersten seiner Art ist, was natürlich nicht belegt werden kann.

Für die Einstiegspunkte in das System bestimmt sich die Kostenrangfolge anhand der Kostenschätzung multipliziert mit Anzahl der Aufrufe, die aus der dynamischen Messung abgeleitet werden kann.

Nachdem bekannt ist welcher Einstiegspunkt und somit welche Aufrufkette optimiert werden soll, bieten sich zwei Vorgehensweisen an. Zum einen kann man versuchen die

Kostenschätzung der Einstiegspunkte				
Name	Kosten pro Aufruf	Anzahl Aufrufe	Gesamtkosten	Rangfolge
A (opt)	8,15E+09	836.006	6,81E+15	2
B (opt)	9,82E+09	22.583	2,22E+14	3
C	2,58E+10	552.236	1,42E+16	1
D	2,37E+08	172	4,08E+10	4
E	5,35E+08	42	2,25E+10	5
F	2,20E+06	5.764	1,33E+10	6
G	101.032	113	1,14E+07	8
H	25.537	512	1,30E+07	7

Tabelle 6.7.: Kostenschätzung der Einstiegspunkte

Aufrufkette zu verändern, also zum Beispiel eine Optimierung dadurch erreichen, dass das Hauptprogramm ein Unterprogramm nur noch einmal statt zehnmal aufruft. Hierzu muss dem Optimierer bekannt sein, welche Folgekosten ein Programmaufruf hat und wie oft das Programm als Folge von einem Aufruf des Einstiegspunktes aufgerufen wird. Da der Einstiegspunkt alle anderen Programmaufrufe verursacht, wird er in dieser Metrik immer der erste Optimierungskandidat sein, gefolgt von den Programmen, die er direkt aufruft.

Zum anderen kann man versuchen, das Programm intern zu optimieren, unabhängig von allen anderen Programmen und der Aufrufkette. Hierzu bestimmt sich die Rangfolge der zu optimierenden Programme anhand ihrer Kosten ohne Unterprogrammaufrufe multipliziert mit der Anzahl der Aufrufe. Diese Metrik lässt sich sowohl für einen einzelnen Einstiegspunkt bestimmen als auch für alle Einstiegspunkte, da ein Programm ja auch in mehreren Einstiegspunkten eine entscheidende Rolle spielen kann und sich die Optimierung somit global auswirkt.

Mit Hilfe dieser Metriken wird nun versucht, die Optimierungshistorie unter der Annahme nachzuvollziehen, dass der Optimierer immer mit dem laut Kostenschätzung teuersten Programm anfängt.

Frage 4a: Werden die zwei teuersten Transaktionen gefunden

Die Optimierer hatten im Vorhinein nur die Information, dass die Transaktionen A und B die teuersten seien. Hätte dies mit der statischen Analyse auch festgestellt werden können?

Zur Kostenschätzung wird das gerade beschriebene Vorgehen angewendet, so dass sich die Gesamtkosten pro Einstiegspunkt aus den Programmkosten multipliziert mit der Anzahl der Aufrufe errechnen.

Das Ergebnis der Schätzung findet sich in 6.7. Dabei folgte die Namensgebung und Anordnung der Transaktionen in der Tabelle dem Ergebnis der dynamischen Messung nach der Optimierung.

Dieses Resultat zeigt, dass die gesuchten Transaktionen zwar nicht Platz 1 und 2 innehaben, jedoch mit 2 und 3 immer noch gut auffindbar sind. Es stellt sich natürlich trotzdem die Frage, wieso Transaktion C offensichtlich so überschätzt wurde. Die wahrscheinlichs-

te Antwort findet sich in der Programmbeschreibung. Während Programm A und B eine Verarbeitung neuer Informationen in die Datenbasis vornehmen, erstellt Programm C eine Art Abschlussbericht. Daher greift Programm C potentiell auf viel größere Systemteile zu. Außerdem müssen, je nach Eingabe, für das jeweilige Element oft noch Zusatzinformationen aus der Datenbank geholt werden. Die Abhängigkeit von der Eingabe zeigt sich in einer doppelt so hohen Zahl der Fallunterscheidungen im Vergleich zu Programm A und B. Da diese jedoch vom Analysewerkzeug nicht erkannt werden, enthält der Bericht jedesmal alle Elemente in allen Varianten und wird daher wahrscheinlich deutlich überschätzt. Das Analysewerkzeug scheint also ohne die Erkennung der Fallunterscheidungen recht anfällig, Steuerprogramme die anhand der Eingabe entscheiden welche Verarbeitung denn ausgeführt wird, zu überschätzen.

Insgesamt zeigt die Schätzung gute Ergebnisse, auch wenn die fehlende Analyse der Fallunterscheidungen eine perfekte Zuordnung verhindert.

Frage 4b: Wird die Optimierung für Einstiegspunkt A gefunden?

Nachdem bekannt ist, dass die Kosten pro Aufruf für Programm A verringert werden müssen, stellt sich die Frage, welches der 90 beteiligten Programme denn optimiert werden soll.

Dazu werden die beiden beschriebenen Metriken für die beteiligten Programme erstellt und der Kostenrang des einen optimierten Programmes analysiert. Da die Optimierung die Kosten des Programmes verringert, aber nicht die Aufrufkette angepasst hat, würde man also erwarten, dass vorallem die ,,Kosten mal Aufrufe''-Metrik gute Ergebnisse liefert.

Die Ergebnisse der ersten Analyse, also der Kosten samt Unterprogramme im Aufrufbaum findet sich in Tabelle 6.8. Das optimierte Programm belegt hier Rang 8 von 90 Programmen.

Die Ergebnisse der zweiten Metrik, also Kosten des Programmes ohne Unterprogrammaufrufe mal Anzahl der Aufrufe ergibt Rang 5 von 90 Programmen und ist in Abbildung 6.9 zu sehen.

Wie erwartet liegt also die zweite Metrik besser als die erste, jedoch finden beide Metriken das optimierte Programm erfreulich schnell. Da aus der Tatsache, dass dieses Programm optimiert worden ist nur ableitbar war, dass es vor der Optimierung teuer war und nicht dass es das teuerste war, können diese Ergebnisse nicht weiter interpretiert werden.

Insgesamt zeigt sich, dass die statische Analyse das optimierte Programm in allen Metriken zügig findet und der Optimierer spätestens bei der Analyse des achten von 90 Programmen fündig werden würde.

Frage 4c: Wird die Optimierung im Programm selbst gefunden?

Vergleicht man den Quellcode des optimierten Programmes vor und nach der Optimierung, so wird ersichtlich, dass hauptsächlich eine Section, also ein Abschnitt optimiert wurde. Daher stellt sich die Frage, ob die Analyse diesen entsprechend teuer bewertet hat.

Dazu wird analog zu den Programmen die Kostenschätzung pro Abschnitt einmal mit und einmal ohne die Kosten für aufgerufene Abschnitte betrachtet.

Die Ergebnisse der Kostenschätzung der Sektionen mit allen programminternen Aufrufen aber ohne Aufrufe von anderen Programmen finden sich in Tabelle 6.10.

Kostenschätzung der Sektionen im optimierten Programm					
Programm	Aufgerufen von	Kosten	Faktor	Gesamtkosten	Reihenfolge
A	CICS	8.155.261.573	1	8.155.261.573	1
B1	A	12.583	1	12.583	30
B2	A	1.631.046.725	5	8.155.233.625	2
C1	B2	1.198.709.612	5	5.993.548.060	3
C2	B2	203.020.508	5	1.015.102.540	4
C3	B2	201.707.326	5	1.008.536.630	5
C4	B2	7.058.908	5	35.294.540	6
C5	B2	812.122	40	32.484.880	7
C6 (opt)	B2	3.050.148	10	30.501.480	8
C7	B2	81.855	10	818.550	9
C8	B2	10.430	75	782.250	10
C9	B2	6.571	95	624.245	11
C10	B2	15.140	30	454.200	12
C11	B2	81.855	5	409.275	13
C12	B2	9.009	40	360.360	14
C13	B2	49.009	5	245.045	15
C14	B2	21.301	10	213.010	16
C15	B2	42.132	5	210.660	17
C16	B2	4.989	40	199.560	18
C17	B2	1.750	95	166.250	19
C18	B2	21.232	5	106.160	20
C19	B2	8.773	10	87.730	21
C20	B2	17.520	5	87.600	22
C21	B2	15.582	5	77.910	23
C22	B2	13.123	5	65.615	24
C23	B2	7.282	5	36.410	25
C24	B2	5.383	5	26.915	26
C25	B2	543	40	21.720	27
C26	B2	1.750	10	17.500	28
C27	B2	3.353	5	16.765	29
C28	B2	843	5	4.215	31
C29	B2	107	5	535	32

Tabelle 6.8.: Auflistung des Aufrufbaumes des Einstiegspunktes A mit Kosten bis zur dritten Aufrufebene. Dabei ist A der Einstiegspunkt. Wie oft ein Programm bei einem Aufruf von Programm A aufgerufen wird, steht in der Spalte „Faktor". Die Gesamtkosten setzen sich dabei aus dem Produkt aus Faktor und Kosten samt Unterprogrammaufrufen zusammen. Insgesamt werden 90 Programme beim Aufruf von A ausgeführt.

Kostenschätzung der Programme für Einstiegspunkte A				
Name	Kosten pro Aufruf	Anzahl Aufrufe	Gesamtkosten	Rangfolge
P1	2,32E+07	2,40E+02	5,57E+09	1
P2	5,47E+07	4,00E+01	2,19E+09	2
P3	1,26E+04	8,87E+03	1,12E+08	3
P4	6,90E+04	6,40E+02	4,41E+07	4
P5 (opt)	1,25E+06	3,50E+01	4,39E+07	5
P6	1,53E+04	1,83E+03	2,80E+07	6

Tabelle 6.9.: Kostenschätzung der Programme ohne Unterprogrammaufrufe für Einstiegspunkt A

Kostenschätzung der Sektionen im optimierten Programm		
Name	Kosten	Rang
S1	506674	1
S2	400684	2
S3 (optimiert)	87101	3
S4	86242	4
S5	47529	5

Tabelle 6.10.: Top 5 Kostenverursacher von 43 Sektionen im optimierten Programm für Einstiegspunkt A. Die Kosten setzten sich dabei aus den Kosten für den Abschnitt sowie den Kosten für alle aufgerufenen Abschnitte zusammen. Programmaufrufe fließen nicht mit ein.

Kostenschätzung der Sektionen im optimierten Programm					
Programm	Ebene	Kosten	Faktor	Gesamtkosten	Reihenfolge
B (opt)	CICS	9.817.904.183	1	9.817.904.183	1
B1	B	1.631.046.725	6	9.786.280.350	2
B2	B	3.050.148	6	18.300.888	3
B3 (opt)	B	36.985	276	10.207.860	4
B4	B	128.808	16	2.060.928	5
B5	B	13.214	31	409.634	6
B6	B	1.783	91	162.253	7
B7	B	6.682	11	73.502	8
B8	B	10.085	6	60.510	9
B9	B	3.353	11	36.883	10
B10	B	3.682	6	22.092	11
B11	B	3.557	6	21.342	12
B12	B	3.365	6	20.190	13
B13	B	1.693	7	11.851	14
B14	B	843	6	5.058	15
B15	B	4.989	1	4.989	16
B16	B	1.711	1	1.711	17
B17	B	129	1	129	18

Tabelle 6.11.: Auflistung des Aufrufbaums des Einstiegspunktes B mit Kosten bis zur zweiten Aufrufebene. Wie oft ein Programm bei einem Aufruf von Programm B aufgerufen wird, steht in der Spalte „Faktor". Die Gesamtkosten errechnen sich dabei aus dem Produkt aus Faktor und Kosten samt Unterprogrammaufrufen. Insgesamt werden 99 Programme beim Aufruf von B ausgeführt.

Die optimierte Section ist also die drittteuerste von 43 für das Programm. Rechnet man nur die Kosten der Section ohne programminterne Aufrufe, so ergibt sich sogar Rang 1 von 43, da S1 und S2 den Großteil ihrer Kosten per Aufruf von S3 verursachen. Der zu optimierende Abschnitt wäre also anhand dieser Schätzung zielsicher gefunden worden.

Frage 4d: Wird die Optimierung für Einstiegspunkt B gefunden?

Auch für Einstiegspunkt B stellt sich die Frage, wie schnell die statische Analyse die zwei darin optimierten Programme findet.

Für die Analyse des Aufrufbaums, abgebildet in Tabelle 6.11, ergeben sich folgende Ergebnisse:

- Da der Einstiegspunkte selbst optimiert wurde, liegt Programm B in dieser Metrik natürlich auf Rang 1 der 99 beteiligten Programme.
- Das zweite Programm findet sich auf Platz 4 der 99 Programme.

Für die zweite Metrik, der Kosten der Programme ohne Unterprogrammaufrufe multipliziert mit der Anzahl der Aufrufe, ergibt sich folgendes Bild:

- Der Einstiegspunkt selbst findet sich auf Rang 57 von 99 Programmen innerhalb der Aufrufkette.

- Das zweite optimierte Programm findet sich auf Rang 15 von 99 Programmen innerhalb der Aufrufkette.

Somit ergibt sich ein gemischtes Ergebnis. Einerseits findet die Analyse des Aufrufbaumes die gesuchten Programme sofort, andererseits liegt die zweite Schätzung deutlich daneben, was darauf schließen lässt, das die Kosten deutlich unterschätzt wurden. Da für diese Optimierung der Maßnahmenkatalog bekannt ist, soll dieser zur Interpretation der Ergebnisse sowie möglicher Fehlerursachen herangezogen werden.

Zum einem gab es für diesen Einstiegspunkt Probleme bei Spitzenlast, was nahelegt das Einstiegsprogramm selbst zu optimieren. Die Optimierungsmaßnahme bestand darin, dass zu Spitzenzeiten unnötige Berechnungen verschoben wurden. Da diese Logik hauptsächlich mit Fallunterscheidungen implementiert wurde, diese aber die Schätzung nicht beeinflussen, kann dies keine Erklärung sein.

Die zweite Maßnahme ersetzte einen Teil der Datenbankzugriffe durch sogenannte Multi-Row-Fetches, verringerte also die Anzahl der nötigen Datenbankzugriffe. Deren Anzahl wurde zusätzlich noch durch das Zusammenlegen von Abfragen verringert. Damit sich solche Optimierungen lohnen, muss es sich entweder um viele Datenbankzugriffe handeln oder um sehr teure Datenbankzugriffe. Da ersteres aufgrund der Schleifenstruktur im Programmcode nicht sichtbar ist und letzteres aufgrund der nicht vorhandenen Kostenschätzung für Datenbankzugriffe nicht ersichtlich ist, könnte dies eine Erklärung sein.

Frage 4e: Wird die Optimierung in den Programmen gefunden?

Aufgrund der gerade beschriebenen Vielfalt der Optimierungen, die hier stattgefunden haben, lässt sich diese Frage hier nicht beantworten. Zum einem beeinflusst nämlich die Veränderung des Aufrufgraphen nicht die Kostenschätzung für das Programm alleine, zum anderem wurde durch den Einbau des Multi-Row-Fetches eine Vielzahl kleiner Änderungen in vielen Bereichen vorgenommen, so dass nicht nachvollziehbar ist, inwiefern welcher Bereich optimiert wurde.

6.5.5. Frage 5: Zeigen die Schätzungen die Einsparungen der Optimierungen?

Wenn die statische Performance-Analyse den Anspruch erhebt, die Kostenrangfolge richtig zu schätzen, dann müsste auch eine optimierte Version eines Programm billiger als seine Ursprungsversion sein. Interessant ist auch die Frage, ob die Verringerung der Kostenschätzung, in Prozent gemessen, vernünftige Werte liefert.

Dazu wurde der Codestand vor und nach der Optimierung kostengeschätzt und die Ergebnisse sowohl auf der Programm- als auch auf der Systemebene verglichen.

Die Optimierung in Programm A ohne Unterprogrammaufrufe zeigte eine Einsparung von knapp 16%. Die Kosten für den Einstiegspunkt A verringerten sich hierbei um knapp 1‰. Die Optimierung in Programm B selbst zeigte eine Einsparung von 3%, während das

zweite veränderte Programm einen Kostenzuwachs von 70% erzielte. Insgesamt verringerte sich die Kostenschätzung, auch aufgrund des veränderten Aufrufgraphens für Einstiegspunkte B, um 0,3 ‰. Derselbe Einsparungswert ergab sich auch für die Summe aller Einstiegspunkte.

Während die Einsparung für Programm A alleine durchaus plausibel erscheint, müssen die anderen Werte genauer betrachtet werden. Die geringe Gesamteinsparung im Gesamtsystem scheint, zusammen mit den Informationen über die Kostenränge der Programme, darauf hinzudeuten, dass die Reihenfolge der Programme eingeschätzt werden kann, aber nicht deren Kostenverhältnisse. Die Aussage „Programm X ist doppelt so teuer wie Programm Y", weil die Kostenschätzung doppelt so hoch ist, scheint also nicht zu gelten.

Der Kostenzuwachs für das zweite optimierte Programm für Einstiegspunkte B erklärt sich aus der Optimierungshistorie. Der hier eingebaute Multi-Row-Fetch benötigt extra Verarbeitungsschritte, wie zum Beispiel zusätzliche Schleifen. Dass das Query nunmehr wesentlich weniger oft aufgerufen wird, ist aber aufgrund der nicht betrachteten Fallunterscheidungen nicht sichtbar, so dass sich logischerweise eine Erhöhung der Kostenschätzung ergibt.

6.5.6. Frage 6: Korreliert die statische Schätzung mit den Messdaten der dynamischen Analyse?

Nachdem die Informationen zur Optimierungshistorie im Rahmen der Evaluation ausgewertet worden sind, widmet sich diese Frage den zukünftigen Optimierungen. Da die dynamischen Messergebnisse für den Quellcode nach der Optimierung bekannt sind, lässt sich nun untersuchen, inwieweit man zur Optimierung aus den statischen Kostenschätzungen dieselben Schlüsse ziehen würde. Dazu muss natürlich angenommen werden, dass die dynamische Messunge das System perfekt analysiert hat.

Zur Ermittlung der Übereinstimmung der beiden Verfahren wird das Verfahren der gewichteten Treffer nach Wall [47] angewandt, das Aussagen über die Trefferquote der teuersten Programme liefert. Während die Trefferquote normalerweise mit „Trefferzahl geteilt durch Gesamtzahl" angegeben wird, berücksichtigt diese Metrik zusätzlich noch das Gewicht der Treffer. Im Fall der Kostenschätzung heißt das also, dass die Trefferquote für eine Menge der teuersten Programme „TopX" folgendermaßen definiert ist:

$$Trefferquote\ nach\ [47] := \frac{\sum_{Treffer\ t\ \in\ Programme\ \in\ TopX} Kosten(t)}{\sum_{Programm\ p\ \in\ TopX} Kosten(p)}$$

Das heißt also, wenn die Top 5 Programme gesucht sind, erhält die statische Analyse für jedes Programm, dass auch in der eigenen Analyse in den Top 5 ist, die Kosten gut geschrieben. Die Trefferquote ergibt sich dann aus der Summe der Kosten der Treffer durch die Summe der Kosten der Top 5 Programme.

Diese Formel trägt der Tatsache Rechnung, dass eine Trefferquote von beispielsweise vier der fünf teuersten Programme nicht berücksichtigt, ob das teuerste Programm oder das billigste der Top 5 nicht enthalten ist. Für die Optimierung kann dies aber einen gewaltigen Unterschied machen, vor allem wenn das teuerste Programme viele Male teurer als das andere Programm ist. Mit Hilfe dieser Metrik wird dieser Effekt berücksichtigt.

Die Ergebnisse der Vergleichsmessung für die CICS Einstiegspunkte finden sich in Tabelle 6.12, die Ergebnisse für die restlichen Einstiegspunkte in Tabelle 6.13.

Vergleich der Ergebnisse der dynamischen und der statischen Kostenschätzung (CICS)						
Programm	dyn. Prozent	Rang		stat. Kosten	stat. Prozent	Rang
A (opt)	51,63%	1		6,81E+15	32,04%	2
B (opt)	28,08%	2		2,22E+14	1,04%	3
C	19,65%	3		1,42E+16	66,91%	1
D	0,49%	4		4,08E+10	0,00%	4
E	0,13%	5		2,25E+10	0,00%	5
F	0,01%	6		1,33E+10	0,00%	6
G	0,00%	7		1,14E+07	0,00%	8
H	0,00%	8		1,30E+07	0,00%	7

Tabelle 6.12.: Vergleich der Ergebnisse der dynamischen und der statischen Kostenschätzung für die CICS Einstiegspunkte

Wie in den Tabellen eingezeichnet, wurde davon ausgegangen, dass der Optimierer ein Viertel aller Programme zur Optimierung, in dem Fall also die zwei bzw. sieben teuersten Programme in Betracht zieht. Für die gewichtete Trefferquote ergibt sich somit:

- Trefferquote Top 25% CICS Einstiegspunkte: 65%
- Trefferquote Top 25% restliche Einstiegspunkte: 96%
- Trefferquote Zufall: 25%

Wendet man dasselbe Verfahren für die billigsten Programme an, wobei die Kosten hier die des teuersten Programms abzüglich der eigenen Kosten darstellen, so erhält man:

- Trefferquote Billig 25% CICS Einstiegspunkte: 100%
- Trefferquote Billig 25% restliche Einstiegspunkte: 57%
- Trefferquote Billig 25% Zufall: 25%

Wie bereits in Frage 4a festgestellt, wurde der Einstiegspunkt C deutlich überschätzt, weshalb die Top 2 Metrik nur 65% statt 100% erzielt. Für die restlichen Einstiegspunkte zeigt sich eine massive Kostendominanz von Programm A, die erfreulicherweise auch in der statischen Analyse als solche erkannt wird. Die restlichen Werte sind jedoch zum Teil erstaunlich, da Programm B2 und E2 aus den Top sieben Kostenverursacher der dynamischen Analyse in der statischen Schätzung als extrem billig dargestellt werden, während Programm AA2 deutlich überschätzt wird.

Bei der Analyse für Programm B2 zeigt sich, dass dieses Programm für die regelmäßige Protokollierung zuständig ist. Die dazu verwendeten Programme fehlen jedoch im Quellcode, so dass ein Aufruf zu einem unbekannten Programm zu finden ist. Dieser Aufruf wird folgendermaßen kommentiert: „Hole alle ...-daten aus der Datenbank und die dazu nötigen Zusatzinformationen". Da die nächste Anweisung eine Schleife ist, die solange iteriert bis ein Flag falsch ist und diese Daten in einer Tabelle mit mehreren Millionen

Vergleich der Ergebnisse der dynamischen und der statischen Kostenschätzung						
Programm	dyn. Prozent	Rang		stat. Kosten	stat. Prozent	Rang
A (opt)	91,87%	1		20.061.524.597.051.200	100,00%	1
B2	3,65%	2		63.327.234	0,00%	17
C2	3,22%	3		69.350.189.244	0,00%	2
D2	0,50%	4		2.383.236.660	0,00%	6
E2	0,31%	5		8.484.148	0,00%	20
F2	0,16%	6		37.219.294.640	0,00%	3
G2	0,13%	7		478.621	0,00%	25
H2	0,05%	8		216.929.986	0,00%	13
I2	0,02%	9		1.130.923	0,00%	23
J2	0,02%	10		1.816.182.900	0,00%	7
K2	0,02%	11		293.262.427	0,00%	9
L2	0,01%	12		2.428.856.458	0,00%	5
M2	0,01%	13		430.795.820	0,00%	8
N2	0,01%	14		258.624.698	0,00%	10
O2	0,01%	15		226.754.730	0,00%	12
P2	0,00%	16		239.152.914	0,00%	11
Q2	0,00%	17		213.094.245	0,00%	14
R2	0,00%	17		120.500.739	0,00%	15
S2	0,00%	19		13.762.074.150	0,00%	4
T2	0,00%	19		7.593.390	0,00%	21
U2	0,00%	21		35.860.848	0,00%	18
V2	0,00%	22		122.928	0,00%	27
W2	0,00%	23		889.120	0,00%	24
X2	0,00%	24		19.777.432	0,00%	19
Y2	0,00%	24		1.454.427	0,00%	22
Z2	0,00%	24		399.232	0,00%	26
AA2	0,00%	27		68.524.532	0,00%	16

Tabelle 6.13.: Vergleich der Ergebnisse der dynamischen und der statischen Kostenschätzung für die restlichen Einstiegspunkte

Einträgen liegen, erscheint dies eine wahrscheinliche Erklärung für die hohen Kosten des Programmes.

Die Erklärung für Programm E2 fällt wesentlich einfacher aus, denn hier ist der Datenbankzugriff und die Verarbeitung direkt ersichtlich. Da jedoch die Kosten pro Datenbankzugriff fest sind, hier aber in einer Schleife auf eine der größten Tabellen im System zugegriffen wird, wird das Programm deutlich unterschätzt. Diese Vermutung wird dadurch bestärkt, dass dieses Programm bei dem Versuch der Kostenbestimmung per Tabellengrößen einen extremen Kostenzuwachs erfahren hatte.

Bleibt die Frage, wieso das Programm AA2 viel zu teuer ist. Das Programm selbst wird richtigerweise recht billig eingeschätzt. Jedoch enthält die Verarbeitung für den Fehlerfall einen Aufruf an das optimierte Programm in Transaktion A. Da dieses extrem teuer ist und die Fallunterscheidung nicht ausgelesen wird, ist dies für den Kostenzuwachs verantwortlich. Da das Programm laut dynamischer Messung aber nur vier Mal aufgerufen wurde, ist es unwahrscheinlich, dass dieser Fehler im untersuchten Zeitraum eingetreten ist, wodurch sich die Abweichung erklärt.

Ebenfalls wichtig ist die Angabe, wieviele der billigsten Programme richtig erkannt werden, denn diese dürfen auf gar keinen Fall optimiert werden. Für die CICS Einstiegspunkte werden diese erfreulicherweise direkt richtig erkannt, für die restlichen Einstiegspunkte ist die Erkennung nicht so gut.

Auf jeden Fall ergibt sich, dass die statische Performance-Analyse in der derzeitigen Implementierung wertvolle Empfehlungen geben kann, die bei Weitem besser als eine zufällige Auswahl sind. Werden zusätzlich noch die genannten Probleme behoben, sollte sich die Präzision spürbar erhöhen.

6.5.7. Frage 7: Bringt die Performance-Anti-Pattern-Suche brauchbare Ergebnisse?

Die letzte Frage die noch verbleibt, ist die Frage, ob die Anti-Pattern-Erkennung einen Beitrag zur Optimierung leisten kann.

Dazu wurde die Erkennung auf den Quellcode angewandt und die gefundenen Ergebnisse per Hand verifiziert, wobei folgende Ergebnisse erzielt wurden:

- Ergebnis Bubble Sort: Kein Treffer
- Ergebnis korrelierte Unterabfrage: keine Treffer
- Ergebnis Schatzsuche: Ein Treffer. Die Analyse ergab hierbei, dass die Pattern-Instanz nicht vorliegt, es sich jedoch um unnötigen Code handelt. Der Pseudocode der gefundenen Codestelle sieht dabei folgendermaßen aus:

  ```
  var foo, payload
  loop:
      SQL: SELECT foo, payload INTO :foo, :payload WHERE foo = :foo
  ```

  Offensichtlich wäre hier die erneute Zuweisung über INTO für die Variable foo nicht notwendig, da foo aufgrund der WHERE-Bedingung exakt den Wert vor der Zuweisung erneut zugewiesen bekommt.

- Ergebnis Hinweise auf Schatzsuche: Zwei Treffer:
  Einmal handelt es sich um einen Fehlalarm, denn hier wird in einer Schleife über mehrere Elemente anhand der Elementbeschreibung eine Zusatzinformation aus der Datenbank geholt.
  Der zweite Hinweis ist hingegen ein Treffer, der auch anhand von den Analyseergebnissen der itestra bestätigt werden konnte. Im Code sieht dies folgendermaßen aus: Eine Section enthält eine Datenbankabfrage. Deren Rückgabewert wird zwischengespeichert und später im Code in die Bedingungsvariable zurückgeschrieben. Da diese Section aber in einer Schleife ausgeführt wird, die erst beim Setzen eines Flags terminiert, handelt es sich hierbei um einen Kandidaten für die Schatzsuche. Die Analyse des Flagmechanismusses zeigt dabei, dass es sich wirklich um eine Schatzsuche handelt.

Insgesamt wurden also drei potentielle Performance-Anti-Pattern gefunden, von denen eines sich als korrekt herausgestellt hat, eines falsch war und eines auf unnötigen Code hinwies. Dies zeigt, dass die Trefferquote zwar nicht gut ist, aufgrund der extrem geringen Fundanzahl aber für die Optimierung durchaus brauchbar, denn der Aufwand, drei Codestellen zu überprüfen, erscheint vertretbar.

Am Rande sei das Ergebnis der Hinweise auf Einzelsatzverarbeitung bemerkt. Diese wurden in 25 von 404 Dateien an insgesamt 71 Stellen, also in 6% der Programme, gefunden. Da diese Warnung per Definition korrekt ist, da eine Schleife mit einem SQL-Statement nunmal eine Tatsache ist, lässt sich diese Ergebnis nicht weiter deuten.

6.5.8. Fazit

Nach der Beantwortung der sechs Teilfragen stellt sich nun die Frage, ob die Analyse als praxistauglich eingestuft werden kann. Ein großes Problem ist sicherlich die fehlende Kostenschätzung der Datenbankzugriffe, die die Genauigkeit der Kostenschätzung doch in Mitleidenschaft zieht und zusammen mit dem mangelnden Auslesen der Fallunterscheidungen wohl für viele Ungenauigkeiten verantwortlich ist. Trotzdem zeigt sich beim Vergleich der Schätzung mit der dynamischen Kostenmessung, dass die Analyse durchaus die wichtigen Programmstellen richtig einschätzen kann. Diese Aussage wird zusätzlich durch die guten Ergebnisse beim Vergleich der Schätzung mit den Optimierungsmaßnahmen und der brauchbaren Trefferquote bei den Anti-Pattern gestützt. Zusammen mit der schnellen Laufzeit und der Robustheit der Ergebnisse gegenüber der Parameterwahl wird die Analyse als praxistauglich eingestuft.

7. Abschluss

7.1. Zusammenfassung

In dieser Arbeit wurden zwei Fragen untersucht:

- Lassen sich mit Hilfe rein statischer Analysen Performance-technisch signifikante Systemteile und somit Optimierungskandidaten in betrieblichen Informationssystemen mit relationalen Datenbanken auffinden?
- Wie müsste ein solches Analysesystem aufgebaut sein und inwieweit lassen sich bekannte Performance-Anti-Pattern nutzen, um Optimierungsmaßnahmen vorzuschlagen?

Nachdem die Literaturrecherche zu diesem Thema keine passende Lösung finden konnte, wurde ein eigenes System zur statischen Kostenschätzung implementiert. Dieses basiert auf der Grundidee des Performance-Modells, einem abstrakten Modell des untersuchten Quellcodes, der auf seine Struktur und die Performance-kritischen Elemente reduziert wurde. Genau wie beim abstrakten Syntaxbaum erlaubt dieser Zwischenschritt einfachere und schnellere Analysealgorithmen zu entwerfen. Zuerst wurden die allgemeinen Anforderungen an ein solches Modell geklärt, wobei vor allem die Sprachen Java, COBOL und SQL untersucht wurden.

Anschließend wurde das gefundene Modell anhand der in der Literatur gefundenen Performance-Anti-Pattern auf Plausibilität geprüft, da eine Kostenschätzung, die ein solches Problemmuster nicht entsprechend teurer als eine optimierte Version desselben Problems einstufen kann, nicht brauchbar ist. Im letzten Schritt wurde eine Implementierung für COBOL vorgenommen und ein betriebliches COBOL-Informationssystem mit Datenbankanbindung damit analysiert. Zur Überprüfung der Praxistauglichkeit des Ansatzes wurde die Analyse anhand von sieben Fragen evaluiert.

In dieser Arbeit konnte gezeigt werden, dass trotz der grundsätzlich eingeschränkten Möglichkeiten der statischen Performance-Analyse, eine starke Eingrenzung der Performance-kritischen Systemteile möglich ist. Natürlich ist diese Abschätzung nicht vergleichbar mit der Präzision einer dynamischen Messung, denn ein Teil aller möglichen Programmkonstrukte ist nunmal statisch nicht auslesbar. Dazu gehören zum Beispiel Design-Pattern wie die Factory oder Abstraktionsschichten durch die der Kontrollfluss auf Basis von dynamischen Sprüngen gesteuert wird und somit kaum mehr statische analysierbar ist. Trotz dieser Widrigkeiten können in der Praxis offenbar mit Hilfe von einfachen Annahmen und Heuristiken zumeist die Kostenverbraucher richtig identifiziert und dank der Anti-Pattern sogar in einigen Fällen direkt die Optimierungswege aufgezeigt werden. Zudem erwiesen sich die konkreten Parameter für die Heuristiken nicht als entscheidend für die erzielten Ergebnisse. Damit sollte in der Praxis mit Hilfe

der wesentlich billigeren und schnelleren statischen Analyse eine zügig Optimierung des Systems möglich sein, auch wenn nicht jeder als teuer befundene Systemteil ein Top-Kostenverbraucher sein wird.

7.2. Ausblick

Die Bewertung des implementierten Analysewerkzeugs konnte im Rahmen dieser Arbeit nur anhand eines einzigen Systems evaluiert werden. Diese Ergebnisse sollten daher auf eine breitere Basis gestellt werden und anhand weiterer Systeme bestätigt werden.

Trotzdem konnten in der Evaluation schon grundlegende Verbesserungsmöglichkeiten aufgezeigt werden. So traten mehrfach Schätzfehler aufgrund der fehlenden Einschätzung der Kosten für Datenbankzugriffe auf. Da diese Schätzung oftmals von der Datenbank selbst erbracht werden kann, wäre die Einbindung dieser Funktionalität in das Werkzeug eine naheliegende Lösung für das Problem. Eine andere Verbesserungsmöglichkeit besteht in der Erweiterung der Analyse um Fallunterscheidungen. Dafür müssten mehrere Messdaten zu Testsystemen vorliegen, um nötige Annahmen und mögliche Heuristiken zur Kostenschätzung, wie zum Beispiel die im Theorieteil Genannten, zu verifizieren. Eine weitere Möglichkeit die Präzision der Analyse zu verbessern, ist die zusätzliche Nutzung gewisser Informationen, die aus dynamischen Verfahren gewonnen wurden, und so die Vorteile beider Welten zu vereinen.

Nachdem in dieser Arbeit gezeigt wurde, dass statische Analyse zum Auffinden von Performance-kritischen Systemteilen brauchbar ist, stellt sich natürlich final die Frage, ob damit ein schnellerer Optimierungsprozess als auf Basis der dynamischen Performancemessungen möglich ist.

Literaturverzeichnis

[1] *DB2 Udb for Z/os Version 8 Performance Topics*. IBM Press, 2005.

[2] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, D. Ramírez, G. Román, and D. Zanardini. Termination and cost analysis with costa and its user interfaces. *Electron. Notes Theor. Comput. Sci.*, 258:109–121, December 2009.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 157–172, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] Christophe Alias and Denis Barthou. Algorithm recognition based on demand-driven data-flow analysis. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, pages 296–, Washington, DC, USA, 2003. IEEE Computer Society.

[5] Thomas Ball and James R. Larus. Branch prediction for free. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 300–313, New York, NY, USA, 1993. ACM.

[6] Cathal Boogerd and Leon Moonen. Prioritizing software inspection results using static profiling. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 149–160, Washington, DC, USA, 2006. IEEE Computer Society.

[7] Cathal Boogerd and Leon Moonen. On the use of data flow analysis in static profiling. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:79–88, 2008.

[8] Raymond P. L. Buse and Westley Weimer. The road not taken: Estimating path execution frequency statically. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 144–154, Washington, DC, USA, 2009. IEEE Computer Society.

[9] Rim Chaabane. Poor performing patterns of code: Analysis and detection. In *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France*, pages 501–502. IEEE, 2007.

[10] Rim Chaabane and Françoise Balmas. Applying static and dynamic analysis in a legacy system to study the behaviour of patterns of code during executions: an industrial experience in workshop on program comprehension through dynamic analysis (pcoda'08). In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 37–41, Washington, DC, USA, 2008. IEEE Computer Society.

[11] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '98, pages 34–43, New York, NY, USA, 1998. ACM.

[12] G. Cong, S. Seelam, I. Chung, H. Wen, and D. Klepacki. Towards a framework for automated performance tuning. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[13] Vittorio Cortellessa, Anne Martens, Ralf Reussner, and Catia Trubiani. Towards the identification of "guilty"performance antipatterns. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, WOSP/SIPEW '10, pages 245–246, New York, NY, USA, 2010. ACM.

[14] Benoit Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. Automatic sql tuning in oracle 10g. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 1098–1109. VLDB Endowment, 2004.

[15] Arie Deursen and T. Kuipers. Rapid system understanding: two cobol case studies. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, 1998.

[16] Bruce Eckel. *Thinking in Java (4th Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[17] Joseph A. Fisher and Stefan M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, ASPLOS-V, pages 85–95, New York, NY, USA, 1992. ACM.

[18] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[19] Hasyim Gautama and Arjan J. C. van Gemund. Static performance prediction of data-dependent programs. In *Proceedings of the 2nd international workshop on Software and performance*, WOSP '00, pages 216–226, New York, NY, USA, 2000. ACM.

[20] Olivier Kaczor, Yann-Gaël Guéhéneuc, and Sylvie Hamel. Identification of design motifs with pattern matching algorithms. *Inf. Softw. Technol.*, 52:152–168, February 2010.

[21] Christian Kramer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*, pages 208–, Washington, DC, USA, 1996. IEEE Computer Society.

[22] Jochen Kreimer. Adaptive detection of design flaws. *Electron. Notes Theor. Comput. Sci.*, 141:117–136, December 2005.

[23] Radu Marinescu. Detecting design flaws via metrics in object-oriented systems. In *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, TOOLS '01, pages 173–, Washington, DC, USA, 2001. IEEE Computer Society.

[24] Radu Marinescu. Measurement and quality in object-oriented design. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 701–704, Washington, DC, USA, 2005. IEEE Computer Society.

[25] R.C. Metzger and Z. Wen. *Automatic algorithm recognition and replacement: a new approach to program optimization*. MIT Press, 2000.

[26] Marianne de Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static loop bound analysis of c programs based on flow analysis and abstract interpretation. In *Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 161–166, Washington, DC, USA, 2008. IEEE Computer Society.

[27] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 13–, Washington, DC, USA, 2001. IEEE Computer Society.

[28] Matthew James Munro. Product metrics for automatic identification of "bad smell"design problems in java source-code. In *Proceedings of the 11th IEEE International Software Metrics Symposium*, pages 15–, Washington, DC, USA, 2005. IEEE Computer Society.

[29] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 338–348, New York, NY, USA, 2002. ACM.

[30] David Lorge Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[31] Trevor Parsons and John Murphy. Detecting performance antipatterns in component based enterprise systems. *Journal of Object Technology*, 7(3):55–91, 2008.

[32] Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 67–78, New York, NY, USA, 1995. ACM.

[33] Santanu Paul. Scruple: a reengineer's tool for source code search. In *Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research - Volume 1*, CASCON '92, pages 329–346. IBM Press, 1992.

[34] P. Puschner and Ch. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Syst.*, 1:159–176, September 1989.

[35] Alan Rodger. Cobol – continuing to drive value in the 21st century. White Paper, Datamonitor, November 2008.

[36] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.

[37] Bart Smaalders. Performance anti-patterns. *Queue*, 4:44–50, February 2006.

[38] Connie U. Smith and Lloyd G. Williams. Software performance antipatterns. In *Proceedings of the 2nd international workshop on Software and performance*, WOSP '00, pages 127–136, New York, NY, USA, 2000. ACM.

[39] Connie U. Smith and Lloyd G. Williams. New software performance antipatterns: More ways to shoot yourself in the foot. In *28th International Computer Measurement Group Conference*. Computer Measurement Group, 2002.

[40] Connie U. Smith and Lloyd G. Williams. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *29th International Computer Measurement Group Conference*. Computer Measurement Group, 2003.

[41] Ahmad Taherkhani. Recognizing sorting algorithms with the c4.5 decision tree classifier. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ICPC '10, pages 72–75, Washington, DC, USA, 2010. IEEE Computer Society.

[42] Nathan R. Tallent, John M. Mellor-Crummey, and Michael W. Fagan. Binary analysis for measurement and attribution of program performance. *SIGPLAN Not.*, 44:441–452, June 2009.

[43] B. Tate, M. Clark, and B. Lee. *Bitter EJB*. Manning Pubs Co Series. Manning, 2003.

[44] Mark G. J. van den Brand, A. Sellink, and C. Verhoef. Control flow normalization for cobol/cics legacy system. In *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98)*, CSMR '98, pages 11–, Washington, DC, USA, 1998. IEEE Computer Society.

[45] M.G.J. van den Brand, M.P.A. Sellink, and C Verhoef. Obtaining a cobol grammar from legacy code for reengineering purposes. In M.P.A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, electronic Workshops in Computing*. Springer verlag, 1997.

[46] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate static estimators for program optimization. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 85–96, New York, NY, USA, 1994. ACM.

[47] David W. Wall. Predicting program behavior using real or estimated profiles. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 59–70, New York, NY, USA, 1991. ACM.

[48] Joachim Wendt. *COBOL.: Einführung mit PC-Spracherweiterung und Übungen.* Gabler, 1994.

[49] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th annual international symposium on Microarchitecture*, MICRO 27, pages 1–11, New York, NY, USA, 1994. ACM.