Performance-Analyse auf Mainframe-Systemen mittels Profiling

Stefan Laner und Matheus Hauder

Lehrstuhl für Informatik 19 - sebis
Technische Universität München (TUM)
Boltzmannstrasse 3
85748 Garching bei München
{laner,hauder}@in.tum.de

Abstract: Eine Optimierung der Performance von Anwendungen verbessert deren Laufzeiten und senkt deren Betriebskosten. Um Performance-Optimierungen vornehmen zu können, müssen zunächst Optimierungspotentiale identifiziert werden. Dazu werden Performancedaten erfasst und analysiert. Unternehmensanwendungen, die auf IBM Mainframes betrieben werden, lassen sich mit aktuellen Verfahren allerdings nur umständlich analysieren. Die derzeit gängigen Werkzeuge liefern eine Vielzahl an Informationen, die manuell ausgewertet werden müssen. Eigentlich zusammengehörige Informationen sind dort zum Teil über verschiedene Reports verteilt und auffällige Programmteile oftmals nur über geeignete Aggregation identifizierbar. Der in diesem Artikel vorgestellte Ansatz überführt die Daten aus einem konkreten Profiling-Werkzeug in ein Meta-Modell, welches für eine automatische Analyse herangezogen werden kann. Zudem beschreibt er Beispiele für Anti-Pattern, also Muster von bekannten Performance-Problemen, die in realen Systemen beobachtet wurden und wie sich diese aus dem Meta-Modell ermitteln lassen.

1 Einführung

IBM Mainframe Systeme finden entgegen früherer Prognosen nach wie vor starke Verwendung im Umfeld von Großunternehmen und Behörden. Eine Neuentwicklung der darauf betriebenen Anwendungen oder gar eine Migration auf andere Plattformen ist aus Zeit- und Kostengründen sowie auf Grund des damit verbundenen Umstellungsrisikos in der Regel nicht ohne Weiteres durchführbar. Eine Wartung und Optimierung dieser Systeme ist somit zwingend erforderlich, um Kosten zu senken und mit wachsenden Datenbeständen umgehen zu können.
Grundsätzlich stehen zwei Ziele im Fokus einer Optimierung - CPU-Nutzung und Laufzeit. IBM berechnet für die Nutzung eines Mainframes Lizenzgebühren, dessen Höhe sich nach einer individuellen Vereinbarung zwischen dem Nutzer und IBM richtet. Im Allgemeinen jedoch stellt die CPU-Nutzung einen wesentlicher Einflussfaktor dafür dar, weshalb eine Optimierung dieser unmittelbare finanzielle Vorteile für den Betreiber erzielt. Verringerte Laufzeiten verkürzen die Wartezeiten für Benutzer bei interaktiven Anwendungen sowie die Zeitfenster, die zur Bearbeitung von Batchläufen notwendig sind.

Aktuell existieren zwar Werkzeuge zur Messung der Performance von Anwendungen auf Mainframes, allerdings gestaltet sich die Auswertung ihrer Ergebnisse schwierig. Performanceexperten, die damit arbeiten sind gezwungen, manuell die notwendigen Informationen zu extrahieren. Abhängig von der Erfahrung dieser Experten werden gewisse Probleme möglicherweise garnicht erkannt. Eine automatische Auswertung der Reports von solchen Messungen soll helfen, die notwendigen Informationen zu erfassen und bereits gewonnenes Expertenwissen in Form von Anti-Pattern einfließen zu lassen. Dazu werden die Reports eines solchen Werkzeuges eingelesen und in ein Meta-Modell überführt. Auf diesem Modell wird anschließend eine automatische Suche nach definierten Anti-Pattern vorgenommen. Performanceexperten sollen dadurch unterstützt werden, die für sie relevanten Informationen zu bekommen und automatisch auf mögliche Optimierungspotentiale hingewiesen werden.
Dazu werden in Abschnitt 2 dieses Artikels die derzeitigen Verfahren und deren Probleme erläutert. Darauf aufbauend wird in Abschnitt 3 das eingesetzte Meta-Modell zur Speicherung von Performance-Daten und beispielhaft einige Anti-Pattern und deren Erkennungsverfahren vorgestellt. Der Abschnitt 4 fasst die Ergebnisse dieses Artikels noch einmal zusammen und gibt einen Ausblick auf weitergehende Forschungsthemen.

2 Derzeitiger Stand der Analyseverfahren

Grundsätzlich unterscheidet man zwei verschiedene Ansätze zur Analyse des Laufzeitverhaltens von Anwendungen: Statische Analyseverfahren und dynamische Analyseverfahren. Die folgenden beiden Unterkapitel beschreiben diese Verfahren mehr im Detail.

2.1 Statische Analyseverfahren

Statische Verfahren untersuchen die zu prüfende Software ohne diese auszuführen. Sie arbeiten in der Regel auf Basis des Sourcecodes oder Objektcodes, aber auch auf höheren Abstraktionsschichten, wie z.B. auf Modellen bei MDD Ansätzen. Statische Analyse-Werkzeuge überführen den Code des zu untersuchenden Programms in ein Modell, welches die möglichen Programmzustände und deren Übergänge repräsentiert. Auf diesem Modell werden anschließend Analysen durchgeführt. Das gebildete Modell umfasst alle möglichen Zustandsübergänge [Ern03], zur tatsächlichen Ausführung des Programms werden jedoch in Abhängigkeit der Eingabe nur bestimmte Zustandsübergängen wirklich ausgeführt (zum Beispiel durch bedingte Anweisungen). Ausführungspfade für konkrete Eingaben lassen sich aus diesem allgemeinen Modell ableiten. Da die Eingaben im Allgemeinen jedoch nicht bekannt sind, müssen Verfahren entwickelt werden, um die Wahrscheinlichkeit für die Wahl eines bestimmten Pfades abzuschätzen.
Ein großer Vorteil statischer Analyseverfahren ist, dass alle potentiellen Ausführungspfade untersucht werden können, da diese aus dem gebildeten Modell abgeleitet werden können. Da die statische Analyse ohne tatsächliche Ausführung des Programms durchgeführt wird, müssen zudem keine Abhängigkeiten zu bestimmten Hardware-Plattformen, Betriebssys-

temen oder externen Programmen erfüllt werden.
Dieser Vorteil erweist sich jedoch gleichzeitig als Nachteil bei der Messung von Performancedaten. Da die Analyse wie beschrieben umgebungsunabhängig stattfinden, können auch keine umgebungsspezifischen Performanceeinflüsse detektiert werden.
Hauptnachteil statischer Analyseverfahren ist jedoch, dass eine Berücksichtigung des dynamischen Verhaltens nicht erfolgen kann. So können beispielsweise die Anzahl an Schleifendurchläufen oder die Ausführungen bedingter Anweisungen nicht ermittelt werden, sondern müssen wie bereits erwähnt mittels besonderer Verfahren näherungsweise abgeschätzt werden.

2.2 Dynamische Analyseverfahren

Bei diesen Verfahren wird auch das dynamische Verhalten der untersuchten Programme berücksichtigt. Die Programme werden dafür zur Ausführung gebracht, die notwendigen Informationen während dessen entweder von zusätzlichem Code, der mit dem zu untersuchenden Programm ausgeführt wird (Messcode) geliefert oder von externen Messprogrammen „beobachtet“ (Sampling). Der Messcode kann beispielsweise einen Zähler definieren und am Anfang des Rumpfes einer bestimmte Funktion diesen inkrementieren, um die Anzahl der Aufrufe dieser Funktion zu zählen. Werkzeuge, die solch dynamische Analysen durchführen nennt man „Profiler“ und deren Anwendung „Profiling“. Unter Instrumentierung versteht man die Einbringung von zusätzlichem Messcode für die dynamische Analyse, welche

- manuell durch den Programmierer
- durch den Compiler
- zur Programmausführung

erfolgen kann. Eine manuelle Instrumentierung des Quellcodes eignet sich, wenn lediglich einzelne Abschnitte, beispielsweise einzelne Funktionen, gemessen werden sollen. Eine Messung größerer Programmabschnitte oder gar ganzer Programme sowie eine feingranulare Messung wird schnell sehr aufwändig, da viele solche Messbefehle eingefügt werden müssen. Zudem wird die Lesbarkeit des Codes durch die zusätzlichen Anweisungen stark beeinträchtigt. Aus diesem Grund werden die Instrumentierungen häufig automatisiert vom Compiler eingefügt. Der Quelltext bleibt dadurch unverändert und eine hohe Granularität kann einfach erreicht werden. Eine weitere Möglichkeit besteht darin, den Messcode erst zur Programmausführung einzufügen. Dies kann vor dem Laden des Programms in den Speicher oder bei Ausführung in einer virtuellen Maschine durch diese Maschine erfolgen. Dynamische Analyseverfahren können ebenfalls danach eingeteilt werden, von welcher Ebene ausgehend die Instrumentierung erfolgt [Net04]:

- Source analysis
  Analyse ausgehend vom Source-Code. Abhängig von der verwendeten Programmiersprache, unabhängig von der Zielplattform (Architektur und Betriebssystem)

- Binary analysis
  Analyse ausgehend von Maschinen-Code. Unabhängig von der verwendeten Programmiersprache, abhängig von der Zielplattform

Nicholas Nethercote beschreibt in seiner Dissertation [Net04] die Software „Valgrind“ [1], welche dynamische Instrumentierung zur Laufzeit verwendet.
Vorteile eines Profilings sind dessen Genauigkeit [Ern03] und die Menge der erfassbaren Informationen. Je nach verwendeter Profiling-Methode können neben Laufzeiten beispielsweise auch Call-Stacks oder Aufrufparameter von Funktionen aufgezeichnet werden. Ungenauigkeiten, die bei einer statischen Analyse durch Näherungen und Annahmen über die Zielplattform entstehen, treten bei dynamischen Verfahren nicht auf, da ein dynamisch analysiertes Programm direkt auf der Zielplattform zur Ausführung kommt. Effekte, die beispielsweise durch I/O Latenzen oder Nebenläufigkeiten entstehen, gehen somit ebenfalls in die Messung ein.
Nachteil dynamischer Verfahren ist, dass lediglich einzelne Programmausführungen gemessen werden, also Ausführungen mit einer konkreten Eingabe und einer konkreten Menge an Randbedingungen. Die Messung eines dynamischen Verfahrens kann also keine allgemeine Aussage für alle möglichen Eingaben treffen, ebenso wenig wie für alle möglichen Rahmenbedingungen (z.B. Auslastung des Systems während der Programmausführung).
Die Einbringung von zusätzlichem Code führt selbstverständlich zu einer Verlangsamung der Anwendung. Ein weiterer Nachteil bei der manuellen Instrumentierung und der Instrumentierung durch den Compiler besteht darin, dass das zu untersuchende Programm neu übersetzt werden muss. Diese Eigenschaft macht die beiden genannten Verfahren ungeeignet für die Messung von produktiven Anwendungen, da mit einem neuen Übersetzen auch ein erneutes Deployment nötig wäre. Eine Instrumentierung zur Laufzeit behebt dieses Problem, bringt jedoch weiteren Overhead mit sich.
Ein weiteres Profiling-Verfahren ist das sogenannte „Sampling“, bei dem periodisch von einem eigenständigen Messprogramm der aktuelle Ausführungszustand einer zu messenden Anwendung abgefragt wird. Die gemessene Anwendung liegt dabei unverändert vor, eine Neuübersetzung entfällt vollständig, weshalb mit dem Sampling-Verfahren auch produktive Anwendungen gemessen werden können. Durch das Sampling ist die Genauigkeit der Messung nicht so hoch wie bei einem Profiling mittels Instrumentierung. Jegliche Programmausführung, die zwischen zwei Samples geschieht, wird nicht erfasst. Auf eine ausreichende Abtastrate ist deshalb zu achten, wobei eine höhere Rate zwar die Genauigkeit, aber auch den Messaufwand erhöht. Da ein Sample lediglich den aktuellen Zustand erfasst und nicht erkennt, wie das Programm in diesen Zustand gelangt ist, werden keine Call-Stacks erfasst.
Die Software „Strobe“ von Compuware [2] führt ein Profiling auf Mainframe-Systemen mittels Sampling durch. Die von Strobe generierten Reports werden für den unter Abschnitt 3 beschriebenen Ansatz eingesetzt.

[1] http://www.valgrind.org, zuletzt aufgerufen am 26.11.2012
[2] http://www.compuware.com, zuletzt aufgerufen am 26.11.2012

2.3 Auswertung

Grundsätzlich liefert ein Profiling von großen Anwendungen eine Vielzahl an Informationen, die nur mittels Filtern, Aggregationen und visuellen Darstellungen menschlich auswertbar sind. [Par08] beschreibt diese Informationsflut im Rahmen der Analyse von JEE Systemen. Strobe beispielsweise erzeugt ein Textdokument, dessen Inhalt häufig mehr als 10000 Zeilen umfasst. Die benötigten Informationen um konkrete Performance-Probleme zu untersuchen sind auf verschiedene Abschnitte (Sections) innerhalb des gesamten Dokuments verteilt. Abbildung 1 zeigt beispielhaft die in einem Strobe-Report enthaltenen Sections.

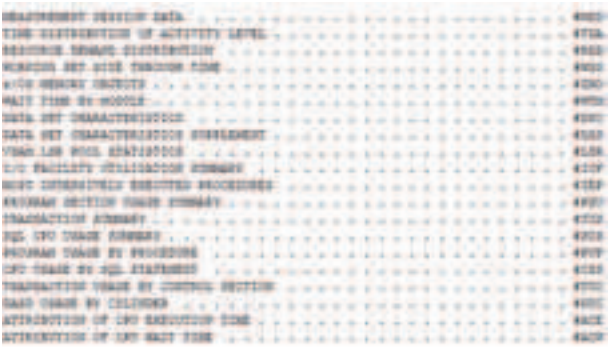

Abbildung 1: Sections innerhalb eines Strobe-Reports

Für einige Systemmodule weist Strobe die CPU-Nutzung aufgeschlüsselt nach Aufrufern aus. Aufrufe von Systemmodule werden stets (direkt oder indirekt) durch Anwendungsprogramme ausgelöst. Somit können Situationen auftreten, in denen ein Anwendungsmodul selbst kaum CPU-Ressourcen verbraucht, jedoch Systemmodule aufruft, welche in Summe dann einen nennenswerten Verbrauch verursachen. Solche Szenarien sind mittels manueller Analyse nur sehr schwer zu erkennen, da ggf. die einzelnen Systemmodule selbst ebenfalls keinen auffällig hohen Verbrauch aufweisen. Abbildung 2 zeigt einen Ausschnitt aus einem solchen Report. Das Modul „ABC530“ verursacht in diesem Beispiel 0,3% des gesamten CPU-Verbrauchs. Ebenso kann man ablesen, in welchem Bereich des Codes wieviel Verbrauch verursacht wurde.

3 Ansatz zur Performance-Analyse auf Mainframe-Systemen

Die von Strobe erzeugten Textdokumente haben eine - wenn auch ggf. versionsabhängige - klar definierte Struktur, was ein automatisches Einlesen ermöglicht. Die dadurch ermittelten Informationen werden in ein Meta-Modell überführt und in einer Datenbank für nachfolgende Analysen persistiert. Anschließend werden Beispiele für häufig auftretende Anti-Pattern in Unternehmensanwendungen vorgestellt und erläutert, wie diese auf Basis

Abbildung 2: Auszug aus einem Strobe-Report

des beschriebenen Datenmodells detektiert werden können.

3.1 Meta-Modell zur Speicherung von Performance-Daten

Die extrahierten Informationen werden in ein Meta-Modell (Abb. 3) überführt, welches später sowohl zur Erzeugung von für manuelle Interpretationen optimierten Darstellungen, als auch zur automatischen Interpretation genutzt wird.

Kern des Modells ist die Klasse *ProgramPart*, welche die Generalisierung der verschiedenen Teile eines Programms repräsentiert. Dabei findet das Composite-Pattern nach Gamma et al. [GHJV94] Verwendung. Für jede Instanz von *ProgramPart* bzw. dessen Subklassen wird die prozentuale CPU-Nutzung und die verursachte Wartezeit gespeichert; der Zugriff auf diese Informationen ist, durch die Vererbung bedingt, für alle Subklassen einheitlich. Diese hierarchische Struktur erlaubt es über einen Drilldown den gewünschten Detaillierungsgrad einzustellen und somit die Menge der Informationen erfassbar darzustellen. Die Klasse *Program* repräsentiert das gesamte gemessene Programm. Die von Strobe für das gesamte Programm gelieferten „ServiceUnits“ stellen einen maschinenunabhängigen Wert für die CPU-Nutzung dar. Dieser Wert ist nicht zu verwechseln mit den von IBM zur Abrechnung der Lizenzkosten genutzten Einheiten, die ebenfalls „ServiceUnits“ genannt werden. IBM’s ServiceUnits sind nicht maschinenunabhängig. Programme bestehen aus mindestens 1 Modul, jedes Modul wiederum kann verschiedene Sections (Control-Sections) beinhalten. Dieser Unterteilung spiegelt sich in den entsprechenden Klassen (*Module*, *Section*) im Meta-Modell wider. Das Sampling von Strobe liefert Ausführungszeiten noch detaillierter unterteilt in hexadezimale Intervalle innerhalb einer Section. Deshalb stellt das Meta-Modell noch die Klasse *SectionInterval* zur Aufnahme der Start- und Endadresse eines solchen Intervalls bereit. Eine Sonderform der Klasse *Module* stellt das *SystemModule*

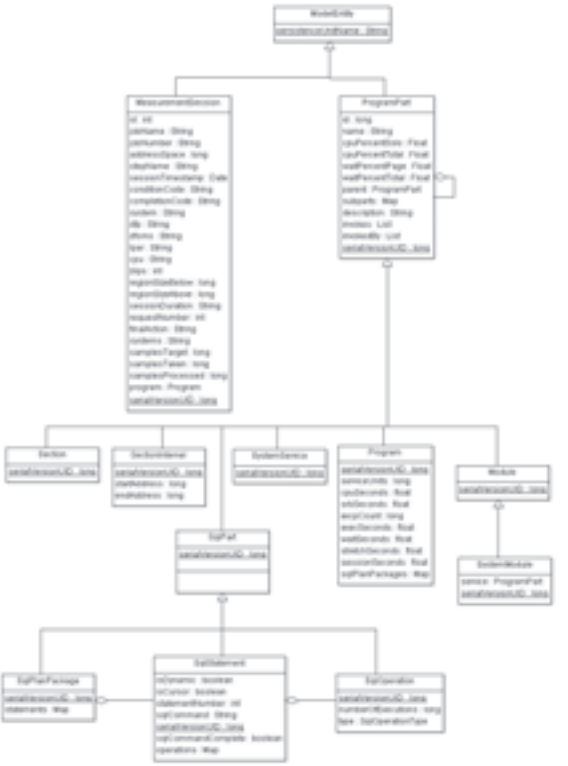

Abbildung 3: Meta-Modell zur Speicherung von Performance-Daten

dar. Ein *SystemModule* ist einem bestimmten Systemdienst (*SystemService*) zugeordnet. Für SQL-Ausführungen auf DB2 liefert Strobe den Namen des Plans (bzw. des Packages) und die ausgeführten Statements mit deren SQL-Text. Je nach Einstellung der Messung ist der SQL-Text jedoch nicht vollständig dargestellt (Standard-Einstellung 300 Zeichen). Die Anzahl der Ausführungen wird separat für jede Operation eines Statements erfasst. Eine Operation ist dabei ein Open, Fetch oder Update. Abbildung 4 zeigt dabei die Darstellung von SQL-Statements in Strobe-Reports.

SQL-Statements können statisch oder dynamisch sein. Statische Statements sind vorcompiliert, deren Zugriffspfad wird bei der Erstellung (Binden zu einem Plan) ermittelt. Dynamische Statements hingegen werden erst zur Laufzeit interpretiert und optimiert. Diese Unterscheidung ist für Performance-Betrachtungen wichtig, da statische Statements zwar den zusätzlichen Aufwand zur Interpretation des SQL-Textes und Erstellung des Zugriffspfads sparen, jedoch den Zugriffspfad nicht auf die zur Laufzeit aktuelle Datenverteilung und dem Statement übergebenen Parameterausprägungen anpassen können. SQL-

Abbildung 4: CPU-Nutzung durch SQL-Statements im Strobe-Report

Statements können einen Cursor verwenden oder ohne Cursor ausgeführt werden. Diese Information wird ebenfalls im Meta-Modell erfasst.
Die Messung betreffende, allgemeine Informationen werden mittels der Klasse *MeasurementSession* verwaltet. Dazu zählen beispielsweise die Anzahl der genommenen Samples, die Sampling-Rate, die eingesetzten Systeme und deren Version usw. Die Klasse *ModelEntity* generalisiert lediglich die Verwaltung der Subklassen wie z.B. deren Persistierung und trägt keine semantische Bedeutung. Der aus Instanzen dieser Klassen aufgebaute Baum mit Performance-Daten kann traversiert werden, um benötigte Informationen, wie z.B. den aggregierten Anteil der CPU-Nutzung aller *UPDATE* Sql-Statements zu extrahieren. Für die Traversierung bietet sich ein Visitor [GHJV94] an.

3.2 Anti-Pattern Suche

Viele Performance-Probleme spiegeln sich in Form von gewissen Pattern in den Profiling-Reports wider. Diese Pattern können auch automatisch auf Basis der eingelesenen Daten gesucht werden. Einen vergleichbaren Ansatz auf Basis von JEE Anwendungen verfolgt auch [Par08]. Nachfolgend werden zwei Beispiele für Anti-Pattern beschrieben, die in Strobe-Messungen von Cobol Programmen mit DB2 Datenbank - einer typischen Konfiguration für Legacy-Anwendungen auf Mainframes - in produktiven Systemen bereits beobachtet wurden.

3.2.1 Einzelsatz-Fetch mit Cursor

Ein Cursor ist eine Datenstruktur, die verwendet wird, um die Ergebnismenge einer Datenbankabfrage zu durchlaufen. Häufig werden bei Datenbankabfragen, welche die Rückgabe

von höchstens einem Wert sicherstellen, Cursor eingesetzt anstatt unmittelbar den gesuchten Wert abzufragen. In diesem Fall ist der Aufwand zur Erzeugung, Nutzung und Freigabe des Cursors vermeidbar. Die Zusicherung, dass nicht mehr als ein Datensatz in der Ergebnismenge auftritt ist z.B. bei Abfragen auf Primärschlüssel oder bei Nutzung von Aggregatfunktionen ohne GROUP BY Klausel gewährleistet.
In den Performance-Daten erkennt man dieses Muster am Verhältnis von Open- zu Fetch-Ausführungen für ein SQL-Statement:

$$N_{Fetch} \leq N_{Open}, bzw. \frac{N_{Fetch}}{N_{Open}} \leq 1 \quad (1)$$

Diese Formel beschreibt eine notwendige, aber nicht hinreichende Bedingung für einen Einzelsatz-Fetch mit Cursor. Eine weitere Untersuchung des eingesetzten SQL-Statements und ggf. der DDL hat in einem weitern Schritt zu erfolgen. Zum Auffinden möglicher Kandidaten für dieses Anti-Pattern, wird für alle im Datenmodell abgespeicherten SQL-Statements das oben beschriebene Verhältnis berechnet. Liegt dieses zwischen einem konfigurierbaren Schwellwert und 1, so wird das betrachtete Statement als Kandidat für dieses Anti-Pattern betrachtet. Der Schwellwert dient dazu, ggf. auftretende Abfragen, die keine Records liefern, ebenfalls zu akzeptieren (z.B. wenn ein Schlüssel gesucht wird, der nicht exisitert).

3.2.2 Cobol Library

Einige Anweisungen in Cobol veranlassen den Compiler dazu, Routinen aus der Cobol Library aufzurufen. Die Aufrufe der einzelnen Routinen finden sich in den Strobe-Reports wieder. Kann einem solchen Aufruf eine hohe CPU-Nutzung zugeordnet werden, deutet dies auf eine potentielle Optimierungsmöglichkeit durch Ersatz des verursachenden Aufrufs hin.
Beispiel hierfür sind der „variable length move“, welcher verwendet wird, wenn eine überlappende Struktur kopiert wird oder „Inspect“ zur Bearbeitung von Strings. Letzterer Aufruf ist nach [She02] mit einer einfachen Schleife und Substring-Operationen deutlich performanter. In typischerweise auf Mainframes ausgeführten Unternehmensanwendungen werden in der Regel keine Fliesskommaoperationen mit doppelter Genauigkeit benötigt. Auch hierfür erzeugt der Cobol Compiler Bibliotheksaufrufe, die sich leicht durch eine Änderung der verwendeten Datentypen eliminieren lassen.
Abbildung 5 stellt einen Auszug aus einem Strobe-Report eines produktiven Systems dar. Gemessen wurde ein Batch, der 67,39 % CPU-Nutzung alleine für Divisionen und Multiplikationen mit doppelter Genauigkeit verursacht.

Eine Nutzung solcher „verdächtigen“ Bibliotheksroutinen kann auf dem vorgestellten Datenmodell einfach gefunden werden, indem dieses traversiert und nach dem betreffenden Modul (im Beispiel „IGZCPAC“), ggf. auch nach einer bestimmten Section gesucht wird. Ein Abgleich der dadurch verursachten CPU-Nutzung mit einem Schwellwert kann dieses Pattern weiter einschränken und nur bei signifikanter Auswirkung entsprechend negativ klassifizieren.

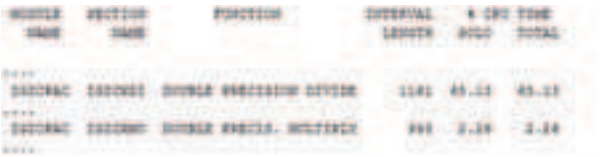

Abbildung 5: Intensive Nutzung der Cobol Library im Strobe-Report

4 Zusammenfassung und Ausblick

Die alleinige Auswertung der Informationen aus den Strobe-Reports bringt bereits einige potentielle Probleme ans Licht. [She02] führt noch weitere Anti-Pattern auf, die mit dem beschriebenen System gesucht werden können. Die Existenz einzelner SQL-Statements, die pro Ausführung viel CPU-Nutzung relativ zu anderen Statements erzeugen, ist ebenfalls eine intessante Fragestellung, die damit beantwortet werden könnte. Eine aufbereitete Darstellung für Performanceexperten könnte eine explorative Analyse (Drilldown etc.) der Daten ermöglichen. Die Modulbezeichnungen betrieblicher Anwendungen unterliegen häufig gewissen Namensschemata, welche bestimmte Funktionalitäten gruppieren. Eine Aggregation an Hand dieses anwendungsspezifischen Namensschemas liefert eine weitere mögliche Sicht auf die Daten. Eine Anreicherung bzw. Verknüpfung der vorliegenden Informationen mit weiteren Daten könnte weitergehende Performance-Analysen ermöglichen. Denkbar wäre eine Verbindung mit einer DDL-Datei für Datenbankzugriffe. Hier könnte man beispielsweise bei Einzelsatz-Fetch mittels Cursor prüfen, ob die Selektion tatsächlich auf Schlüsselattribute erfolgt. Ebenso könnten die oben beschriebenen Informationen verwendet werden, um statische Analysverfahren zu optimieren, indem Gewichtungen für die Kosten aus den Ausführungszeiten einzelner Module oder Programmabschnitte abgeleitet werden.

Literatur

[Ern03] Michael D. Ernst. Static and dynamic analysis: synergy and duality. *Proceedings of the ICSE Workshop on Dynamic Analysis (WODA 2003)*, Seiten 25–28, 2003.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*, Jgg. 1. Addison-Wesley Longman, Oktober 1994.

[Net04] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation*. Dissertation, University of Cambridge, 11 2004.

[Par08] John Parson, Trevor Murphy. Detecting Performance Antipatterns in Component Based Enterprise Systems. *Journal of Object Technology*, 7(3):55–90, März 2008. Sucht Anti-Patterns in JEE Anwendungen - Profiling mit Ïnstrumentierung"(Proxy).

[She02] Tony Shediak. Performance Tuning Mainframe Applications "Without trying too hard". 2002.