Forschungszentrum Jülich

Zentralinstitut für Angewandte Mathematik

Basiswerkzeuge zur automatischen Auswertung von Apprentice-Leistungsdaten

Alexander Lucas

Jül-3652

Basiswerkzeuge zur automatischen Auswertung von Apprentice-Leistungsdaten

Alexander Lucas

Berichte des Forschungszentrums Jülich ; 3652
ISSN 0944-2952
Zentralinstitut für Angewandte Mathematik Jül-3652

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek
D-52425 Jülich · Bundesrepublik Deutschland
☎ 02461/61-6102 · Telefax: 02461/61-6103 · e-mail: zb-publikation@fz-juelich.de

Kurzfassung

Die Entwicklung effizienter Anwendungen für Parallelrechner stellt durch die komplexe Hardware der Rechner eine große Herausforderung bei der Softwareerstellung dar. Konventionelle Werkzeuge zur Unterstützung der Optimierung paralleler Programme zeichnen die Leistungswerte einzelner Programmteile auf und visualisieren sie durch graphische Anzeigen. Zur Suche nach Leistungsengpässen sind jedoch umfangreiche Kenntnisse und Erfahrungen in der Software-Optimierung erforderlich.

Diese Arbeit beschreibt eine exemplarische Implementierung von Basiswerkzeugen, die im Rahmen des am Zentralinstitut für Angewandte Mathematik begonnenen KOJAK-Projektes (Kit for Objective Judgement and Automatic Knowledge-based detection of Bottlenecks) eine Grundlage für die Realisierung einer automatischen Leistungsanalyse auf einer CRAY T3E bilden können.

Diese Werkzeuge umfassen eine Erweiterung von Apprentice um eine Schnittstelle zum Export der Daten und eine Datenbank zur Aufnahme der exportierten Leistungsdaten. Durch ein Java-Applet werden der Zugriff auf die Datenbank demonstriert und erste Schritte einer automatischen Analyse implementiert.

Abstract

The development of efficient applications in parallel computing is due to the complex parallel hardware architecture a great challenge to software designers. Conventional tools assisting optimization of parallel applications collect performance data of single program parts and visualize them graphically. However, detecting performace bottlenecks requires extensive knowledge and experience in software optimization.

This thesis describes an exemplary implementation of tools supporting automatic performance analysis on a CRAY T3E within the framework of the KOJAK project (Kit for Objective Judgement and Automatic Knowledge-based detection of bottlenecks) at the Central Institute for Applied Mathematics.

These tools consist of an extension of Apprentice by an export interface. Additionally, a database storing performance data and information about programs and their different versions is designed. A Java applet demonstrates the access to the database and implements the first steps of an automatic analysis.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Leistungssteigerung gegenüber traditionellen sequentiellen Rechnersystemen ist ein primärer Grund für die Entwicklung und Verwendung von Programmen auf Parallelrechnern. Solche Anwendungen werden daher bevorzugt für performancekritische Problemstellungen eingesetzt, die ein hohes Maß an Rechenkapazität erfordern. Parallele Systeme verlieren also ihre Vorteile, wenn sie nicht effizient genutzt werden.

Aufgrund der Art und der Vielfalt der Hardwarearchitekturen von parallelen Systemen ist eine Vorhersage der Performance jedoch schwieriger als bei herkömmlichen Rechnern. Die Programmentwicklung hat somit einen experimentellen Charakter und bildet einen zyklischen, inkrementellen Prozeß mit dem folgenden typischen Ablauf: Ein erster Programmentwurf weist eine nicht zufriedenstellende Performance auf. Mit Hilfe eines Analysewerkzeugs werden dann die relevanten Engpässe eingegrenzt, bevor der Programmierer in einem weiteren Schritt versucht, das Programm zu restrukturieren oder zu optimieren. Dadurch wird ein großer Teil des Programmieraufwandes beim *Performance Debugging*, also beim Finden und Eliminieren leistungsmindernder Programmteile, die durch Designfehler oder Portierung verursacht werden, beansprucht. Das Performance Debugging nimmt somit eine zentrale Stellung in der parallelen Programmierung ein.

Die üblichen Werkzeuge zur Analyse versuchen, Programmteile oder Ereignisspuren durch graphische Anzeigen zu visualisieren. Zur Interpretation dieser Informationen muß ein Programmierer daher neben dem problemspezifischen Wissen auch über Erfahrung im Umgang mit Analysewerkzeugen und mit potentiellen Engpässen verfügen.

Da die Analysesysteme sich auf den verschiedenen Plattformen unterscheiden, mangelt es ihnen meist an Akzeptanz und sie werden gar nicht oder nur wenig genutzt, wodurch die Analyse zusätzlich erschwert wird. Einfache Werkzeuge, die auf allen Systemen verfügbar sind, wie z. B. das UNIX-Kommando `gprof`, weisen hingegen eine unkomfortable Bedienung auf und liefern eine unzureichende Anzahl von Laufzeitdaten.

Im Rahmen des KOJAK-Projektes [10] wird in dieser Arbeit ein Instrumentarium entwickelt, das eine automatische Performanceanalyse auf einer CRAY T3E ermöglicht. Dabei werden die vom CRAY-Werkzeug *Apprentice* ermittelten Laufzeitdaten als Datenbasis herangezogen. Zu diesem Zweck wird Apprentice um eine dem „Report“ ähnliche Funktion erweitert, bei der dann alle zur weiteren Analyse verwendbaren Daten in einem sinnvollen Format ausgegeben werden können.

Den Kern der Arbeit bildet eine relationale Datenbank, in der die von Apprentice ausgegebenen Daten gesammelt und für eine weitere Analyse bereitgestellt werden. Durch ein Java-Applet werden dann der Zugriff auf diese Datenbank demonstriert und erste Komponenten und Techniken zur automatischen Leistungsanalyse exemplarisch realisiert.

Im Anschluß an diese Einleitung werden in Kapitel 2 eine Einordnung des Themas und das KOJAK-Projekt vorgestellt. Nachfolgend wird in Kapitel 3 auf die internen Strukturen, die Arbeitsweise sowie die zur Verfügung stehenden Daten von Apprentice eingegangen, bevor in Kapitel 4 der Entwurf der Datenbank beschrieben wird, die sowohl die Leistungsdaten als auch Programm- und Versionsinformationen sowie verschiedene Testszenarien aufnimmt. Nach einer Darstellung der Datenbank-Schnittstellen zur Client/Server-Programmierung in Kapitel 5 werden in Kapitel 6 in einem letzten Schritt die Funktionsweise sowie die Implementierungsaspekte des entwickelten Java-Applets vorgestellt. Abschließend folgt in Kapitel 7 eine Zusammenfassung.

Kapitel 2

Überblick und Einordnung

2.1 CRAY T3E, Message Passing, Werkzeuge

2.1.1 Architektur der Cray T3E

Die CRAY T3E ist ein massiv paralleles System, das für den Einsatz im technischen und wissenschaftlichen Bereich entwickelt wurde. Aufgrund der skalierbaren Performance kann die Rechenleistung, die einer Anwendung zur Verfügung gestellt wird, durch die Auswahl der Anzahl von Prozessoren an die erforderliche Größe angepaßt werden.

Der im Rahmen dieser Arbeit eingesetzte Rechner ist mit 16 bis 2048 DEC Alpha 21164 RISC-Prozessoren mit 600 MFLOPS ausgestattet. Die Prozessoren verfügen über Integer- und Fließkommapipelines, 32 Integer-Register sowie 32 Fließkommaregister.

Durch die verteilte Speicherarchitektur hat jeder Prozessor nur auf seinen lokalen Speicher direkten Zugriff. Ein globaler Speicherzugriff ist nur über spezielle Funktionen möglich und mit hohen Kosten verbunden. Die Prozessoren sind in einem dreidimensionalen Torus angeordnet und können miteinander über ein Netzwerk mit einer Bandbreite von bis zu 500 MByte/sec. kommunizieren.

Der DEC Alpha 21164 besitzt keine Instruktion zur Integer-Division. Stattdessen wird zur Durchführung eine Bibliotheksroutine aufgerufen. Sollte nicht der volle Wertebereich beansprucht werden, so kann es bei einer großen Zahl von Integer-Divisionen sinnvoll sein, Fließkomma-Arithmetik zu verwenden.

Bei der parallelen Programmierung kommen die Programmiermodelle MPI (Message Passing Interface), PVM (Parallel Virtual Machine) und Shared Memory zum Einsatz. Da MPI sich zum weitestverbreiteten Standard entwickelt hat, wird im folgenden nur dieses Modell betrachtet.

2.1.2 MPI

MPI ist eine standardisierte Message-Passing-Bibliothek, die Funktionen zum Versenden und Empfangen von Nachrichten zur Verfügung stellt und dadurch eine Kommunikation unterschiedlicher Prozessoren ermöglicht. Mit ihrer Anbindung zu den Programmiersprachen C/C++ und Fortran bietet die Bibliothek eine Unterstützung bei der Implementierung portabler paralleler Programme.

Obwohl MPI in der neuesten Version 2.0 über 200 Funktionen beinhaltet, lassen sich bereits mit Hilfe der folgenden fünf Basisfunktionen viele parallele Programme realisieren und das Prinzip von MPI illustrieren (vgl. [11, 17]):

- `MPI_Init(argc, argv)` initialisiert die Kommunikation zwischen Prozessoren.
- `MPI_Send(buf, count, datatype, dest, tag, com)` sendet `count` Werte des Typs `datatype` an der Speicheradresse `buf` an den Prozessor mit der logischen Nummer `dest`. Dabei wird die Nachricht mit einer eindeutigen Kennung `tag` versehen. Der Parameter `com` dient zur Gruppierung von Prozessoren.
- `MPI_Recv(buf, count, datatype, source, tag, com, status)` empfängt diese versendeten Werte. Die zusätzliche Variable `status` gibt Auskunft über die Eigenschaften der Nachricht.
- `MPI_Bcast(buf, count, datatype, root, com)` überträgt eine Nachricht vom Prozessor `root` an alle anderen Prozessoren der Gruppe `com`. Hierzu muß `root` ebenfalls Mitglied von `com` sein. Diese Funktion wird sowohl vom sendenden als auch von den empfangenden Prozessoren in gleicher Weise aufgerufen.
- `MPI_Finalize()` beendet die Kommunikation.

Bei allen beschriebenen Funktionen handelt es sich auf der CRAY T3E um synchrone Kommunikationsfunktionen, die die beteiligten Prozessoren blockieren, bis die Gegenstelle(n) zum Nachrichtenaustausch bereit ist bzw. sind. MPI enthält auch asynchrone Sende- und Empfangsfunktionen, auf deren Beschreibung jedoch hier verzichtet wird.

2.1.3 Werkzeuge

Zur Performanceanalyse stehen dem Anwender auf der CRAY T3E mit VAMPIR, PAT und Apprentice drei Werkzeuge zur Verfügung.

VAMPIR dient der Analyse von Ereignisspuren von Message-Passing-Programmen. Dieses Werkzeug erfordert bei C-Programmen eine manuelle Instrumentierung, während für Fortran 77 ein zusätzliches Werkzeug existiert, das die Instrumentierung automatisch vornimmt. In einer graphischen Benutzeroberfläche zeigt VAMPIR Zeitlinien-Diagramme, Parallelismus sowie eine Vielzahl statistischer Auswertungen

an. Der Benutzer kann dabei die Skalierung der Zeitachse frei wählen und so das dargestellte Zeitintervall verfeinern.

Durch die Komplexität und Vielfalt ist VAMPIR jedoch nicht einfach zu bedienen: Gerade bei Programmen mit längerer Laufzeit können die Diagramme sehr unübersichtlich werden, so daß zur genauen Analyse kleinere Intervalle gewählt werden müssen. Damit werden jedoch andere Informationen ausgeblendet und es besteht die Gefahr, daß Performanceprobleme übersehen werden.

PAT (Performance Analysis Tool) ist ein von CRAY bereitgestelltes Werkzeug, das auf der Technik des *Sampling* basiert. Hierbei ist keine Instrumentierung oder Unterstützung durch den Compiler erforderlich. PAT greift während der Laufzeit in das Programm ein, indem es in regelmäßigen Zeitabständen die Ausführung unterbricht und die Leistungsdaten mißt. Dadurch erhält der Anwender jedoch nur eine statistische Abschätzung der realen Daten. Hierbei werden ergänzend zu den herkömmlichen Meßverfahren auf Softwarebasis die Hardware-Performancecounter des DEC Alpha ausgenutzt.

Für eine detaillierte Analyse besteht weiterhin die Möglichkeit, Teile des Programms auf Objektcode-Ebene dennoch zu instrumentieren oder Ereignisspuren zu generieren.

Apprentice [7] ist das zweite CRAY-Standardwerkzeug für die T3E. Es wird nach der Ausführung des Programms gestartet und basiert sowohl auf statischen Daten, die bereits bei der Übersetzung generiert, als auch auf dynamischen Daten, die während der Ausführung gemessen werden. Dem Anwender werden reine Summeninformationen über Ausführungszeiten, Kommunikationszeiten und Zahl der Instruktionen, jedoch keine Ereignisspuren zur Verfügung gestellt.

Da sich die weiteren Betrachtungen auf dieses Werkzeug konzentrieren, wird es in Kapitel 3 näher beschrieben.

2.2 Literatur

2.2.1 Paradyn

Paradyn [18, 13] basiert auf dem Prinzip der *dynamischen Instrumentierung*. Zur Minimierung des Instrumentierungsaufwandes sowie zur Erzielung einer höheren Genauigkeit versucht Paradyn nur solche Programmteile zu instrumentieren, die für die auftretenden Performanceprobleme verantwortlich sind. Die Suche beginnt mit einer groben, wenig Overhead verursachenden Instrumentierung des gesamten Programms. Nachdem auf dieser höchsten Ebene das Problem eingekreist wurde, werden die kritischen Programmteile detaillierter instrumentiert, so daß die Fehlerquellen aufgespürt werden können.

Entgegen herkömmlicher Werkzeuge instrumentiert und analysiert Paradyn ein Programm während der Ausführung.

Die Analyse basiert auf dem W^3-Modell. Dieses führt ein Performanceproblem auf die drei Fragen „warum?“, „wo?“ und „wann?“ zurück:

- **Warum:** Paradyn repräsentiert potentielle Engpässe durch eine Kombination von Hypothesen mit Tests. Grundlage ist eine relativ kleine Menge von Hypothesen, die die meisten Performanceprobleme aufdecken können. Der Test einer solchen Hypothese wird durch den Vergleich eines gemessenen Wertes mit einem Schwellenwert durchgeführt. Hypothesen werden hierarchisch dargestellt.
- **Wo:** Die Suche nach der genauen Position des Problems wird über verschiedene Resourcen-Hierarchien realisiert. Typische Resourcen sind z. B. Synchronisationsobjekte, Prozesse oder Quelltexte. Jede dieser Hierarchien kann unabhängig voneinander durchlaufen werden.
- **Wann:** Die Ausführung eines Programms wird als eine Folge von unterschiedlichen Phasen repräsentiert, die jeweils ohne Einfluß auf andere Phasen getestet werden können. Dadurch kann der Zeitpunkt bestimmt werden, an dem ein Problem auftritt.

Die Analyse erfolgt durch sukzessives Verfeinern der Fokusse auf den einzelnen Achsen. Paradyn kann eine vollautomatische Suche durchführen, der Anwender hat jedoch die Möglichkeit, in die Auswahl der jeweiligen Verfeinerungen einzugreifen.

Der Nachteil dieses Verfahrens besteht darin, daß durch die Suche entlang der mehrdimensionalen Achsen ein sehr breiter Suchbaum entsteht. Darüber hinaus können durch die dynamische Instrumentierung Seiteneffekte und bei kleinen Programmen ein größerer Overhead als bei traditioneller Instrumentierung auftreten. Es sind sogar Konstellationen denkbar, in denen das Programm bereits beendet ist, bevor Paradyn in der Lage ist, die Probleme aufzuspüren. In einem solchen Fall müßte das Programm erneut gestartet werden, um die Suche zu vervollständigen.

Die Stärken von Paradyn liegen daher deutlich bei großen Programmen mit Laufzeiten von mehreren Stunden oder Tagen.

2.2.2 KAPPA-PI

Das von A. Espinosa und T. Margalef entwickelte KAPPA-PI [8, 9] (**K**nowledge-based **A**utomatic **P**arallel **P**rogram **A**nalyzer for **P**erformance **I**mprovement) basiert auf einer vordefinierten Liste von häufig auftretenden Performanceproblemen und ihren Ursachen. Das Analysewerkzeug führt ein „Pattern Matching“ zwischen den Trace-Daten des Programms und den bekannten Fehlern durch und versucht auf diesem Weg, Schwachstellen zu lokalisieren.

Die Trace-Dateien des externen Werkzeugs TapePVM [16], die nach ineffizienten Intervallen durchsucht werden, dienen als Grundlage für die Analyse, die erst nach

der Beendigung des Programms durchgeführt wird. Danach werden diese ineffizienten Programmteile den bekannten Fehlern zugeordnet. In einem letzten Schritt wird dann – ebenfalls über die Liste der bekannten Fehler und ihrer Merkmale – der Bezug zum Quelltext hergestellt.

Folgende Fehler werden von KAPPA-PI erkannt:

- **Verteilung der Prozesse**: Leerlaufende Prozessoren stehen wartenden Prozessen in arbeitenden Prozessoren gegenüber.
- **Blockierter Sender**: Ein blockierter Prozessor wartet auf die Nachricht eines anderen Prozessors. Dieser wartet jedoch wiederum auf einen dritten Prozessor, so daß sich eine Kommunikationsabhängigkeit von mindestens drei Prozessoren ergibt.
- **Mehrfaches Versenden** (*Multiple Output*): Hier wird das Versenden mehrerer Nachrichten an verschiedene Empfänger serialisiert. Daher muß der jeweilige Empfänger warten, bis das Versenden an alle seine Vorgänger abgeschlossen ist.
- **Lange Kommunikationszeiten**.
- **Master/Slave-Probleme**: Die Anzahl der Master bzw. Slave-Prozessoren ist schlecht gewählt.
- **Barrier-Probleme**.
- **Ausnutzung der Parallelität**: Die Anwendung liefert nicht genügend Daten für alle Prozessoren.

Die Arbeit an KAPPA-PI ist noch nicht fortgeschritten. Die Liste der erkannten Fehler ist noch unausgereift und soll in Zukunft erweiterbar gestaltet werden. Ebenso soll die Analyse auf weitere Programmiermodelle ausgedehnt werden, da zur Zeit nur PVM unterstützt wird. Ein Vergleich von verschiedenen Programmläufen oder ein Test auf Skalierbarkeit ist mit KAPPA-PI nicht möglich.

2.2.3 Poirot

Ähnlich wie KAPPA-PI führt Poirot [12] keine eigenen Messungen durch, sondern verwendet die bereits gemessenen Daten externer Analyseprogramme. Aufbauend auf dem Konzept der *heuristischen Klassifikation* [4], einem Ansatz aus dem Bereich der Expertensysteme, werden mittels einer *Abstraktion* die wesentlichen Werte aus den Performancedaten des *Ausführungsraums*, d. h. aus der Menge aller möglichen Programmläufe herausgefiltert und summiert. Im nachfolgenden heuristischen Vergleich versucht Poirot, die abstrahierten Eigenschaften einem bekannten Performanceproblem aus dem *Lösungsraum* zuzuordnen. Analog zu Paradyn werden Hypothesen hierarchisch dargestellt, so daß eine erste Lösung verfeinert werden kann.

Im Gegensatz zu anderen Systemen, die sich ebenfalls externer Werkzeuge bedienen, stellt Poirot weder ein integriertes System, das auf bestimmte Werkzeuge und Diagnosemethoden fixiert ist, noch ein Toolkit dar, das zwar die Steuerung des Zusammenspiels verschiedener Werkzeuge übernimmt, dem Anwender aber keine Hilfe bei der Diagnose bietet. Die Implementierung von Poirot gliedert sich in die zwei Grundelemente *Problem Solver* und *Environment Interface.*

Der Problem Solver ist ein wissensbasiertes System, das eine Menge von Diagnoseaufgaben enthält, denen jeweils eine Reihe von Aktionen zugeordnet sind, die zur Erfüllung der Aufgabe ausgeführt werden müssen. Diese Aktionen können entweder eine neue Diagnoseaufgabe generieren und somit die Aufgabe dynamisch erweitern oder durch das Environment Interface Transformationen durchführen, wie z. B. externe Programme anwenden oder den Anwender zu einer Eingabe auffordern. Poirot beendet seine Analyse, wenn alle Aufgaben erfüllt wurden.

Durch das Environment Interface ruft Poirot externe Werkzeuge nicht direkt auf, sondern generiert abstrakte Anweisungen, die von der Schnittstelle übersetzt werden. Somit wird eine hohe Anpassungsfähigkeit und Erweiterbarkeit erreicht.

Zu diesem Ansatz sind weder eine Realisierung des Designs noch weiterführende Erkenntnisse veröffentlicht worden.

2.2.4 Online Monitoring

Das Online-Monitoring-System [26] ermittelt die Performance-Informationen auf Grundlage von deklarativen Anfragen während der Laufzeit. Dabei werden *Sensoren* verwendet, die – ähnlich zur Instrumentierung – in den Programmcode eingefügt werden und Performancemessungen durchführen. Sensoren besitzen jedoch die Eigenschaft, aktivierbar und deaktivierbar zu sein. Die Aktivierung erfolgt automatisch durch Auswertung des Anfrageausdrucks. Die Analyse wird in fünf Schritten vollzogen:

1. **Konfiguration der Sensoren**
   Die Programmstellen, an denen die Sensoren, d. h. die Operationen zur Performancemessung, eingefügt werden sollen, werden ausgewählt. Zusätzlich wird entschieden, welche Informationen gemessen werden. Die Spezifikation der Sensoren erfolgt in einer eigenen Definitionssprache als eine Sammlung von einfachen Ereignissen und Intervallbeziehungen.

2. **Installation der Sensoren**
   Die zusätzlichen Operationen werden in den Programmcode eingefügt. Dieser Schritt wird automatisch ausgeführt.

3. **Analysespezifikation**
   Der Benutzer bestimmt den Anfrageausdruck, der durch das System ausgewertet werden soll.

4. **Anzeigespezifikation**
   Die Anzeigeart wird als eine Verbindung von graphischen Elementen mit Eigenschaften und Beziehungen der gemessenen Daten definiert.

5. **Ausführung**
   Die Anfrage wird in einen Ausdruck der Relationenalgebra übersetzt. Mit Hilfe der Sensorspezifikation und des Ausdrucks wird automatisch entschieden, welche Sensoren aktiviert werden müssen.

Zur Realisierung wird eine historische Datenbank eingesetzt, die eine Erweiterung einer relationalen Datenbank um Zeit- und Ablaufinformationen über durchgeführte Änderungen darstellt. Durch diese Speicherung wird neben einem effizienten Zugriff erreicht, daß der Anwender einerseits die Daten erneut abrufen und andererseits die Anzeigeart variieren kann, ohne daß das Programm erneut ausgeführt und ausgewertet werden muß.

2.3 KOJAK

Die Komplexität und Vielfalt der heute eingesetzten Systeme zur Performancediagnose auf Parallelrechnern erfordert Werkzeuge zur Automatisierung des Analysevorgangs. Aus diesem Grund wurde das Projekt KOJAK [10] (**K**it for **O**bjective **J**udgement and **A**utomatic **K**nowledge-based detection of bottlenecks) initiiert. KOJAK soll den Programmierer bei der Entwicklung seiner Anwendung auf einer CRAY T3E unterstützen.

Die Ziele des Projektes sind:

- häufig auftretende Fehler automatisch aufzudecken,
- ein allgemeines Werkzeug zu entwickeln, das auf verschiedenste Programmiermodelle und -umgebungen angepaßt werden kann, sowie
- das neue Werkzeug in existierende Umgebungen zu integrieren, um so bereits vorliegende Performancedaten wiederzuverwenden.

Die Struktur von KOJAK ist in Abbildung 2.1 illustriert. Die Analyse-Umgebung gliedert sich in die drei Hauptelemente Informationslieferung und -transformation, Performance-Datenbank sowie KOJAK selbst. Dabei sollen die Informationen aus bereits bestehenden Quellen, wie z. B. aus Compilern oder herkömmlichen Werkzeugen, zur Performanceanalyse gewonnen werden.

Die Datenbank bildet den Aufbewahrungsort aller gesammelten Daten. Für jede Informationsquelle muß hier eine Import-Schnittstelle zur Verfügung gestellt werden. Diese Schnittstelle soll ausgewählte, für eine weitere Analyse sinnvolle Daten in die

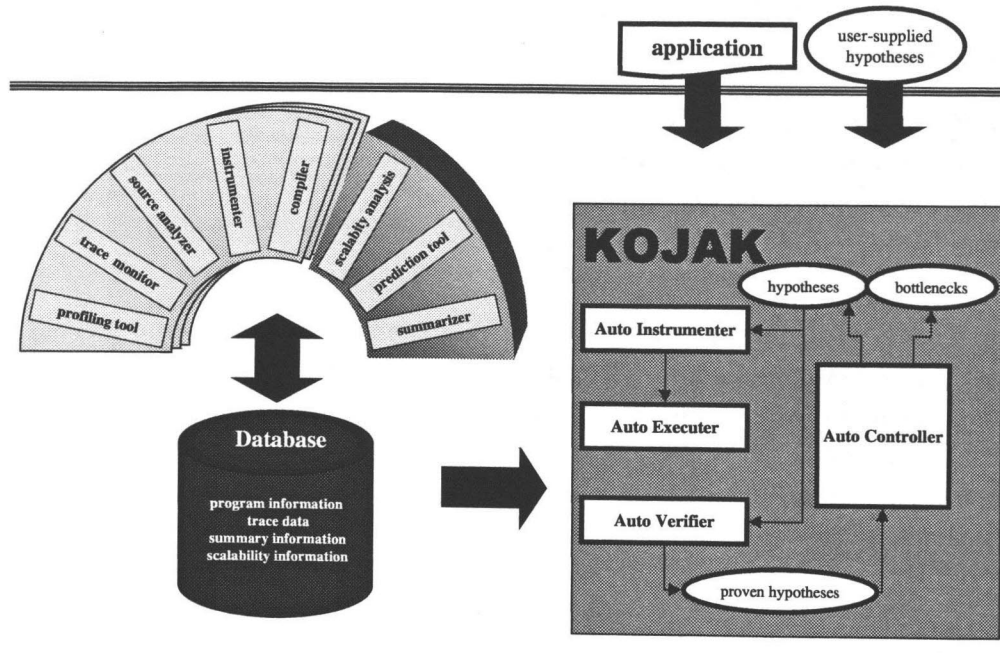

Abbildung 2.1: Struktur der automatischen Analyse-Umgebung KOJAK

Datenbank übertragen. Neben bereits aufgearbeiteten Daten, wie Ausführungszeiten oder Informationen über Quelltexte, sollen in der Datenbank auch komplette Dateien mit allen Laufzeitinformationen gespeichert werden. Auf diese kann dann zwar kein effizienter Zugriff erfolgen, aber sie stehen dadurch dennoch für komplexere Auswertungen zur Verfügung.

Eine Analyse der Daten soll – ähnlich wie bei Paradyn – auf Grundlage von Hypothesen und Tests realisiert werden. Sollte eine Hypothese sich als wahr erweisen, so muß sie zur genauen Lokalisierung und Klassifikation des Fehlers verfeinert werden.

KOJAK besteht aus den vier Komponenten *Auto Controller*, *Auto Instrumenter*, *Auto Executer* und *Auto Verifier*, die die gesamte Analyse-Umgebung steuern:

1. Der **Auto Controller** stellt die zentrale Komponente dar, die den Analyseprozeß und das Verfeinern der nachgewiesenen Hypothesen regelt. Alle anderen Komponenten werden vom Auto Controller aufgerufen.

2. Der **Auto Instrumenter** entscheidet, auf welchem Weg die Daten für den Test der möglichen Hypothesen gewonnen werden sollen. Liegen für eine Hypothese in der Datenbank noch keine relevanten Informationen vor, so muß das Programm geeignet instrumentiert werden.

3. Der **Auto Executer** führt die vom Auto Instrumenter instrumentierten

Programme aus und überträgt die ermittelten Daten in die Performance-Datenbank.

4. Der **Auto Verifier** testet die Hypothesen mit Hilfe der Informationen aus der Datenbank und entscheidet, ob ein Engpaß vorliegt.

Mit EARL [30] wurde bereits ein erster Baustein zur Nachbearbeitung von Ereignisspuren realisiert, der im Auto Verifier für den Test von komplexen Mustern zur Anwendung kommen kann.

In dieser Arbeit sollen die Grundbausteine für die weitere Entwicklung der KOJAK-Umgebung geschaffen werden. Aus Gründen des effizienten Zugriffs, der Datenerhaltung sowie der Client/Server-Unterstützung wird die Performance-Datenbank auf der Grundlage einer relationalen Datenbank entwickelt. Die Komponenten „Informationslieferung" und „KOJAK" werden exemplarisch durch Auswertung der vom CRAY-Analysewerkzeug Apprentice generierten Performancedaten sowie durch ein Java-Applet implementiert, das durch eine Analyse von Laufzeiten verschiedener Testszenarien mit unterschiedlichen Prozessorzahlen in der Lage ist, auf Skalierbarkeitsprobleme hinzuweisen.

Kapitel 3

Apprentice

3.1 Leistungsanalyse mit Apprentice

MPP Apprentice ist ein Werkzeug zur Visualisierung von Leistungsdaten von C/C++, Fortran und HPF[1]-Programmen, das auf den CRAY-Systemen T3D und T3E verfügbar ist. Die von diesem Werkzeug berechneten Leistungsdaten sollen die Grundlage der automatischen Performance-Analyse bilden.

Apprentice stellt dem Anwender sowohl Laufzeitdaten als auch statische, bereits zur Übersetzungszeit bekannte Informationen zur Verfügung. Diese werden über eine komfortable graphische Benutzeroberfläche in Form von Balkendiagrammen visualisiert und geben Auskunft über Ausführungszeiten von Anweisungen und Funktionen, Integer- und Fließkommaoperationen, Kommunikationszeiten sowie Speicheroperationen.

Dabei werden die Balken farblich markiert, so daß erkennbar ist, womit die angezeigte Zeit verbracht wurde. Hierbei wird unterschieden in:

- Overhead
- Overhead uninstrumentierter Funktionen
- Parallele Verarbeitung
- Ein-/Ausgabe
- Ein-/Ausgabe uninstrumentierter Funktionen
- Funktionsaufrufe
- Sonstige Messungen uninstrumentierter Funktionen

[1] High Performance Fortran.

Die Ergebnisse können jedoch neben der üblichen, graphischen Darstellung auch textuell in Form eines Reports abgerufen werden. Dieser Report kann entweder aus der Benutzerschnittstelle heraus oder direkt aus der Kommandozeile über die Option `-r` erstellt werden.

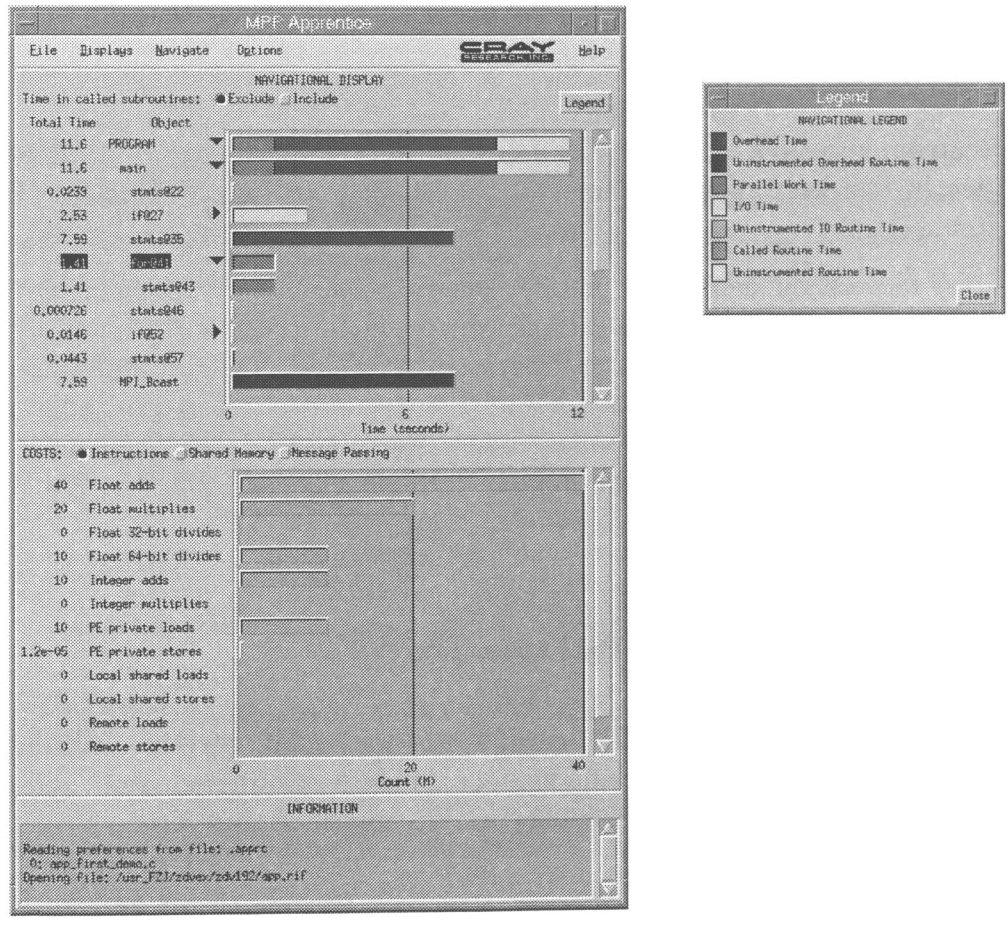

Abbildung 3.1: Benutzeroberfläche von MPP Apprentice

Apprentice liefert dem Anwender rein zusammenfassende Informationen, keine Ereignisspuren. Das Werkzeug führt daher während der Laufzeit im Gegensatz zu Debuggern oder Simulatoren keine Analyseschritte durch, sondern protokolliert lediglich die gemessenen Werte. Erst nach Ausführung des Programms wird Apprentice zur Analyse und Berechnung der über die Anzahl der Prozessoren und über die gesamte Laufzeit summierten Werte gestartet. Diese sind dann sowohl für das gesamte Programm als auch für Schleifen, Bedingungen und zugehörige Blöcke von Anweisungen verfügbar. Abbildung 3.1 zeigt die Benutzeroberfläche von Apprentice.

Dennoch wird natürlich die Ausführungszeit eines Programms durch den Instrumentierungsaufwand, d. h. durch das Hinzufügen der erforderlichen Anweisungen zur Zeitmessung von Programmfragmenten sowie zur Ermittlung der Anzahl ausgeführter Operationen verlängert. Dieser Overhead wird aber durch Apprentice ab-

geschätzt und die ermittelten Werte nachträglich automatisch korrigiert.

Die Instrumentierung erfolgt weitestgehend automatisch und wird durch einen Schalter beim Aufruf des Compilers gesteuert. Für die C/C++-Compiler ist dies die Option `-happrentice`, für die Fortran-Compiler geschieht dies durch Angabe von `-eA`. Im Quelltext ist jedoch in gewissen Grenzen eine Steuerung der Instrumentierung möglich. So gibt es Funktionen, die einzelne Prozessoren von der Messung ausschließen oder die Zähler zurücksetzen.

Da die Instrumentierung erst nach der Optimierungsphase durchgeführt wird, ist Apprentice in der Lage, sowohl unoptimierte als auch optimierte Programme zu verarbeiten. [7]

Die Analyse basiert auf den vom Compiler generierten *Compiler Information Files* sowie dem während der Programmausführung ermittelten *Runtime Information File*. Intern werden diese Dateien in einem Programmbaum dargestellt, der die Struktur des Programms hierarchisch wiedergibt und die gemessenen Werte in jeder Hierarchiestufe enthält. Im folgenden sollen die Auswertung und Verarbeitung dieser Dateien sowie die Werte, die durch sie zur Verfügung gestellt werden, näher beschrieben werden.

3.2 Compiler Information File

Ein Compiler Information File (CIF) enthält alle statischen, zur Übersetzungszeit bereits bekannten Daten, die für eine spätere Analyse erforderlich sind, d. h. alle relevanten Informationen aus den Quelltexten. Im einzelnen sind dies:

- Compiler-Optionen und die Eigenschaften des Rechners, auf dem das Programm übersetzt wurde,
- Namen, Argumente und ggf. Rückgabetypen von Funktionen und Prozeduren,
- Namen von Variablen, Konstanten und anderen Bezeichnern,
- (Fehler-)meldungen und Warnungen des Compilers,
- Typen, Zeilennummern und Quelldateien von Anweisungen und
- Querverweise für alle Bezeichner, Variablen, Funktionen, Funktionsargumente sowie symbolische Konstanten im Quelltext.

Diese Daten sind im CIF in verschiedenen Datensätzen organisiert, die unabhängig voneinander ausgelesen und ausgewertet werden können. Die häufig auftretenden Datensatz-Typen sind in Tabelle 3.1 dargestellt.

Von besonderer Bedeutung sind hierbei die `CIF_STMT_TYPE`-Datensätze, die alle Informationen über den Programmbaum beinhalten, der eine hierarchische Struktur

von Programmregionen darstellt. Diese Regionen können z. B. eine Schleife, einen `if`-Block oder einen Basisblock umfassen.

Ein Basisblock wird dabei aus zusammenhängenden Anweisungen gebildet, die sowohl einen definierten Anfangs- als auch Endpunkt haben und in einer festen Sequenz ohne Verzweigungen oder Sprünge ausgeführt werden. Dadurch bildet er eine logische Einheit, für die durch die Instrumentierung Messungen durchgeführt werden können.

Die Blätter des Baums repräsentieren solche Basisblöcke. Die Knoten des Baums stellen komplexe Gebilde dar, die Basisblock-Eigenschaften besitzen, aber auch die Attribute aller ihrer Kinder summieren. So beinhaltet z. B. ein `if`-Knoten einerseits den Basisblock des Tests der Bedingung, umfaßt andererseits aber auch den Rumpf des Blockes.

Datensatztyp	Beschreibung
`CIF_SUMMARY`	Zusammenfassung
`CIF_HEADER`	Kopf
`CIF_STMT_TYPE`	Programmregionen
`CIF_C_ENTRY`, `CIF_C_ENTRY_END`	Funktionen in C/C++
`CIF_ENTRY`, `CIF_F90_ENTRY`	Funktionen in Fortran
`CIF_UNIT`, `CIF_END_UNIT`	Begrenzung einer logischen Einheit[2]
`CIF_BE_NODE`	Zuordnung der Basisblöcke

Tabelle 3.1: Datensatztypen im Compiler Information File (CIF)

Zusätzlich zu diesen strukturellen und organisatorischen Daten werden den Basisblockdaten des CIF vom Compiler bereits die Anzahl der Instruktionen je Durchlauf sowie Abschätzungen über die erforderliche Anzahl von Taktzyklen hinzugefügt.

Ursprünglich liegt ein CIF im ASCII-Format vor, es kann aber in ein effizienteres Binärformat konvertiert werden. In beiden Fällen ist ein einfacher Zugriff durch Bibliotheksfunktionen möglich. Ein CIF, erkennbar durch die Dateiendung `.T`, wird vom Compiler für jede Quellcode-Datei generiert, bei deren Übersetzungsaufruf die Apprentice-Option mit angegeben wurde. [6]

3.3 Runtime Information File

Das Runtime Information File (RIF) wird durch die Instrumentierung des Programms bei Programmausführung generiert. Enthalten sind alle dynamischen gemessenen Daten, die zur Übersetzungszeit noch nicht bekannt waren. Es besteht aus einer globalen Header-Information – dem RIF-Header – sowie aus getrennten Datenblöcken für jede an der Übersetzung beteiligte Quelldatei (vgl. Abbildung 3.2).

[2]Nur in Fortran erforderlich, da in C/C++ je eine Quelldatei eine logische Einheit bildet.

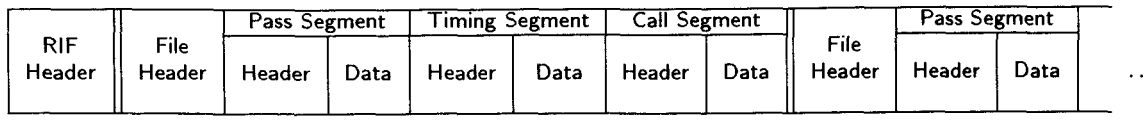

Abbildung 3.2: Aufbau des Runtime Information File (RIF)

Der RIF-Header beinhaltet die Anzahl der Prozessoren, Datum und Zeit der Ausführung des Programms sowie die Taktrate der Prozessoren. Für jede Quelldatei wird dann ein File-Header mit drei nachfolgenden Segmenten angelegt. Im File-Header werden lediglich der Dateiname des Quelltextes, das Arbeitsverzeichnis und die Argumente des Programmaufrufs gesichert.

Jedem File-Header folgen drei Segmente, die jeweils aus einem Segment-Header, der den Typ und die Länge angibt, sowie den Datensätzen mit den während der Ausführung gemessenen Werten bestehen. Im Pass-Segment wird die Anzahl der Durchläufe von Basisblöcken, im Timing-Segment die Messung der Taktzyklen von Anweisungen und im Call-Segment Angaben zur Anzahl und Ausführungszeit von Funktionsaufrufen abgelegt.

Dabei bestehen die Einträge der jeweiligen Datensätze nicht aus einem einzelnen Wert, sondern aus einem komplexen Datentyp, der

- die Summe $\sum_i x[i]$ der gemessenen Werte aller Prozessoren,
- die Summe $\sum_i x[i]^2$ der quadratischen Werte aller Prozessoren,
- die minimalen und maximalen Werte $\mathsf{min}(x[i])$ und $\mathsf{max}(x[i])$ sowie
- die logischen Nummern der Prozessoren, auf denen $\mathsf{min}(x[i])$ und $\mathsf{max}(x[i])$ ermittelt wurden,

enthält.

Eine Information zur späteren Zuordnung zu den Basisblöcken, etwa eine Satznummer oder ein Verweis auf einen CIF-Datensatz, liegt an dieser Stelle jedoch nicht in direkter Form vor. Die Datensätze des RIF werden lediglich durch ihre Position in der Datei identifiziert.

Ursprünglich war in der Spezifikation von Apprentice noch ein weiteres Segment vorgesehen. Ebenso besteht durch die Organisation der Datenstrukturen die Möglichkeit, eine unterschiedliche Anzahl und Reihenfolge der Segmente zu verarbeiten. Beide Fähigkeiten werden jedoch von Apprentice für das MPI-Programmiermodell nicht mehr unterstützt: Die Reihenfolge der Segmente ist festgelegt, die Anzahl wird konstant auf drei gesetzt. Zusätzliche Segmente werden nicht mehr generiert. Daher wird auf diese Punkte im Rahmen dieser Arbeit nicht näher eingegangen.

3.4 Verarbeitung von CIF und RIF in Apprentice

Die Auswertung eines Programmablaufs erfolgt in vier wesentlichen Schritten, die in Abbildung 3.3 verdeutlicht sind.

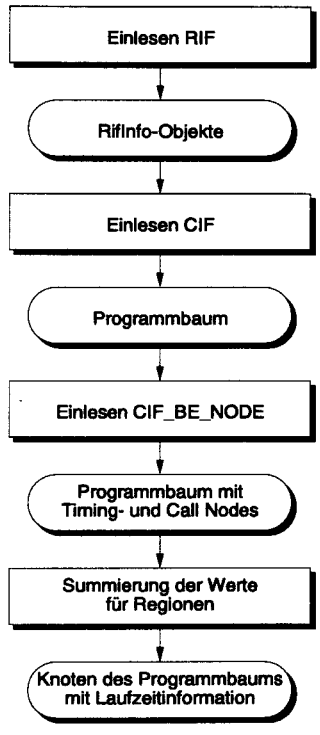

Abbildung 3.3: Generierung des Programmbaums

Durch das Einlesen des RIF werden die Taktfrequenz des Rechners, alle erforderlichen Informationen über Quelldateien bzw. Arbeitsverzeichnisse sowie die gemessenen Daten – zu diesem Zeitpunkt jedoch noch ohne Bezug zum Quelltext – gewonnen.

Das Einlesen erfolgt gemäß der in Abbildung 3.2 dargestellten Reihenfolge. Dadurch können die Segmentdatensätze intern genau in der Sequenz, in der sie auch in der Datei vorliegen, in einem Feld gespeichert und somit über ihren Index adressiert werden.

Dabei wird intern ein globales RIF-Objekt angelegt, das dann Verweise auf einzelne *RifInfo*-Objekte enthält, die die Daten der einzelnen Quelldateien repräsentieren und in denen die Inhalte der Segmente getrennt voneinander in drei Listen verwaltet werden. Es existiert also je Quelldatei genau ein RifInfo-Objekt.

Beim Einlesen eines CIF werden zunächst die Informationen aus den Datensätzen des Typs `CIF_HEADER` und `CIF_SUMMARY` ausgewertet. Hieraus werden dann die Sprache sowie die Dateinamen und internen Identifikationsnummern der beteiligten Quelltexte ermittelt. Danach werden innerhalb einer programmiersprachenspezifischen

Funktion die einzelnen Positionen des CIF verarbeitet. Dabei werden alle Funktionen des Quelltextes – im CIF als `CIF_C_ENTRY`, `CIF_C_ENTRY_END` bzw. `CIF_F90_ENTRY` markiert – in einer Funktionsliste gespeichert.

Aus diesen Funktionen wird dann, zusammen mit allen CIF-Einträgen des Typs `CIF_STMT_TYPE`, der Programmbaum gebildet. Die Wurzel dieses Baums ist durch einen `PROGRAM`-Knoten bestimmt; die Knoten der ersten Ebene sind genau die im Programm vorkommenden Funktionen. An diesen Funktionen hängen dann die ein-

```
 1 /*
 2  * app_first_demo.c
 3  *
 4  * Sequentielle Berechnung von Pi
 5  *
 6  */
 7
 8 #include "mpi.h"
 9
10 void main(int argc, char **argv)
11 {
12   int numprocs, myid;
13   int i, n;
14   double mypi, pi, sum, h, x;
15   double PI25DT = 3.14159265358979323846264 3;
16   double time1, time2;
17   char input[80];
18
19   MPI_Init(&argc,&argv);
20   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
21   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
22
23   if (myid == 0) /* Master */
24     {
25     printf("Bitte Aufteilung des Intervalls [0,1] eingeben: ");
26     gets(input);
27     sscanf(input, "%d", &n);
28     time1 = MPI_Wtime();
29     }
30   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
31
32   sum = 0;
33   h = 1.0/n;
34   for (i=myid+1; i<=n; i+=numprocs)
35     {
36     x = (i-.5)*h;
37     sum += 4.0/(1.0+(x*x));
38     }
39   mypi = sum * h;
40
41   MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
42
43   if (myid == 0)
44     {
45     time2 = MPI_Wtime();
46     printf("%d PEs:\nPi=%.15f \nAbweichung: %.15f \n", numprocs, pi, PI25DT-pi);
47     printf("Zeit: %.12f\n\n", time2-time1);
48     }
49   MPI_Finalize();
50 }
```

Abbildung 3.4: C-Programm als Beispiel zur Auswertung mit Apprentice

zelnen Knoten der Regionen der Unterprogramme.

Als Einführungsbeispiel zeigt Abbildung 3.4 ein kurzes C-Programm, das der sequentiellen Berechnung von π als $\int_0^1 4/(1+x^2)\,dx$ dient. Zur Übersichtlichkeit wurden Zeilennummern ergänzt. Der resultierende Programmbaum ist in Abbildung 3.5 dargestellt: Jeder Knoten enthält neben seiner Identifikation, die aus dem Anweisungstyp mit erster Zeilennummer oder aus dem Funktionsnamen besteht, eine Liste aller Zeilen, über die sich dieser Knoten erstreckt, sowie einen Verweis auf den Knoten der aufrufenden Funktion. Da bei Knoten, die eine Funktion oder ein Unterprogramm repräsentieren, mehr Informationen, so z. B. die Anzahl der Aufrufpunkte, verfügbar sind, wird hier zwischen *Function*-Knoten für Funktionen und Unterprogramme und *CodeObject*-Knoten für in der Hierarchie darunter liegende Programmregionen unterschieden.

Die in Abschnitt 3.1 beschriebene farbliche Darstellung der verschiedenen Zeitarten erfordert eine Klassifikation der Zeiten bereits während der Messung. Dies ist möglich, da bereits zur Übersetzungszeit, d. h. während der Instrumentierungsphase, jede Anweisung genau einer Zeitart zugeordnet werden kann.

Da ein Basisblock durchaus aus mehreren Anweisungen bestehen kann, müssen darin häufig unterschiedliche Zeittypen betrachtet werden. Aus diesem Grund ist ein weiteres Aufsplitten der Basisblöcke in gemeinsam instrumentierte Teilstücke derselben

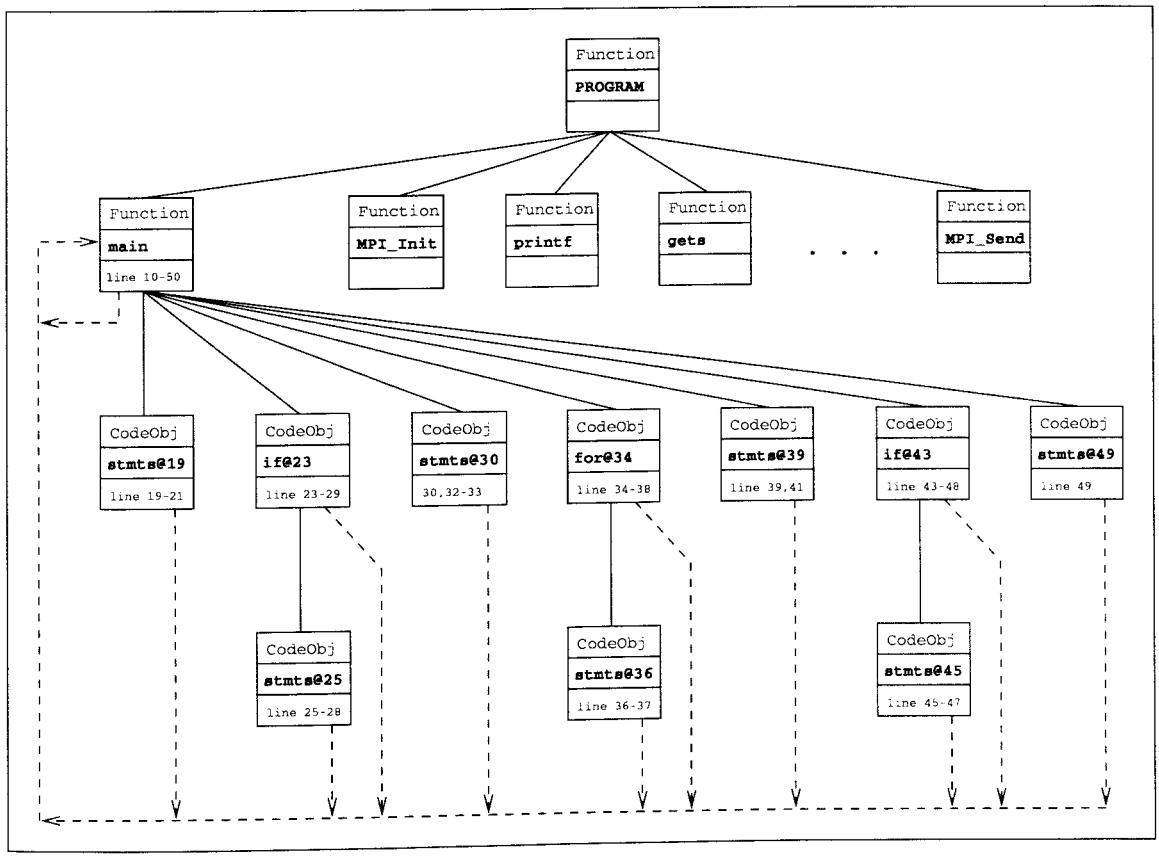

Abbildung 3.5: Beispiel-Programmbaum

Zeitart erforderlich. Dies kann bei zusammengesetzten Anweisungen sogar so weit führen, daß eine einzelne Programmzeile in mehrere Messungen aufgebrochen wird, wie es z. B. bei komplexen Berechnungen, die womöglich noch Typkonvertierungen enthalten, mit anschließender Variablenzuweisung der Fall ist.

Zur Realisierung dieses Aufsplittens werden die `CIF_BE_NODE`[3]-Datensätze eingelesen. Diese repräsentieren jeweils ein gemeinsam instrumentiertes Fragment eines Basisblocks.

Aus diesen Datensätzen werden dann *Timing-Nodes* oder *Call-Nodes* gebildet:

- Die **Timing-Nodes** dienen – analog zu den Timing-Segmenten – der Speicherung von Meßdaten auf Anweisungsebene. Sie enthalten Ausführungszeiten, Angaben zur Anzahl der Durchläufe sowie die Anzahl der Instruktionen und die bei der Übersetzung des Programms schon geschätzten Taktzyklen.

- Die **Call-Nodes** als Gegenstück zu den Datensätzen der Call-Segmente enthalten alle bereits bei den Timing-Nodes beschriebenen Werte mit Ausnahme der geschätzten Taktzyklen. Hinzu kommen Angaben über die Anzahl der Aufrufpunkte sowie die Minimal- und Maximalmessungen mit zugehörigen Prozessornummern[4], die Streuung `stdev` sowie das arithmetische Mittel `mean` der Werte.

Diese werden an die zugehörigen Baumknoten gebunden und dienen als Verknüpfung von CIF und RIF, so daß hierüber ein Zugriff auf die zu Beginn der Auswertung eingelesenen zusammenhanglosen Datensätze des RIF erfolgen kann. Zu jedem Timing- bzw. Call-Node existiert in den RifInfo-Objekten, d. h. in den Segmenten des RIF, genau ein Datensatz, der die Messung einer Zeitart enthält.

Apprentice teilt die Messungen intern in zwei verschiedene Klassen von *Timing-Types* ein: Zeiten mit parallel genutzter CPU-Zeit und Overhead-Zeiten, die durch das parallele Programmiermodell entstehen. Die Timing-Types dieser Klassen sind in Tabelle 3.2 dargestellt und entsprechen genau den Zeitarten, die in der Benutzeroberfläche visualisiert werden. Einige Typen werden dabei direkt übernommen, alle Overhead-Typen und die Ein-/Ausgabe werden jedoch zu einer Größe zusammengefaßt.

In Tabelle 3.3 wird zusätzlich noch eine dritte Klasse unterschieden, in die zu informativen Zwecken alle Typen aufgenommen wurden, die in der betrachteten Version von Apprentice nicht mehr unterstützt werden.

Eine Identifizierung des passenden Datensatzes im RIF wird einzig über den in Abschnitt 3.3 beschriebenen Feldindex, also über die Position in der Datei realisiert.

[3]Der `CIF_BE_NODE`-Datensatz wird nur für die Kooperation zwischen den CRAY Compilern und Apprentice benötigt. Er ist daher für eine allgemeine, von Apprentice unabhängige Verwendung eines CIF irrelevant und daher in [6] undokumentiert.

[4]Diese werden unverändert aus den in Abschnitt 3.3 beschriebenen Strukturen übernommen.

Zeittyp	Beschreibung	
tt_work	Benutzerzeit	
tt_call	Funktionsaufruf	
tt_read	Eingabe	
tt_write	Ausgabe	Zeiten mit
tt_uninst_ovhd	Overhead uninstrumentierter Funktionen	paralleler Verarbeitung
tt_uninst_io	Ein-/Ausgabe uninstrumentierter Funktionen	
tt_uninst	Andere Daten uninstrumentierter Funktionen	
tt_entry	Eintritt in Funktion	
tt_barrier	Barrier wait	
tt_get	Shared Memory **get**	Durch MPI und Zugriffe
tt_put	Shared Memory **put**	auf verteilten Speicher
tt_pvm	MPI Routine	verursachter Overhead
tt_lock	SET_LOCK()	
tt_event	WAIT_EVENT()	

Tabelle 3.2: Apprentice-Zeittypen

Dies ist möglich, da die exakte Reihenfolge der Abarbeitung von Anweisungen aufgrund der Basisblock-Eigenschaft bereits zur Übersetzungszeit bekannt ist. Daher ist auch bei der Instrumentierung bereits vorgegeben, in welcher Sequenz die Zeiten der einzelnen Zeittypen im RIF gespeichert werden. Da der Compiler auch das CIF generiert, kann dadurch in den CIF_BE_NODE-Datensätzen der spätere Index des RIF-Datensatzes schon bestimmt werden und somit in die Timing- bzw. Call-Nodes aufgenommen werden.

Die CIF_BE_NODE-Datensätze enthalten zusätzlich noch einen Index, der in das Pass-Segment des RIF verweist und durch den auf die Anzahl der Durchläufe durch das betreffende Programmfragment zugegriffen werden kann. Eine denkbare Generierung eines „Pass-Nodes“ in Analogie zu den Timing- und Call-Nodes ist für die Sätze des Pass-Segmentes nicht erforderlich. Dies beruht auf der Tatsache, daß die Anzahl der Durchläufe für alle Anweisungen eines Knotens im Programmbaum aufgrund der Basisblock-Eigenschaft identisch ist. Daher können die Werte des Pass-Segmentes direkt den Timing- und Call-Nodes zugeordnet werden, die bekanntlich Fragmente

Zeittyp	Beschreibung
tt_master	Warten des Master-Prozessors
tt_end_master	Warten von $n-1$ PEs auf den Master
tt_gmr	Globaler Speicherzugriff
tt_pfq_wait	Warten auf Blocktransfer
tt_critical	Eintritt in Critical Section
tt_end_critical	Austritt aus Critical Section
tt_atomic	Atomares Update
tt_loopstart	Shared Loop Start
tt_loopdist	Shared Loop Verteilung
tt_loopend	Shared Loop Ende

Tabelle 3.3: Nicht mehr unterstützte Apprentice-Zeittypen

eines Basisblocks sind.

Durch die verfügbaren Pass-Informationen können nun in den Timing- und Call-Nodes mit Hilfe der aus dem CIF gewonnenen Anzahl der Instruktionen im Segment durch einfache Multiplikation die exakt benötigten Schritte verschiedener Instruktionstypen (vgl. Tabelle 3.4) ermittelt werden. Ebenso können die geschätzten Taktzyklen in Ausführungszeiten umgerechnet werden, falls wider Erwarten kein Index für einen passenden RifInfo-Eintrag aus dem Timing- oder Call-Segment vorliegt.

Instruktionstyp	Beschreibung	Gruppierung	
it_fadd	Fließkommaaddition	f_inst()	total_inst()
it_fmul	Fließkommamultiplikation		
it_fdiv_s	Fließkommadivision 32-bit		
it_fdiv_t	Fließkommadivision 64-bit		
it_iadd	Integer Addition	i_inst()	
it_imul	Integer Multiplikation		
it_idiv	Integer Division		
it_load	LOAD	load_inst()	
it_load_ls	Local shared LOAD		
it_load_r	Remote LOAD		
it_store	STORE	store_inst()	
it_store_ls	Local shared STORE		
it_store_r	Remote STORE		
it_other	Andere Instruktionen		

Tabelle 3.4: Apprentice-Instruktionstypen

Abschließend werden alle Knoten des nun komplett aufgebauten, um Basisblockfragmente und Laufzeitinformationen erweiterten Programmbaums nochmals rekursiv durchlaufen, die Werte aus den Timing- und Call-Nodes sowie aus den Kind-Knoten aufsummiert und in den Knoten gespeichert. Dadurch ist während der Analysephase ein schnellerer Zugriff auf die Daten möglich. Abbildung 3.6 zeigt den um Timing- und Call-Nodes ergänzten Programmbaum aus Abbildung 3.5.

3.5 Ausgabe der Leistungsdaten für KOJAK

3.5.1 Anpassung von Apprentice

Apprentice bietet keine Schnittstelle für den Export der berechneten Werte. Aus diesem Grund mußten zur Realisierung dieser Arbeit geringfügige Eingriffe in den Quelltext von Apprentice vorgenommen werden. Es wurde jedoch darauf geachtet, die Veränderungen so gering wie möglich zu halten, damit einerseits die Arbeitsweise nicht beeinträchtigt wird und andererseits eine eventuell erforderlich werdende neue Anpassung für Nachfolgeversionen von Apprentice leicht nachvollziehbar ist. Daher wurde auf eine Modifikation der graphischen Oberfläche verzichtet, obwohl die

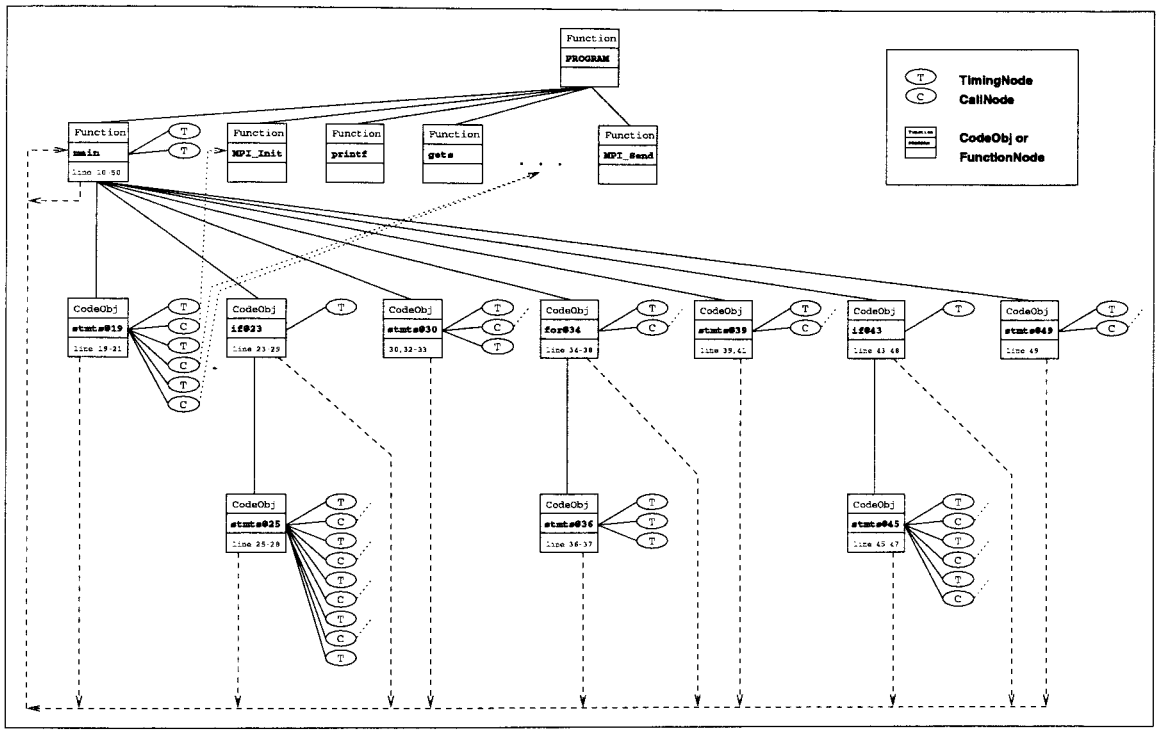

Abbildung 3.6: Beispiel-Programmbaum mit Timing- und Call-Nodes

Ergänzung eines zusätzlichen Menüpunktes, mit dem die Ausgabe der Exportdatei angewiesen werden kann, sinnvoll gewesen wäre.

Die Generierung der Exportdatei wird – analog zum Erstellen des Reports – über einen Schalter in der Kommandozeile gesteuert. Dies ermöglicht auch eine automatische Generierung, wie sie z. B. in einer Batch-Datei oder durch Aufruf in einem externen Werkzeug denkbar wäre. Dazu wurde Apprentice um eine zusätzliche Option `-k` ergänzt, bei deren Aufruf in eine neu erstellte Funktion verzweigt wird, die die internen Strukturen durchläuft und die für KOJAK relevanten Daten ausliest.

Apprentice wurde ausschließlich in C++ mit den Möglichkeiten der objektorientierten Programmierung geschrieben. So wurde für die Speicherung des Programmbaums, der RIF- und CIF-Daten sowie der Timing- und Call-Nodes das Konzept der Klassenhierarchie unter Ausnutzung von Vererbung und virtuell definierten Funktionen verwendet.

Bei den Modifikationen wurde die definierte Struktur der Klassen akzeptiert und in keiner Form verändert. Dies bedeutet, daß die behandelten Daten nur aus den öffentlichen Variablen und Funktionen der einzelnen Objekte ermittelt werden. Weiterhin wurde versucht, die Erweiterungen in Form von eigenständigen Quelltexten vorzunehmen und bestehende Dateien nicht zu berühren. Dieses Konzept ließ sich infolge einer erforderlichen Schnittstelle nicht ausnahmslos durchsetzen. Dennoch ist es gelungen, die unmittelbaren Eingriffe auf wenige Punkte in der zentralen Programmdatei `main.C` sowie in der Datei `Efficiency/app_env.C` für die Umgebungssteuerung der Anwendung zu beschränken.

In Tabelle 3.5 sind die direkten Änderungen, die erforderlich waren, kurz zusammengestellt. Die expliziten Programmstellen und ihre Modifikationen werden in Anhang A beschrieben. Auf eine genaue Ausführung soll an dieser Stelle verzichtet werden, da die Darstellung der internen Strukturen und Funktionen von Apprentice für das weitere Vorgehen nicht von Interesse ist.

Datei	Funktion	Beschreibung der Änderung
`app_env.C`	`validate_options`	Der Schalter `-k` wird bei der Prüfung der Kommandozeilenparameter zugelassen.
`app_env.C`	`process_command_options`	Bedeutung von `DisplaySupport::report`. Bisher wurde mittels des Wertes 0 oder 1 gespeichert, ob über die Kommandozeile ein Report angefordert wurde. Es wurde der Wert 2 hinzugenommen, der anzeigt, daß ein KOJAK-Export generiert werden soll, falls in der Kommandozeile die Option `-k` gewählt wurde.
`app_env.C`	`process_command_options`	Einige `if`-Bedingungen prüfen den Inhalt von `DisplaySupport::report` auf den Wert 1. Da die Behandlung für Report und Export hier identisch ist, sollte stattdessen entweder auf `> 0` oder auf `TRUE` geprüft werden.
`app_env.C`	`usage`	Ein Benutzerhinweis wird im Hilfetext mit ausgegeben.
`main.C`	`apprentice`	Auswertung von `DisplaySupport::report`. Je nach Wert 1 oder 2 wird jetzt die alte Funktion `make_report()` oder eine neue Funktion `do_kojak()` aufgerufen.

Tabelle 3.5: Erforderliche Änderungen an den Apprentice-Quelltexten

Die Funktion `do_kojak()` beinhaltet die für KOJAK relevanten Erweiterungen. Darin wird die Apprentice-Baumstruktur durchlaufen und die Daten gesammelt, die für eine spätere Analyse interessant sein könnten. Aufgrund der starken Vernetzung des Programmbaums gestaltet sich die Verarbeitung relativ einfach. So kann z. B. ein Zugriff auf die Daten eines Funktionsaufrufs wahlweise über eine Aufruferliste des Knotens der betreffenden Funktion oder über den Call-Node, der den Aufruf repräsentiert, erfolgen. Zusätzlich hierzu existiert von diesem Call-Node dann noch ein Verweis auf den entsprechenden Knoten. Ferner gibt es an jedem Knoten im Programmbaum eine direkte Möglichkeit, auf den Knoten der Funktion zuzugreifen, der dieser Knoten untergeordnet ist – unabhängig von der Tiefe, in der sich der Knoten befindet. Diese Zusammenhänge sind auch in Abbildung 3.6 verdeutlicht.

Die Abarbeitung des Baums erfolgt durch `do_kojak()` in einer rekursiven Funktion, die beginnend mit der Wurzel alle Knoten des Baums untersucht. Ein Durchlaufen der Timing- und Call-Nodes des jeweils betrachteten Knotens ist an dieser Stelle nicht erforderlich, da durch die komplexe Struktur der Knoten bereits hier alle Informationen der Timing- und Call-Nodes durch vereinfachte Zugriffsfunktionen weitergereicht werden.

Funktion/Variable	Beschreibung
`stmt_type()`	Anweisungstyp (Zuweisung, Bedingung, Schleife etc.)
`label()`	Name des Knotens
`line_range(int)`	Zeilennummern in der betreffenden Quelldatei
`func_node()`	Funktion, der der Knoten angehört
`call_to(int)`, `called_from(int)`	Bei Funktionen Angaben zu Aufrufpunkten
`timing(Rif *, int)`	Gemessene Zeit des angegebenen Zeittyps
`excl_time(Rif *)`	Summe aller Zeittypen außer `tt_call`
`incl_time(Rif *)`	Summe aller Zeittypen
`ovhd_time(Rif *)`	Summe aller Zeittypen der Overhead-Klasse
`est_time(Rif *)`	Im CIF bereits geschätzte Zeit
`i_inst(Rif *)`	Anzahl der Integer-Instruktionen
`f_inst(Rif *)`	Anzahl der Fließkomma-Instruktionen
`load_inst(Rif *)`	Anzahl der LOAD-Instruktionen
`store_inst(Rif *)`	Anzahl der STORE-Instruktionen
`total_inst(Rif *)`	Gesamtanzahl der Instruktionen

Tabelle 3.6: Verfügbare Daten in den Knoten des Programmbaums

Die in den Objektknoten verfügbaren relevanten Daten sind in Tabelle 3.6 zusammengestellt. Einige Funktionen liefern ein einzelnes Ergebnis, andere verdecken ein internes Feld oder eine verkettete Liste und müssen durch eine Schleife mit einem Index ausgelesen werden.

Im folgenden soll nun das genaue Ausgabeformat der KOJAK-Exportdatei beschrieben werden.

3.5.2 Schnittstelle

Der Export der Daten wird durch eine ASCII-Datei mit Datensätzen acht verschiedener Typen realisiert. Die einzelnen Datensätze sind durch ein CR/LF voneinander getrennt, d. h., eine Zeile repräsentiert einen Satz. Dabei wird der Typ durch ein Zeichen in der ersten Spalte festgelegt, beginnend mit der zweiten Spalte folgen dann in Abhängigkeit des Datensatztyps durch Kommata getrennt die einzelnen Datenfelder. Tabelle 3.7 zeigt die verschiedenen Datensätze.

Alle Felder mit Ausnahme von Zeitstempeln, die im Format `DD-MON-YY HH:MM:SS` abgelegt werden, sind in ihrer Länge variabel. Gemessene Zeiten werden in Taktzyklen angegeben. Ein Wechsel der Zeitskalierung wie in der Benutzerschnittstelle von Apprentice ist nicht möglich. Eventuell erforderliche Konvertierungen können aber, falls erforderlich, mit Hilfe des Taktzyklus ermittelt werden. Mittelwerte sowie Standardabweichungen werden als Fließkommazahlen, die übrigen Werte als ganze Zahlen ausgegeben.

Das vollständige Format der Exportdatei ist in Anhang B dargestellt. Der Aufbau der Datei orientiert sich stark am Entwurf der im folgenden Kapitel beschriebenen

Kennzeichen	Beschreibung
*	Programm- und Testlaufinformationen
S	Quellcode
F	Funktionsdaten
C	Funktionsaufruf
N	Programmregion
T	Summierte Laufzeiten
Y	Laufzeiten nach Apprentice-Zeittypen
L	Quellcode-Referenz

Tabelle 3.7: Datensätze der Apprentice-Exportdatei

Datenbank. Insbesondere die zur Identifikation der Programmregionen, Quellcodes und Testläufe erforderlichen Daten lehnen sich bereits an die später verwendeten *Schlüssel* der Datenbankrelationen an. So ist z. B. zur eindeutigen Zuordnung einer Programmregion die Angabe des Namens in der Form `stmts@42` nicht ausreichend, da es möglich ist, daß in verschiedenen Quelltexten in Zeile 42 eine Gruppe von Anweisungen beginnt. Durch Hinzunahme des Funktionsnamens, in der die Region auftritt, kann sie jedoch jederzeit erkannt werden.[5]

Ebenfalls aus Gründen der Eindeutigkeit wurde der C-Datensatz für Funktionsaufrufe um eine ID-Nummer ergänzt, um den Fall abzudecken, daß innerhalb einer Region dieselbe Funktion mehrfach aufgerufen wird. Diese, vom angepaßten Apprentice zusätzlich generierte Zahl, wird durch simples Durchnumerieren der Aufrufe innerhalb einer Region ermittelt.

[5]Genaugenommen müßte nicht der Funktionsname, sondern Name, genauer Pfad, Datum sowie Zeit der Quelldatei hinzugenommen werden, da sich eine Funktion theoretisch über mehrere Quelltexte erstrecken kann. Da dies jedoch in der Programmierpraxis nicht üblich ist, wurde hierauf zugunsten der Übersichtlichkeit verzichtet.

Kapitel 4

Design und Realisierung einer relationalen Datenbank

4.1 Relationale Datenbanken

Eine Datenbank ist die Integration aller Daten einer zugehörigen Anwendung in einen Bestand. Sie bietet den Vorteil einer zentralen Verwaltungsinstanz für die Daten aller Nutzer der Anwendung. Dies gewährleistet Schutz vor unberechtigten Zugriffen sowie vor Inkonsistenzen bei der Aktualisierung der Daten. Daraus resultiert eine Unabhängigkeit der Daten von den mit ihnen arbeitenden Benutzern, so daß Entwickler von Softwaresystemen sich stärker auf die Kernprobleme ihrer Anwendungen konzentrieren können, statt sich mit der Implementierung und Organisation ihrer Datenstrukturen zu beschäftigen.

In der Terminologie der Datenbanken ist zwischen den Begriffen *Datenbank*, *Datenbank-Verwaltungs-System* (Database Management System, kurz DBMS) und *Datenbanksystem* zu unterscheiden: Der Datenbestand wird als Datenbank betitelt, die Datenbanksoftware, die die Datenbank verwaltet und dem Anwender verschiedene Schnittstellen zur Datenbank zur Verfügung stellt, wird als DBMS bezeichnet. Die Kombination von Datenbank und DBMS bildet schließlich das Datenbanksystem.

Im Zusammenhang mit der automatischen Performance-Analyse gibt es bereits Ansätze von R. Borgeest und Ch. Rödel [2], die in einer Erweiterung ihres Analyse-Werkzeugs TATOO [1] eine relationale Datenbank statt interner Datenstrukturen verwenden. Dadurch werden Speicherplatzprobleme beim Einlesen der teilweise sehr großen Trace-Dateien vermieden. Des weiteren müssen die Dateien nur noch einmal eingelesen werden und stehen danach mehreren Entwicklern gleichzeitig zur Verfügung. Die Messungen von R. Borgeest und Ch. Rödel ergaben, daß die Performance etwa eine Größenordnung unter der der ursprünglichen Version liegt, was in Anbetracht der obigen Vorteile durchaus akzeptabel erscheint. R. Snodgrass [26] verwendete bereits 1988 das relationale Modell als Grundlage zur Entwicklung eines Trace-Monitors, um zu entscheiden, welche Daten gemessen werden und wie sie

gesammelt werden sollen.

Zur Beschreibung der physikalischen Strukturen sowie der Zusammenhänge in einer Datenbank werden *Datenmodelle* verwendet, deren Entwicklung auf 1965-1975 zurückgeht. Das in kommerziellen Datenbanksystemen am weitesten verbreitete Modell ist das von E. F. Codd entwickelte *Relationenmodell* [5], bei dem die zu speichernden Informationen durch *Tabellen* dargestellt werden, deren Zeilen jeweils die konkreten Angaben der einzelnen Datensätze enthalten. Die Kopfzeile einer Tabelle bildet dabei eine Auflistung der *Attribute*, also aller möglichen Eigenschaften eines jeden Datensatzes. Das Relationenmodell eignet sich für alle Arten von Informationen, die sich mit Hilfe von Tabellen repräsentieren lassen. Komplexere Zusammenhänge, die sich nicht durch eine einzige Tabelle modellieren lassen, werden durch miteinander verknüpfte Tabellen dargestellt. Der Erfolg der relationalen Datenbanken beruht dabei auf der Anschaulichkeit der Tabellen.

Da diese Tabellen – wie der Name des Modells schon andeutet – Relationen repräsentieren, sind die Datenzeilen jedoch nicht als Folge, sondern als Menge aufzufassen. Aus diesem Grund kann jeder Datensatz auch nur ein Mal in die Menge aufgenommen werden, folglich sind die enthaltenen Zeilen wohlunterscheidbar.

Zur eindeutigen Identifikation eines Satzes sind im allgemeinen nicht alle Attribute erforderlich. Meist genügt die Angabe eines oder mehrerer *Schlüsselattribute*, durch die alle anderen Attribute bereits bestimmt sind, wie es beispielsweise in einer Kundendatenbank durch die Kundennummer üblich ist. Verschiedene Tabellen können über ihre Schlüsselattribute miteinander verknüpft sein. Zur besseren Unterscheidung werden in diesem Zusammenhang die Attribute, die den Schlüssel der Relation enthalten, auf die verwiesen wird, als *Fremdschlüssel* bezeichnet. Der eigene Schlüssel hingegen ist der *Primärschlüssel*. Für den Anwender sind diese Verweisstrukturen zunächst unsichtbar.

Für Manipulation und Zugriff auf die Daten stehen drei Grundoperationen zur Verfügung:

- Die *Projektion* blendet einen Teil der Spalten einer Tabelle aus, „verkürzt " die Tabelle also in vertikaler Richtung. Da das Ergebnis bei Betrachtung als Menge keine doppelten Zeilen besitzen darf, kann das Weglassen einiger Spalten auch das Ausblenden einiger jetzt doppelt auftretender Zeilen zur Folge haben.

- Die *Selektion* als horizontales Pendant zur Projektion wählt eine oder mehrere Zeilen einer Tabelle nach vorgegebenen Vergleichskriterien aus und blendet alle übrigen Zeilen aus.

- Der *Verbund* oder *Join* verbindet zwei Tabellen zu einer neuen. Die Attribute dieser neuen Relation werden aus der Vereinigung der Attribute der am Join beteiligten Tabellen gebildet. Der Vergleich aller gemeinsamen Spalten, die in beiden Ursprungsrelationen vorkommen, dient hierbei als Kriterium zur Bildung der Zeilen der neuen Relation. Existieren keine gemeinsamen Spalten,

so entspricht der Join einem Kartesischen Produkt. Für den Join-Operator gelten sowohl das Kommutativ- als auch das Assoziativgesetz.

Zur Arbeit mit den Daten stellen relationale DBMS spezielle Kommandos zur Verfügung. Diese Kommandos werden eingeteilt in die *Data Definition Language* (DDL) zur Erzeugung und Verwaltung der Tabellen sowie in die *Data Manipulation Language* (DML), mit deren Hilfe Daten angelegt, abgerufen oder gelöscht werden können. In der Praxis hat sich hierzu die Sprache SQL[1] durchgesetzt, die sowohl DDL- als auch DML-Anweisungen beinhaltet und als Standard von allen größeren relationalen DBMS unterstützt wird. SQL ist eine einfache, aber dennoch mächtige, nicht-prozedurale Sprache, bei der der Anwender in einer der Umgangssprache ähnlichen Form beschreibt, welche Daten bzw. Tabellen er abfragen oder manipulieren möchte. Der SQL-Compiler des DBMS übersetzt diese Anfragen in interne prozedurale Anweisungen, die dann auf die Datenbasis zugreifen und die Anweisung ausführen.

Ein typischer Datenzugriff mittels SQL erfolgt über ein `SELECT`-Kommando, das die Form

```
SELECT <Attributliste>
FROM <Tabellenliste>
WHERE <Bedingungen>;
```

hat und genau die drei beschriebenen Grundoperationen bereitstellt.

Der Befehl `SELECT <Attributliste>` stellt hier die Projektion dar, der Join wird durch die Angabe von mehreren Tabellen in der `FROM`-Klausel realisiert. Mit Hilfe der optionalen `WHERE`-Anweisung können beliebige Zeilen anhand der Bedingungen selektiert werden.

4.2 Entity-Relationship-Diagramme

Zur Vereinfachung von Datenbankentwürfen existiert ein graphisches Datenmodell, das *Entity-Relationship-Modell* (ER-Modell). Ein Entwurf im ER-Modell legt die wichtigsten Datenstrukturen in systematischer und redundanzarmer Weise fest, ohne dabei auf die Technologie des später eingesetzten Datenbanksystems einzugehen. Die Ursprünge des ER-Modells gehen auf eine Arbeit von P. Chen [3] im Jahre 1976 zurück. Hieraus haben sich jedoch viele verschiedene Variationen mit leicht abweichenden Bestandteilen und Notationen entwickelt. In der Praxis werden heute *erweiterte Entity-Relationship-Modelle* (EER-Modelle) verwendet.

Kernbestandteile eines ER-Modells sind *Objekte* (*Entities*) und die *Beziehungen* (*Relationships*) zwischen ihnen. Chen definiert Objekte als Dinge der realen Welt, die

[1] Structured Query Language.

eindeutig identifiziert werden können. Alle ähnlichen, vergleichbaren oder zusammengehörigen Objekte werden zu einem *Objekttyp* zusammengefaßt. Wie später auch im Relationenmodell werden schon Objekttypen mit entsprechenden Attributen versehen, wobei eine Teilmenge der Attribute den Charakter eines Schlüssels hat (vgl. Abschnitt 4.1). Zusammen mit seinen Attributen beschreibt ein Objekttyp somit den unveränderlichen abstrakten Aufbau, den jedes Objekt dieser Art besitzt.

Beziehungen sind sinnvolle Wechselwirkungen, die zwischen den Instanzen von zwei oder mehr Objekttypen definiert sind. Sie können wie Objektarten mit eigenen Attributen versehen werden, die ihre Eigenschaften beschreiben.

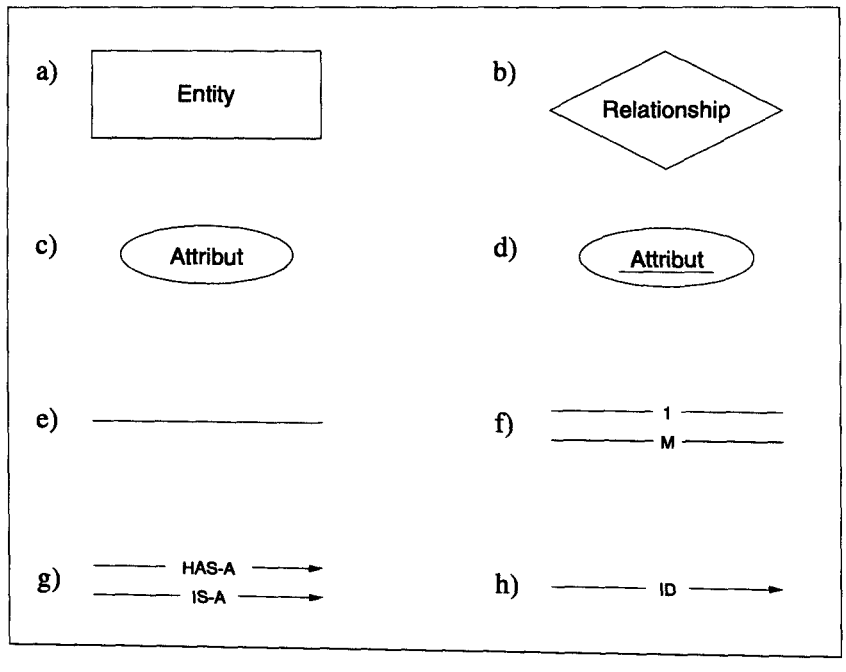

Abbildung 4.1: Diagrammsymbole im ER- und EER-Modell

Abbildung 4.1 zeigt die verschiedenen Diagrammsymbole für die Elemente eines ER-Diagramms: Objekttypen werden durch Rechtecke (a), Beziehungen durch Rauten (b) repräsentiert. Attribute werden als Ovale an das entsprechende Element angefügt (c). Bei Schlüsselattributen wird die Bezeichnung zusätzlich unterstrichen (d). Aus Gründen der Übersichtlichkeit wird jedoch bei einer großen Zahl von Attributen häufig auf ihre graphische Angabe verzichtet. Objektarten und Beziehungen können durch ungerichtete Kanten in Zusammenhang gebracht werden (e).

Im EER-Diagramm können diese Kanten um Kardinalitäten ergänzt werden, durch die festgelegt wird, mit wie vielen anderen Objekten ein Objekt in Beziehung stehen soll oder darf (f). Hier können durch *HAS-A*- und *IS-A-Beziehungen* auch Super- bzw. Subtypenbeziehungen zwischen einzelnen Objekttypen modelliert werden. Diese werden durch gerichtete Kanten zwischen verschiedenen Objekttypen zum Ausdruck gebracht (g). Durch gerichtete *ID*-Kanten können die Schlüsselattribute eines

Objekttyps in einen anderen Objekttyp übernommen werden (h). Sämtliche Kanten können mit einer zusätzlichen *Mandatory*-Eigenschaft versehen werden, durch die die Verbindung als obligatorisch gekennzeichnet wird.

Der Entwurf im (E)ER-Modell kann systematisch in das relationale Datenmodell überführt werden. Stark vereinfacht dargestellt entsteht dabei aus einem Objekttyp jeweils eine Tabelle; Beziehungen werden je nach Kardinalität in eigene Tabellen überführt oder ihre Attribute einer der zugeordneten Relationen hinzugefügt.

Für eine genauere Beschreibung der ER-Diagramme wie auch des Relationenmodells sei auf [15, 28] verwiesen.

4.3 Entwurf

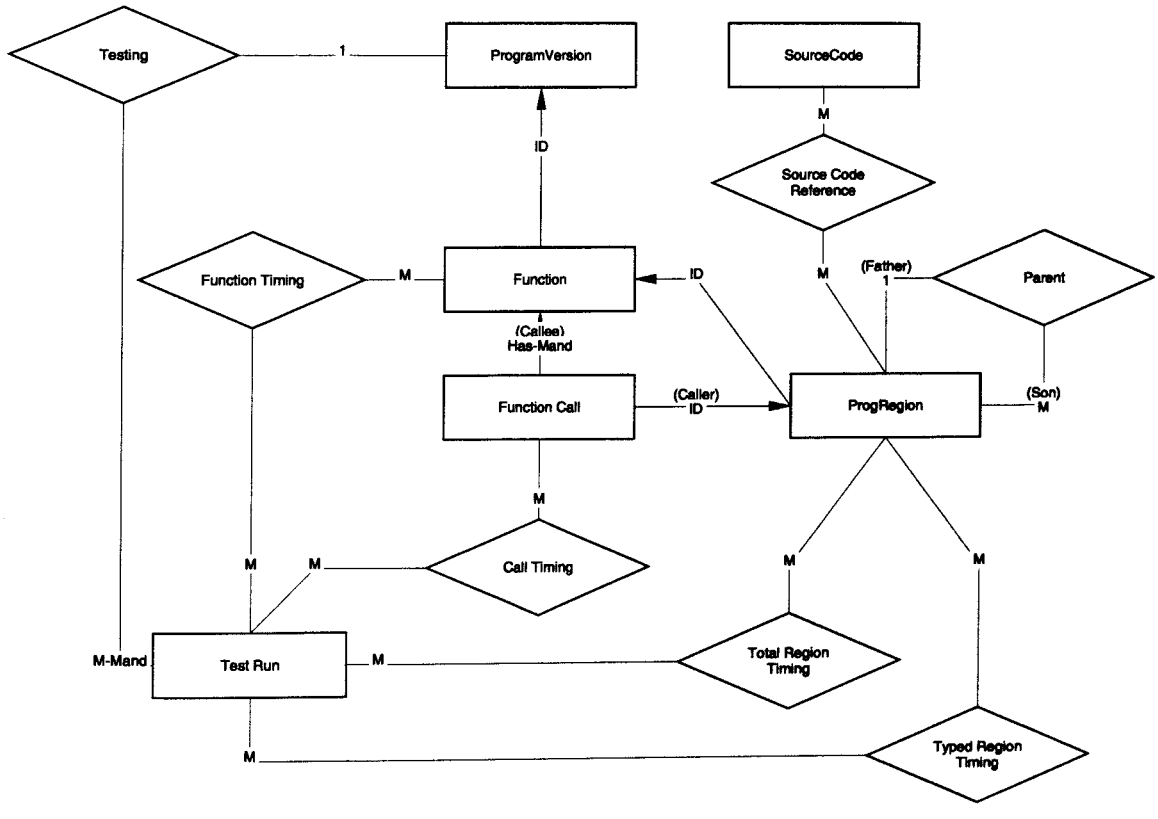

Abbildung 4.2: Erweitertes Entity-Relationship-Diagramm des Datenbankentwurfs

Im folgenden wird das Konzept der Datenbank zur Aufnahme der Apprentice-Leistungsdaten vorgestellt. Der Entwurf erfolgt zunächst auf abstrakter Ebene durch ein EER-Modell, das anschließend in das relationale Datenmodell übersetzt wird.

Zur Erstellung des EER-Diagramms wurde der von E. Szeto und V. Markowitz entwickelte Editor ERDRAW [27] eingesetzt. Dieses Werkzeug ermöglicht neben dem Design des Diagramms die Überprüfung der Konsistenz sowie eine graphische Ausgabe des Entwurfs. Des weiteren ist eine direkte Umsetzung in das Relationenmodell mit einer Optimierung von redundanten Relationen und Attributen implementiert, bei der die entsprechenden SQL-Anweisungen zur Anlage der Relationen für die gängigsten Datenbanksysteme generiert werden können.

Das Ziel einer automatischen Speedup-Analyse stellt an das Datenmodell die Anforderung zur Darstellung von

- verschiedenen Programmen,
- verschiedenen Testszenarien eines Programms,
- Quelltexten,
- Basisblöcken und Programmregionen mit Referenzen auf ihren Quelltext sowie
- Zeitinformationen.

Im Zusammenhang mit den im gegebenen Fall vorliegenden Apprentice-Leistungsdaten ergibt sich daraus der in Abbildung 4.2 dargestellte Datenbankentwurf. Die Wahl der Objekttypen und Beziehungen wurde dabei stark von den in Apprentice verwendeten Datenstrukturen beeinflußt, um alle vorliegenden Daten speichern zu können.

Durch die erforderliche Möglichkeit eines vergleichenden Zugriffs auf unterschiedliche Testläufe lassen sich die modellierten Objekte und Beziehungen in zwei Gruppen einteilen:

- Die *testlaufunabhängigen* Objekte enthalten alle statischen Daten, die sich in den verschiedenen Testläufen nicht ändern. Alle diese Elemente haben also gemeinsam, daß sie keine Informationen zu gemessenen Zeiten beinhalten. Im einzelnen fallen in diese Gruppe die Daten zum Programm selbst sowie die Daten zu den Funktionen, Funktionsaufrufen und Programmregionen (vgl. Tabelle 4.1).
- In den *testlaufabhängigen* Relationen werden die gemessenen Zeitdaten der einzelnen Testläufe gespeichert. Diese Gruppe umfaßt alle übrigen Objekte und Beziehungen des Entwurfs (vgl. Tabelle 4.2).

Ausgehend von diesen Gruppen baut der Entwurf auf den zwei Basisrelationen `ProgramVersion` und `Test Run` auf, die die jeweiligen Informationen der verfügbaren Programmversionen und Testläufe enthalten. Die eindeutige Identifizierung der Datensätze erfolgt für die Programmversionen durch eine ID-Nummer, obwohl eine Kombination von Programmname und -datum ebenso ein Schlüsselkandidat wäre.

Objekttyp/Beziehung	Inhalt
`ProgramVersion`	Programmversionen mit Namen, Datum und ID
`ProgRegion`	Programmregionen mit Namen in der von Apprentice vorgegebenen Form
`Function`	Funktionen mit Namen und Instrumentierungskennzeichen
`Function Call`	Funktionsaufrufe, identifiziert durch die Call-ID
`Parent`	Beziehung zwischen zwei Objekten von `ProgRegion`
`SourceCode`	Vollständiger Quelltext mit Dateiname, Pfad und Datum
`Source Code Reference`	Beziehung zwischen Programmregionen und Quellcodes, Zuordnung durch Angabe der ersten und letzten Programmzeile

Tabelle 4.1: Testlaufunabhängige Elemente des Datenbank-Entwurfs

Dies erzeugt zwar in der `ProgramVersion`-Relation einen geringen Overhead, der Primärschlüssel von `ProgramVersion` setzt sich jedoch durch alle anderen Relationen als Sekundärschlüssel fort, so daß ein zusätzliches, kürzeres Attribut hier sinnvoll ist.

Zum eindeutigen Zugriff auf die Testläufe ist diese Vorgehensweise nicht erforderlich. Hier dienen Datum und Uhrzeit als Primärschlüssel. Dies ist im Vergleich zu einem kurzen ID zwar ein längerer Schlüssel, der Einfluß auf andere Relationen ist jedoch nicht so groß wie im Falle der Programmversionen.

Zentrales Element des Entwurfs bildet die Relation der Programmregionen, die aufgrund ihrer Bedeutung in Apprentice mit den meisten anderen Objekttypen und Beziehungen in direktem Zusammenhang steht. Durch die Basisblockeigenschaft werden alle Zeitmessungen auf ein Objekt dieser Relation bezogen. Darüber hinaus läßt sich ein direkter Bezug zu einzelnen Quelltextzeilen ermitteln. Zur Abstraktion des hierarchischen Apprentice-Programmbaumschemas (vgl. Abbildung 3.5) verweist die Relation über die Vater/Sohn-Beziehung `Parent` sogar auf sich selbst. Über `ProgRegion` kann durch diese enge Vernetzung ein einfacher Zugriff auf alle gemessenen Daten, auf Funktionsaufrufe sowie auf die Quellcodedateien erfolgen.

Die Modellierung der Funktionen und ihrer Aufrufe ist durch die starke Anlehnung an die in Apprentice verwendeten Strukturen komplexer. Analog zu Apprentice wird eine Funktion sowohl in `Function` als auch in `ProgRegion` aufgenommen. Jedes Ob-

Objekttyp/Beziehung	Inhalt
`Test Run`	Testläufe mit Datum, Anzahl der Prozessoren
`Timing`	Zusammenhang zwischen Programmen und Testläufen
`Call Timing`	Zeiten für Funktionsaufrufe
`Function Timing`	Zeiten für Funktionen
`Total Region Timing`	Aufsummierte Zeiten für Programmregionen
`Typed Region Timing`	Zeiten für Programmregionen, aufgeteilt nach Apprentice-Zeittypen

Tabelle 4.2: Testlaufabhängige Elemente des Datenbank-Entwurfs

ProgramVersion (E)		
Compilation	datetime	NO NULLS
ProgId	varchar(10)	NO NULLS (1)
Name	varchar(255)	NO NULLS
Function (E)		
instrumented	bit	NO NULLS
Name	varchar(255)	NO NULLS (1)
ProgRegion (E)		
Name	varchar(255)	NO NULLS (1)
Test_Run (E)		
testtime	datetime	NULLS ALLOWED (1)
pe	smallint	NULLS ALLOWED
clockspeed	float	NULLS ALLOWED
Function_Call (E)		
Call_ID	int	NO NULLS (1)
Function_Timing (R)		
num_called_from	float	NO NULLS
passes	int	NO NULLS
SourceCode (E)		
FileName	varchar(255)	NO NULLS (1)
FileDate	datetime	NO NULLS (1)
Code	text	NULLS ALLOWED
Total_Region_Timing (R)		
excl	float	NO NULLS
incl	float	NO NULLS
ovhd	float	NO NULLS
est_clocks	float	NO NULLS
Source_Code_Reference (R)		
beginrange	int	NO NULLS (1)
endrange	int	NO NULLS (1)
Call_Timing (R)		
ncall_min	float	NO NULLS
ncall_min_pe	float	NO NULLS
ncall_max	float	NO NULLS
ncall_max_pe	float	NO NULLS
ncall_mean	float	NO NULLS
ncall_stdev	float	NO NULLS
time_min	float	NO NULLS
time_min_pe	float	NO NULLS
time_max	float	NO NULLS
time_max_pe	float	NO NULLS
time_mean	float	NO NULLS
time_stdev	float	NO NULLS
Typed_Region_Timing (R)		
type	int	NO NULLS (1)
timing	float	NULLS ALLOWED

Abbildung 4.3: Attribute des EER-Diagramms

jekt aus `ProgRegion` wird jedoch, wie in 3.5.2 schon geschildert, nicht nur durch seinen Namen, sondern zusätzlich auch durch die umgebende Funktion identifiziert. Daher wird der Schlüssel von `Function` zum Schlüssel von `ProgRegion` hinzugenom-

men. Dies wird im EER-Diagramm durch die ID-Verbindung zwischen `Function` und `ProgRegion` modelliert. Zusätzlich existiert zur Modellierung von Funktionsaufrufen über den Objekttyp `Function Call` eine weitere Verbindung zwischen der aufgerufenen Funktion und den aufrufenden Programmregionen.

Die Speicherung der gemessenen Zeiten erfolgt über die Relationen `Function Timing`, `Call Timing`, `Total Region Timing` und `Typed Region Timing`. Da in Apprentice für jede Programmregion entsprechende Zeiten vorliegen, wird zu jeder Region und zu jedem Testlauf eines Programms jeweils ein Datensatz in diese Relationen eingefügt. Ausnahme bilden lediglich die Informationen über die Daten der einzelnen Zeittypen, da diese nur in die Apprentice-Exportdatei geschrieben werden, wenn sie größer als Null sind. Es wird folglich nur ein Datensatz für einen Zeittyp in die Datenbank aufgenommen, falls Apprentice in der betreffenden Region einen positiven Wert für diesen Typ gemessen hat.

In Abbildung 4.3 sind die einzelnen Attribute der Elemente des EER-Diagramms mit ihren Datentypen im von ERDRAW generierten Ausgabeformat aufgeführt. Beziehungen, die nicht angegeben sind, besitzen keine eigenen Attribute. Schlüsselattribute sind durch eine `(1)` gekennzeichnet. Zusätzlich sind Angaben zum Verhalten bei leeren Datenfeldern („Nullfeldern") angegeben.[2] Während bei einigen Feldern das Fehlen eines Inhaltes durchaus sinnvoll ist, kann falls erforderlich ein Feldinhalt erzwungen werden. Dies wird durch die `NOT NULL`-Eigenschaft festgelegt. Bei Schlüsselfeldern ist diese Angabe obligatorisch.

Bei Generierung der SQL-Anweisungen, die die entsprechenden Tabellen in einer relationalen Datenbank erzeugen, werden durch ERDRAW einige Vereinfachungen vorgenommen. So werden die Beziehungen `Timing` und `Parent` nicht mehr benötigt, da ihre Informationen mit in die Objekttypen `Test Run` und `Prog Region` aufgenommen werden können. Alle weiteren Tabellen entsprechen genau den in Abbildung 4.2 dargestellten Elementen mit den Attributen aus Abbildung 4.3. Die von ERDRAW generierten SQL-Kommandos werden in Anhang C detailliert beschrieben.

Durch die Konvertierung in das Relationenschema ergeben sich eine Reihe von eng verknüpften Tabellen. Abbildung 4.4 zeigt eine Auswahl dieser Tabellen mit ihren Verknüpfungen und einigen Beispiel-Datensätzen. Hier ist auch erkennbar, wie `Timing` und `Parent` in `Test Run` und `Prog Region` integriert wurden.

4.4 Oracle

Die Performance-Datenbank des KOJAK-Projektes wurde auf Basis des im Forschungszentrum Jülich verfügbaren relationalen Datenbanksystems Oracle 7 auf einer IBM RS/6000-G40, einem Dual-Prozessor-System mit 512 MByte Hauptspei-

[2] Das Relationenmodell unterscheidet zwischen der Zahl 0 und dem Wert `NULL`, der ein leeres bzw. nicht initialisiertes Feld beschreibt.

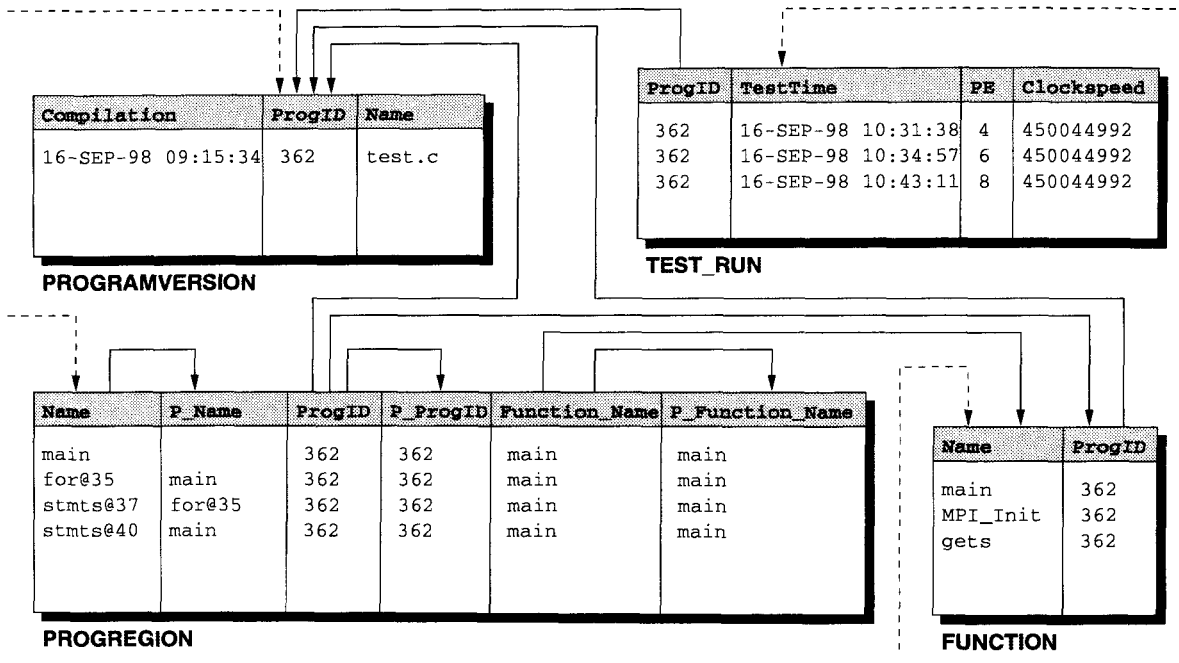

Abbildung 4.4: Tabellen des Datenbank-Entwurfs

cher, realisiert. Oracle hat sich in der Vergangenheit als Standard etabliert und läuft in einer Client/Server-Umgebung mit Clients auf den verschiedensten Betriebssystemen. Zu allen gängigen Programmiersprachen existieren Schnittstellen, so daß die Erweiterungsmöglichkeiten eines Systems zur automatischen Performance-Analyse auch über das experimentelle Stadium hinaus gewährleistet sind.

Da ein Oracle-Server in jeder größeren Umgebung zur Verfügung stehen sollte, stellt die Wahl dieses speziellen Systems nur eine geringe Einschränkung des entwickelten Werkzeugs dar. Dennoch wurden sämtliche Zugriffe auf die Datenbank durch standardisierte Schnittstellen implementiert, so daß eine Unterstützung weiterer Datenbanksysteme leicht integrierbar ist.

Durch die interaktive Schnittstelle *SQL*Plus* kann der Anwender SQL-Kommandos an das DMBS übermitteln. SQL*Plus unterstützt dabei die über den Standard von SQL herausgehende Spracherweiterung PL/SQL. Diese Sprache wurde um einige prozedurale Elemente erweitert, so daß z. B. die Steuerung von Mehrfachabfragen mittels Schleifen möglich ist.

Mit der Import-Schnittstelle *SQL*Loader* können Daten aus externen Dateien in die Datenbank übernommen werden. Der Importvorgang wird dabei von einer Kontrolldatei gesteuert, in der das Format der Eingabedatei beschrieben wird. Die Eingabefelder müssen dabei durch Trennzeichen voneinander abgegrenzt sein oder eine feste Länge besitzen. Die Kontrolldatei beinhaltet Definitionen der folgenden Form:

```
LOAD DATA
INFILE <Import-Datei>
INTO TABLE <Tabelle>
```

```
WHEN <Bedingung>
(<Felddefinitionen>)
INTO TABLE ...
```

Mittels der **WHEN**-Anweisung ist SQL*Loader in der Lage, Daten zu filtern und nur bei Erfüllung festgelegter Kriterien in den Datenbestand einzufügen. Ebenso besteht die Möglichkeit, durch die Angabe mehrerer **INTO TABLE**-Blöcke die Daten in mehrere Tabellen zu übernehmen. Zum jeweils betrachteten Datensatz werden dann in Abhängigkeit der **WHEN**-Bedingung für jeden solchen Block die entsprechenden Felder in die Tabelle eingefügt.

In Fehlersituationen wird der betreffende Datensatz entweder ignoriert oder zurückgewiesen. Aus der Sicht des Anwenders sind diese Verhaltensweisen zwar identisch (der Datensatz wird nicht eingefügt), dennoch unterscheidet SQL*Loader streng zwischen diesen beiden Varianten. Stimmen die in der Kontrolldatei festgelegten Felddefinitionen nicht mit den aus der Importdatei gelesenen Werten überein, so wird der Datensatz ignoriert. Sind hingegen alle Definitionen korrekt und der Versuch, die Daten einzufügen, wird vom Oracle-Server abgelehnt, so wird der Datensatz zurückgewiesen. Werden zu viele Sätze zurückgewiesen, so bricht SQL*Loader den Importvorgang ab.

4.5 Datenübernahme aus Apprentice

Die Übernahme der Daten sollte aus Gründen der Erweiterbarkeit und der Organisation der einzelnen Komponenten von KOJAK (vgl. Abbildung 2.1) nicht direkt aus dem Analyse-Werkzeug heraus, sondern durch den Auto Executer erfolgen.

Eine automatische Ausführung des zu analysierenden Programms mit anschließender Übernahme der Leistungsdaten ist jedoch zur Entwicklung eines Grundbausteins nicht von näherem Interesse. Da die generelle Funktionsweise der Datenbank sowie des entwickelten Werkzeugs hiervon unabhängig ist, soll im Rahmen dieser Arbeit auf den Auto Executer verzichtet werden. Die Datenübernahme erfolgt stattdessen mittels manuell generierter, vorbereiteter Exportdateien.

Oracle besitzt eine große Anzahl verfügbarer Schnittstellen zum Datenimport. Nachstehend sollen die wichtigsten prinzipiell geeigneten Verfahren kurz geschildert werden.

- Die Verwendung der in 4.4 beschriebenen Import-Schnittstelle **SQL*Loader** ist ohne weiteres nur eingeschränkt möglich. Es besteht zwar generell die Möglichkeit, Datensätze in der von Apprentice generierten Form direkt einzulesen, ein Ignorieren oder Überspringen von ausgegebenen Feldern ist bei Verwendung von Feldern variabler Länge jedoch nicht möglich. Es können lediglich ganze Datenfelder gefiltert werden. Da die einzelnen Datensätze, bzw.

Teile von ihnen, wie bereits erwähnt mehrere Relationen beeinflussen können, führt dies zu Problemen bei der Erkennung der Felder. Abhilfe könnte eine weitergehende Modifikation von Apprentice schaffen, die die Datensätze genau im für SQL*Loader erforderlichen Format ausgibt. Dies würde jedoch zur Folge haben, daß gewisse Felder mehrmals ausgegeben werden müssen. Elementare Schlüsselfelder, wie die von `ProgramVersion` oder `ProgRegion` müßten sogar in nahezu jeder Zeile ausgewiesen werden. Eine zusätzliche Hürde bildet bei diesem Ansatz, daß Apprentice keinen Zugriff auf die bereits in der Datenbank befindlichen Daten hat. Es müßte also „blind" versucht werden, alle verfügbaren Werte an die Datenbank zu übergeben, da bei Bearbeitung eines neuen Testszenarios nicht bekannt ist, ob die testlaufunabhängigen Daten bereits früher korrekt eingelesen wurden. Da das Schreiben von bereits existierenden Daten gleichzeitig zu Zugriffsfehlern führt, kann diese Vorgehensweise bei Überschreiten einer Fehlergrenze den Import-Prozeß zum Abbruch zwingen.

- Die Nutzung der **Schnittstelle zu C++** oder anderen Programmiersprachen wurde nicht näher untersucht, da das Einlesen und Zerlegen der Datensätze sehr unkomfortabel ist und einen hohen Programmieraufwand für einen im wesentlichen unwichtigen Sachverhalt erfordern würde. Darüber hinaus würde die Verwendung einer Compiler-Sprache eine Systemabhängigkeit schaffen und die Verwendbarkeit der Import-Schnittstelle stark einschränken.

- Als Skriptsprache mit dem Charakter einer klassischen Programmiersprache bietet **Perl** durch die Unterstützung von Stringvergleichen mit regulären Ausdrücken sehr gute Möglichkeiten zur Zerlegung und Bearbeitung von Datensätzen der vorliegenden Form. Des weiteren existieren einfache Schnittstellen zu Netzwerkdiensten, so daß auch ein Zugriff auf vorbereitete Export-Dateien oder Quelltexte, die sich noch auf dem entfernten Rechner befinden, einfach zu implementieren ist. Durch DBI, eine standardisierte Datenbank-Schnittstelle, kann sowohl lesender als auch schreibender Zugriff auf eine Datenbank erfolgen. Somit können Programmversionen, Quelltexte und auch Testszenarien auf ihre Existenz geprüft werden, bevor sie in die Datenbank eingefügt werden. Allerdings ist der Datenbankzugriff mittels Perl sehr langsam.

- Der Datenbankzugriff über ein **Java-Applet** bietet in bezug auf Plattformunabhängigkeit und einfache Netzwerknutzung ähnliche Vorteile wie Perl. Dennoch ist auch Java für den Import nur bedingt geeignet, da der Zugriff auch hier nur mit mäßiger Geschwindigkeit möglich ist. Der Start innerhalb von Skripten ist bei Implementierung als Applet nicht möglich. Dies verhindert eine spätere Integration in automatische Systeme.

Die vom modifizierten Apprentice generierten Exportdateien nehmen aufgrund der Vielzahl von verschiedenen Objekttypen sehr schnell eine beachtliche Größe an. Bereits die Datei zum Beispiel-Programm aus Abbildung 3.4 hat eine Größe von 110 Zeilen. Da eine Zeile unter Umständen in mehrere Relationen einfließt, können sich

bei größeren Programmen schnell mehrere tausend einzufügende Datensätze ergeben, so daß der Datenimport bei den geringen Geschwindigkeiten der Datenbankschnittstellen einen empfindlichen Engpaß darstellt.

Tabelle 4.3 zeigt den Vergleich eines Datenbankzugriffs mit DBI und SQL*Loader. Hierzu wurden zufällig generierte Beispieldaten in eine Relation mit drei Attributen geschrieben.

Anzahl Datensätze	SQL*Loader	DBI
100	0.63 sec	8 sec
500	1.37 sec	38 sec
1000	2 sec	75 sec
5000	16 sec	371 sec

Tabelle 4.3: Zugriffszeiten für den Datenimport

Zur Übernahme der Apprentice-Daten wurde daher zur Ausnutzung des hohen Datendurchsatzes von SQL*Loader und der Flexibilität von Perl [29] eine Mischung dieser beiden Möglichkeiten implementiert. Das Perl-Skript `kojakimport` fordert den Anwender nach dem Aufruf, der ohne weitere Parameter erfolgt, zunächst auf,

- Benutzername und -paßwort der Datenbank,
- Rechnername, auf dem die Apprentice-Exportdateien und die Quelltexte bereitliegen,
- Benutzername und -paßwort auf diesem Rechner sowie
- Pfad und Dateiname der Apprentice-Exportdatei

einzugeben. Sind diese Daten eingelesen, so wird nach erfolgreichem Verbindungsaufbau zur Datenbank und zum entfernten Rechner die Reportdatei übertragen und eingelesen. Es folgt eine zeilenweise Auswertung der Daten, bei der basierend auf dem Zeichen in der ersten Spalte die in die Datenbank einzufügenden Datensätze bestimmt werden. Handelt es sich bei der betreffenden Relation um eine Relation mit voraussichtlich wenigen neuen Datensätzen, wie es bei neuen Programmversionen oder Testläufen der Fall ist[3], so wird der Datensatz direkt über DBI in die Datenbank eingefügt. In dieser Kategorie werden ebenfalls Funktionen behandelt, da im allgemeinen angenommen wird, daß die Anzahl der Funktionen wesentlich geringer ist als die der Programmregionen oder Zeitdaten. Alle übrigen Datensätze werden in eine neue Datei geschrieben, die dann später von SQL*Loader bearbeitet wird. In Anhang B sind neben den Felddefinitionen der Exportdatei auch die jeweils betroffenen Datenbankrelationen beschrieben.

[3]Diese Relationen können durch die Struktur der Exportdatei je Importlauf nur einmal vorkommen.

Diese Vorgehensweise vereint die Vorteile von Perl und SQL*Loader, indem einerseits bei Relationen mit vielen Einfügungen die Geschwindigkeit der direkten Datenbankschnittstelle ausgenutzt wird, andererseits jedoch die Größe der Konfigurationsdatei von SQL*Loader durch Relationen, die selten berührt werden, nicht unnötig vergrößert wird. So wird zwar das Einfügen der einzelnen Datensätze teurer, jedoch wird die Bearbeitungszeit erheblich reduziert, da die Konfigurationsdatei von SQL*Loader bei jedem einzufügenden Datensatz vollständig durchsucht werden muß.

Das beschriebene Verfahren beruht auf Zeitmessungen, die mittels einer Test-Exportdatei, aus der ca. 4000 einzufügende Datensätze resultieren, durchgeführt wurden. Die alleinige Verwendung von Perl benötigte dabei über 15 Minuten, was offensichtlich nicht akzeptabel ist. Ein Einfügen aller Daten über SQL*Loader – nach einer Vorverarbeitung durch ein Perl-Skript – dauerte nur noch drei Minuten. Durch die geschilderte Verkürzung der Konfigurationsdatei konnte die Bearbeitungszeit schließlich auf zwei Minuten reduziert werden. Alle Messungen wurden auf einer Sun SPARCstation 4 durchgeführt und konnten auf der zentralen IBM R50 des Zentralinstituts für Angewandte Mathematik in ihrer Größenordnung bestätigt werden.

Auch wenn die Zeiten für den Datenimport durch das dargestellte Verfahren erheblich reduziert werden können, so kann diese Komponente in der beschriebenen Rechnerkonfiguration beim Import mehrerer Testläufe eines großen Programms immer noch einen kritischen Engpaß darstellen. Die Untersuchung weiterer Datenbanksysteme wäre sicherlich sinnvoll, würde aber den Rahmen dieser Arbeit sprengen.

Kapitel 5

Schnittstellen zur Client-Server-Programmierung

Die Auswahl der Programmiersprache und -umgebung für das zu entwickelnde Analyse-Werkzeug orientiert sich vor allem an den für die gewählte Oracle-Datenbank verfügbaren Schnittstellen. Die klassischen Schnittstellen sind jedoch aus der Sicht des Anwenders bzw. Entwicklers sehr ähnlich. Sie unterscheiden sich häufig nur in der Art ihrer Implementierung sowie in ihrer Geschwindigkeit. Die Vorgehensweise bei der Erstellung einer Verbindung zum Oracle-Server sowie bei der Generierung und Übermittlung einer Abfrage ist jedoch vergleichbar.

In diesem Kapitel sollen einige dieser herkömmlichen Schnittstellen, aber auch neue Ansätze des Datenbankzugriffs, vorgestellt werden.

5.1 Oracle Call Interface

Das *Oracle Call Interface* (OCI) ist eine Datenbankschnittstelle, durch die die nicht-prozeduralen Vorzüge von SQL in prozedurale Hochsprachen integriert werden können. Datenbankanfragen werden dabei in derselben Form wie auch bei der interaktiven Datenbank-Schnittstelle SQL*Plus formuliert. OCI unterstützt den vollen SQL-Sprachumfang relationaler Datenbanksysteme sowie die Erweiterungen von PL/SQL.

Auf diese Weise können sehr mächtige und flexible Anwendungen entwickelt werden, die die Fähigkeiten von reinen SQL-Anweisungen übersteigen. Das Übersetzen und Binden einer OCI-Anwendung weicht dabei, wie Abbildung 5.1 veranschaulicht, nicht von der Vorgehensweise bei einer „normalen" Anwendung ohne Datenbankzugriffe ab.

OCI-Unterstützung existiert für Oracle 7 in Form von Bibliotheken für die Programmiersprachen Ada, C/C++, COBOL, Fortran und PL/I.

Eine Datenabfrage innerhalb einer OCI-Anwendung gliedert sich in die folgenden Schritte (vgl. [23]):

1. Definition der OCI Datenstrukturen und Cursor.
2. Erstellen der Verbindung zum Oracle-Server.
3. Öffnen der benötigten Cursor zur Verarbeitung der SQL-Statements.
4. Ausführen der SQL-Anweisung und Abrufen der selektierten Datensätze.
5. Schließen der Cursor.
6. Schließen der Verbindung zur Datenbank.

Die Cursor dienen dabei der Verarbeitung von Ergebnissen, die mehr als einen Datensatz umfassen. Solche Abfragen werden in vorbereitete Arrays ausgegeben.

Für jeden der beschriebenen Schritte sowie für sonstige administrative Aufgaben stellen die OCI-Bibliotheken in den einzelnen Programmiersprachen vielfältige Funktionen und Prozeduren bereit. Da eine nähere Beschreibung an dieser Stelle zu umfangreich wäre, sei auf [23] verwiesen.

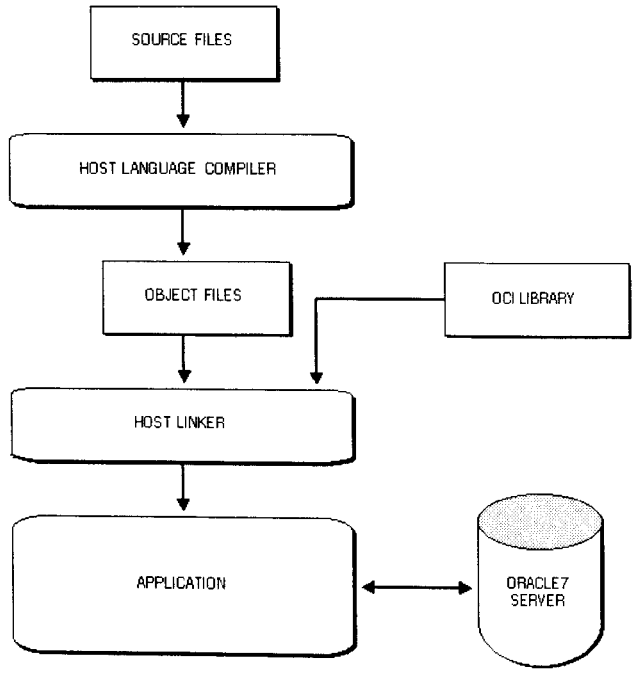

Abbildung 5.1: Entwicklung von OCI-Anwendungen (Quelle: [23])

5.2 Oracle Precompiler

Die Verwendung eines Precompilers erlaubt wie OCI den Einsatz von SQL-Statements in prozeduralen Programmiersprachen. Der Unterschied zu OCI-Anwendungen besteht darin, daß die Verwendung von SQL-Anweisungen durch Makros einfacher zu handhaben ist.

Der Ablauf einer Datenabfrage erfolgt in derselben Form wie beim Oracle Call Interface. Durch den bei der Übersetzung vorgeschalteten Präprozessor besteht jedoch die Möglichkeit, die komplexen Funktionsaufrufe zur Steuerung der Datenbank durch einfachere eingebettete SQL-Kommandos der Form

```
EXEC SQL ...
```

zu ersetzen.

Diese eingebundenen SQL-Makros werden dann durch den Precompiler in entsprechende Aufrufe der Oracle-Laufzeitbibliothek (SQLLIB) übersetzt. Dadurch verlängert sich, wie Abbildung 5.2 veranschaulicht, der Übersetzungsprozeß um

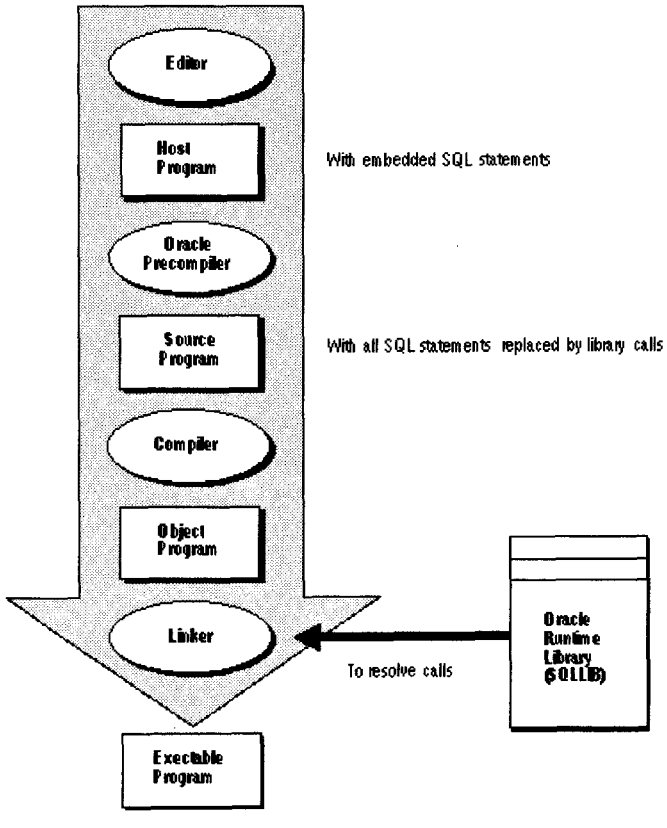

Abbildung 5.2: Datenbank-Anwendungen mit eingebetteten SQL-Anweisungen (Quelle: [24])

einen zusätzlichen Schritt. Dies erspart dem Anwender jedoch die manuelle Generierung der Aufrufe.

Oracle Precompiler sind für die Programmiersprachen C bzw. C++, Fortran sowie COBOL verfügbar (vgl [24]).

5.3 Java Database Connectivity

Für den Zugriff aus Java werden von Oracle Datenbanktreiber nach dem *JDBC*-Standard (Java Database Connectivity) bereitgestellt. Dabei existieren mit *JDBC OCI* und *JDBC Thin* zwei Varianten von Datenbanktreibern:

- Der **JDBC OCI**-Treiber basiert auf der bereits beschriebenen OCI-Unterstützung, wie sie auch in den klassischen Programmiersprachen Anwendung findet. Diese Variante setzt eine lokale Installation des Oracle-Clients zwingend voraus und ist somit plattformabhängig. Leider stehen die Treiber nicht auf allen Betriebssystemen zur Verfügung. Ein Einsatz ist auch nur in Java Applikationen möglich, da bei Web-basierten Applets die Existenz eines Oracle-Clients auf dem startenden Rechner nicht vorausgesetzt werden kann.
- Der **JDBC Thin**-Treiber verwendet zur Verbindung zur Datenbank eigene Java-Sockets. Dadurch ist diese Methode plattformunabhängig und nicht auf eine bestehende Client-Installation angewiesen. Ein Einsatz ist daher sowohl in Applikationen als auch in Applets möglich. Der ausführende Rechner muß lediglich Zugriff auf die entsprechenden JDBC-Klassen haben.

Die JDBC-Treiber sind ab Java Development Kit (JDK) 1.0.2 verfügbar. Hinsichtlich der Anwendung in Java-Programmen unterscheiden sich die beiden Varianten nur geringfügig. Für den Zugriff auf die Datenbank muß zunächst der korrekte Datenbanktreiber durch Aufruf der Funktion `registerDriver` registriert werden. Danach kann mittels `getConnection` eine Verbindung erstellt werden. Diese wird im weiteren Ablauf durch ein Objekt der Klasse `Connection` repräsentiert. Eine SQL-Abfrage gliedert sich dann in zwei Schritte:

1. Erstellen und Absenden der SQL-Abfrage. Als Rückgabewert wird ein Objekt der Klasse `ResultSet` geliefert.
2. Durch das `ResultSet` kann dann das Ergebnis der Abfrage ausgelesen werden. Sollte das Ergebnis aus mehreren Datensätzen bestehen, so müssen diese einzeln abgerufen werden. Hierzu dient die Funktion `next` der Klasse `ResultSet`.

Die Organisation der erforderlichen Cursor, die bei den bisher vorgestellten Schnittstellen vor der Ausführung einer SQL-Abfrage deklariert werden müssen, wird hier

vollständig von den JDBC-Klassen übernommen und bleibt für den Anwender verdeckt.

Das separate Abrufen der einzelnen Ergebnis-Datensätze führt bei großen Abfragen in der Praxis häufig zu Engpässen. Aus diesem Grund wurde in die Oracle-Treiber die Option eines *Prefetching* integriert, mit der, vergleichbar zum Array-Fetching des OCI, mehrere Datensätze gleichzeitig ausgelesen werden können. Diese Funktion ist jedoch im JDBC-Standard nicht vorgesehen und erfordert eine geringfügige Oracle-spezifische Anpassung des Java-Programms. Ohne eine solche Anpassung wird ein voreingestellter Wert von zehn Datensätzen gewählt.

Für das im Rahmen dieser Arbeit eingesetzte Datenbanksystem Oracle 7 existiert nur eine sehr rudimentäre Umsetzung der JDBC-Klassen, da Oracle seine Entwicklungen bezüglich Java auf Oracle 8 konzentriert. Dennoch beinhalten die JDBC-Klassen für Oracle 7 den vollen Umfang der im JDBC-Standard definierten Funktionen. Lediglich eine Optimierung der Zugriffszeiten ist nicht implementiert.

5.4 CORBA/Java

Die von der *Object Management Group* (OMG) entwickelte *CORBA*-Spezifikation (Common Object Request Broker Architecture) soll die Idee realisieren, daß kleine, aufgabenspezifische Software-Objekte in einem Netzwerk verteilt verfügbar sind und ohne Abhängigkeit von Betriebssystemen oder Programmiersprachen zusammenarbeiten können. CORBA definiert einen objektorientierten Ansatz, bei dem Software-Komponenten wiederverwendet und von verschiedenen Anwendungen gemeinsam eingesetzt werden können.

Durch die Plattformunabhängigkeit kann ein CORBA-Objekt von jedem Rechner aus abgerufen und gestartet werden. Die Sprachenunabhängigkeit gewährleistet, daß bereits existierende Objekte in jeder Entwicklungsumgebung wiederverwendet werden können.

Zur Kommunikation zwischen Anwendungen und Objekten bzw. zwischen Objekten untereinander beinhaltet CORBA mit IDL (*Interface Definition Language*) eine sprachenneutrale Möglichkeit, ein Objekt und seine Schnittstelle zu beschreiben. IDL ist eine rein deklarative und auf verteilte Objekte zugeschnittene Sprache. Die Syntax besteht aus einem Subset von C++ mit zusätzlichen Schlüsselwörtern, die verteilte Konzepte unterstützen. Die Sprache, in der das CORBA-Objekt implementiert ist, ist dabei irrelevant. Die Entwicklung von Objekten kann in jeder Programmiersprache erfolgen, für die CORBA-Bindungen bestehen.

Gemeinsam mit Java stellt CORBA somit ein sehr mächtiges System dar, in dem die Welt der portierbaren Programme mit der Welt der Web-Objekte verknüpft wird. Der Ablauf eines Zugriffs auf ein CORBA-Objekt über eine Java-Anwendung erfolgt in vier Schritten:

1. Der Browser lädt die HTML-Seite vom HTTP-Server.
2. Der Browser empfängt das Java Applet vom HTTP-Server.
3. Das Applet wird gestartet.
4. Das Applet ruft die CORBA-Server-Objekte ab.

Die Verbindung einer Anwendung mit einem CORBA-Objekt wird dabei durch den *Object Request Broker* (ORB) gesteuert, der eine Objektanforderung des Client entgegennimmt, den Server des Objektes sucht, die Anforderung sowie die Parameter an den ORB des Servers weitervermittelt und schließlich das Ergebnis empfängt. Zur Kommunikation zwischen einzelnen CORBA-Objekten und dem ORB wird das *Internet Inter-ORB Protocol* (IIOP) eingesetzt.

Ein CORBA-Objekt kann somit mittels IIOP sowie durch den Client- und Server-ORB die Dienste eines Objektes auf einem HTTP-Server beanspruchen. Dieses Objekt kann seinerseits wiederum auf weitere, dahinterliegende Dienste zurückgreifen. Hierdurch wäre auch der Zugriff auf ein relationales Datenbanksystem denkbar. Voraussetzung hierfür ist jedoch, daß das einzusetzende System über einen eigenen ORB verfügt.

Dies entspricht einer Client/Server-Architektur, bei der der Objektzugriff in drei Schichten erfolgt: Die erste Schicht repräsentiert die sichtbaren Eigenschaften eines Objektes, die mittlere Schicht enthält die statischen, beständigen Objektdaten und -funktionen. Die dritte Schicht besteht schließlich aus den dynamischen Objektdaten, die in Form von Datenbanken oder vergleichbaren Anwendungen vorliegen. Abbildung 5.3 verdeutlicht dieses Modell.

CORBA stellt durch die beschriebenen Strukturen einen Objektbus zur Verfügung, der Grundlage für die nächste Generation von Internet-Anwendungen ist. Wenn

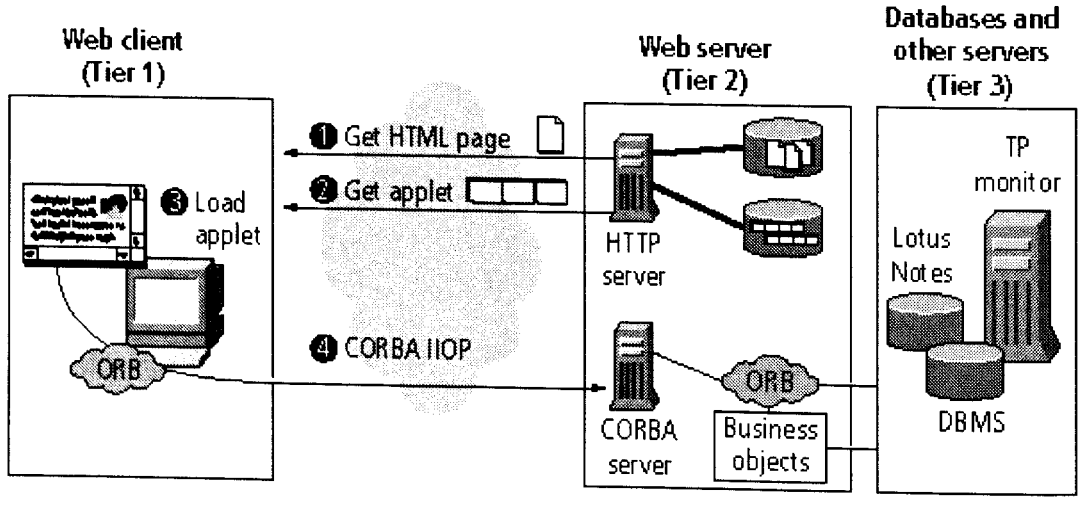

Abbildung 5.3: Die CORBA 3-Schichten-Architektur (Quelle: [20])

auch die OMG aus mehr als 700 Mitgliedern besteht und die Arbeiten auf diesem Gebiet schnell fortschreiten, so steckt die Entwicklung jedoch immer noch in den Anfängen. Die CORBA-Unterstützung wird von den beteiligten Herstellern nur in den neuesten bzw. künftigen Anwendungen implementiert.

Da Oracle 7 wie bereits in 5.3 beschrieben nur über eine eingeschränkte Java-Unterstützung verfügt, ist die Integration einer Schnittstelle, die die Verwendung von CORBA-Objekten erlaubt, nicht zu erwarten.

Eine präzisere Darstellung des CORBA/Java-Konzepts ist in [20, 25] beschrieben.

Kapitel 6

Realisierung eines Speedup-Analyse-Werkzeugs

6.1 Analyseablauf

Den letzten Teil der entwickelten Basiswerkzeuge bildet eine Anwendung zur Demonstration des Zugriffs auf die Leistungsdaten der Datenbank sowie zu einer ersten exemplarischen Auswertung der Daten. Dieser Baustein wurde durch ein Java-Applet realisiert, das infolge seiner Plattformunabhängigkeit und Client/Server-Fähigkeiten eine Anwendung auf jedem Rechnersystem erlaubt, das über einen Java-fähigen WWW-Browser verfügt. Durch Veröffentlichung im World Wide Web werden Wartungsvorgänge sowie Aktualisierungen erheblich vereinfacht. Das Werkzeug kann stets auf aktuellem Stand gehalten werden, ohne daß der Anwender lästige Update-Prozeduren durchlaufen muß.

Abbildung 6.1 zeigt die vollständige Struktur der Analyseumgebung, die sich durch diese letzte Komponente ergibt: Nach Übersetzung bzw. Instrumentierung und Lauf eines Programms auf der CRAY T3E liegen die CIF und RIF vor, aus denen Apprentice die Laufzeitdaten gewinnt. Diese Daten werden durch eine zusätzliche Aufruf-option in eine Exportdatei geschrieben. Dann wird auf der Workstation das Importskript `kojakimport` aufgerufen, das eine Netzwerkverbindung zur CRAY T3E herstellt und die Exportdateien sowie Quelltexte auf den eigenen Rechner überträgt. Hier werden die Daten aufgearbeitet und danach an den Datenbankserver übergeben, auf dem sie in der KOJAK-Datenbank abgelegt werden. Dieser Vorgang wird für beliebig viele Testläufe mit unterschiedlicher Prozessoranzahl wiederholt.

Der Start des Analyseapplets erfolgt nach dem Import aller erforderlichen bzw. verfügbaren Testläufe durch Laden eines HTML-Dokumentes vom WWW-Server. Dieses Applet greift dann auf die Datenbank zurück und gibt die Leistungsdaten sowie weitere Auswertungen aus.

Nachfolgend werden der Funktionsumfang sowie die Implementierung des Applets detaillierter dargestellt.

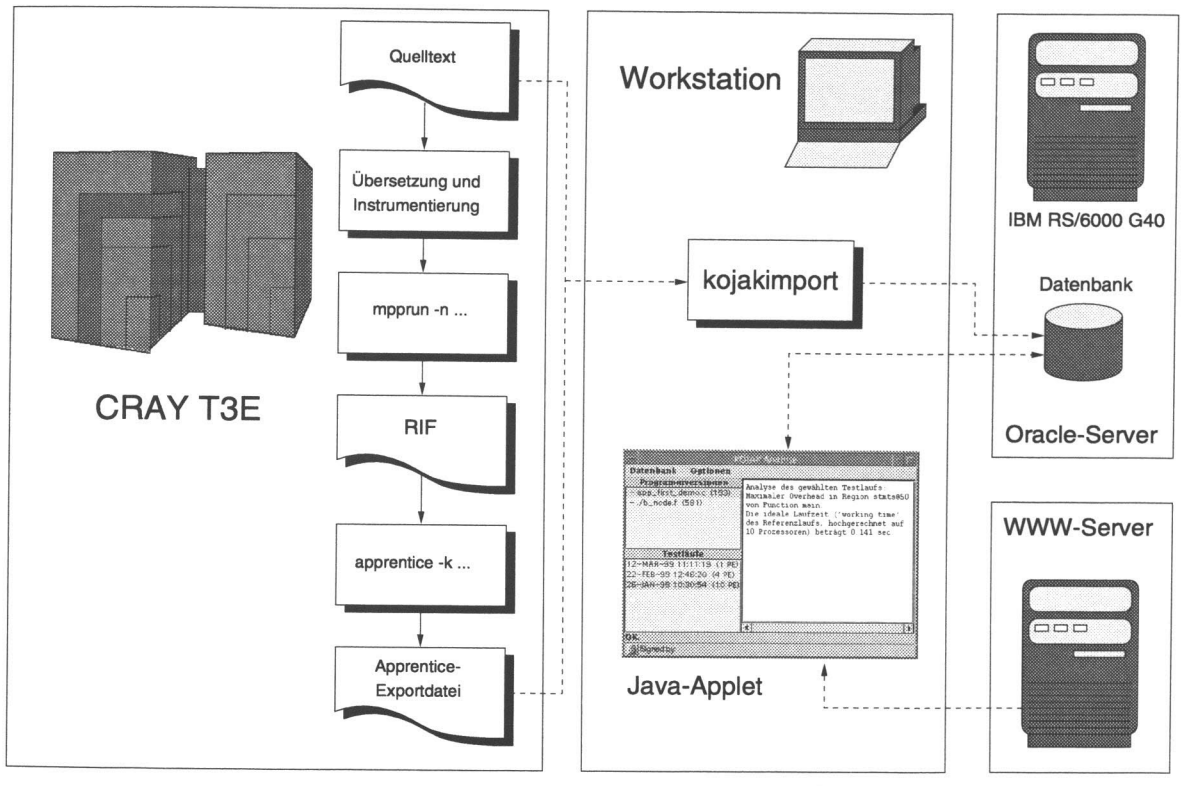

Abbildung 6.1: Struktur der automatischen Analyse-Umgebung

6.2 Funktionsumfang

6.2.1 Grundfunktionen

Abbildung 6.2 zeigt das zentrale Fenster, durch das die Analyse gesteuert wird. Nach dem Öffnen der Datenbank werden dem Anwender nur wenige Kerndaten gezeigt, die die wichtigsten Analyseinformationen bereitstellen. Das Fenster gliedert sich dabei in drei Teile:

- Die Liste der **Programmversionen** enthält Name und ID aller in der Datenbank gefundenen Programme. Existiert in dieser Liste ein Programm mehrfach, so handelt es sich um verschiedene Versionen dieses Programms. Eventuelle Versionsunterschiede werden über das Programmdatum ermittelt. Dieses Programmdatum wurde bereits in der Exportschnittstelle durch den jüngsten Quellcode bestimmt. Eine Veränderung des Quellcodes hat somit eine Identifikation als neues Programm zur Folge.

- Die Liste der **Testläufe** zeigt nach Auswahl eines Programms alle verfügbaren Testszenarien mit Datum des Programmlaufs sowie Anzahl der Prozessoren an.

- Ist in beiden Listen ein Element ausgewählt, so zeigt die **Textfläche** am rechten Rand des Fensters eine erste kurze Analyse des Programms in textueller,

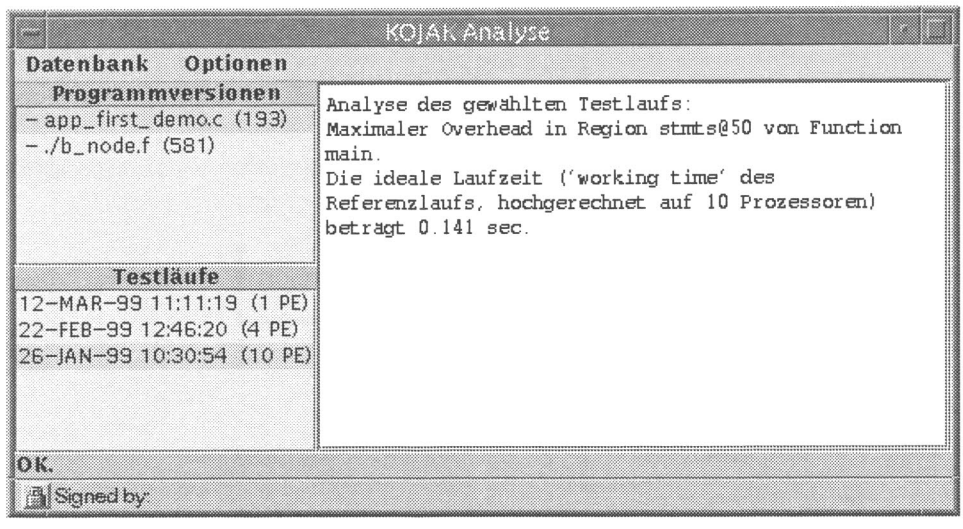

Abbildung 6.2: Hauptfenster des Java-Applets

ausformulierter Form an. Hierbei wird ein Hinweis auf die Region mit dem größten Overhead sowie die ermittelte ideale Laufzeit des gewählten Testlaufs ausgegeben.

Die ideale Laufzeit ergibt sich dabei aus der Annahme, daß das Programm optimal skalierbar ist und ein Programmlauf mit n Prozessoren genau $1/n$ der Rechenzeit eines Laufs mit einem Prozessor benötigt. Da die Daten für einen Prozessor nicht zwingend vorliegen, wird als Referenzlauf der Lauf mit der geringsten Prozessorzahl gewählt. Zusätzlich wird der Idealfall vorausgesetzt, daß kein Overhead entsteht und keine Rechenzeit durch Ein- oder Ausgabe verbraucht wird. Liegen solche Zeiten im Referenzlauf vor, so werden sie ignoriert. Für die Bestimmung der idealen Laufzeit t_{ideal} in bezug auf einen Testlauf mit P Prozessoren ist somit nur die reine Benutzerzeit des Referenztestlaufs mit $P_{\min}$ Prozessoren relevant. Da die Benutzerzeit genau dem Apprentice-Zeittyp `tt_work` entspricht, ergibt sich:

$$(t_{\text{ideal}})_P = \frac{\left(t_{\text{timingtype(tt\_work)}}\right)_{P_{\min}} \cdot P_{\min}}{P}$$

6.2.2 Erweiterte Funktionen

Die weitere Steuerung des Applets erfolgt ausschließlich über Menüs, deren Struktur in Abbildung 6.3 wiedergegeben wird. Durch das `Datenbank`-Menü können Datenbankverbindungen kontrolliert, durch das `Optionen`-Menü Erweiterungen des Hauptfensters aktiviert oder zusätzliche Analysefenster geöffnet werden.

Nach dem Start des Programms muß durch die Anwahl von `Verbinden...` zunächst eine Verbindung zur Datenbank erstellt werden. Es öffnet sich ein Fenster, in dem alle

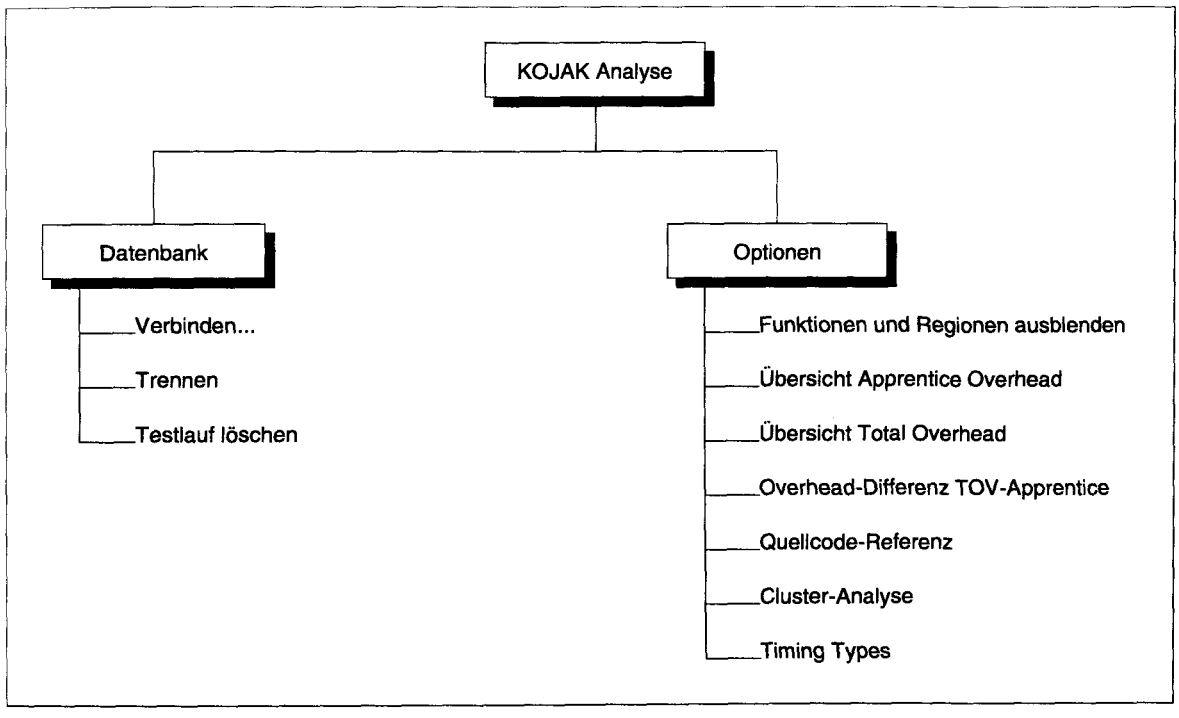

Abbildung 6.3: Menüaufbau des Java-Applets

relevanten Daten eingelesen werden, die für die Initialisierung des Oracle-Treibers erforderlich sind. Im einzelnen sind dies:

- der *Uniform Resource Locator* (URL) des Datenbankservers,
- der *Port* des Servers,
- die *Server ID* (SID),
- der Benutzername sowie
- das Paßwort.

Die Felder sind mit den für die zentrale Datenbank des Forschungszentrums Jülich gültigen Werten voreingestellt. Nach dem Öffnen der Verbindung werden alle verfügbaren Programmversionen angezeigt. Es kann jeweils nur eine Verbindung zu einer Datenbank geöffnet sein, auch wenn der Anwender über Zugriffsmöglichkeiten auf verschiedene Datenbankserver mit KOJAK-Leistungsdaten verfügt. Zum Wechsel der Datenbasis muß zunächst die alte Verbindung mit `Trennen` geschlossen und dann eine neue Verbindung erstellt werden.

Durch den Menüpunkt `Testlauf löschen` wurde eine Möglichkeit zur Manipulation der Datenbank integriert, da das in 4.5 beschriebene Import-Skript keine Funktionen zum Löschen von versehentlich importierten oder alten bzw. falschen Daten beinhaltet. Bei Anwahl dieser Funktion wird nach einer zusätzlichen Sicherheitsabfrage

der selektierte Testlauf sowohl aus der Anzeige als auch aus der Datenbank entfernt. Wird auf diese Art der letzte verfügbare Testlauf eines Programms gelöscht, so werden hierdurch auch die Einträge zur betreffenden Progammversion entfernt.

Das Beenden des Programms ist, wie bei Applets im allgemeinen üblich, nicht aus der Anwendung heraus möglich. Ein Abbruch wird durch das Verlassen der entsprechenden HTML-Seite veranlaßt.

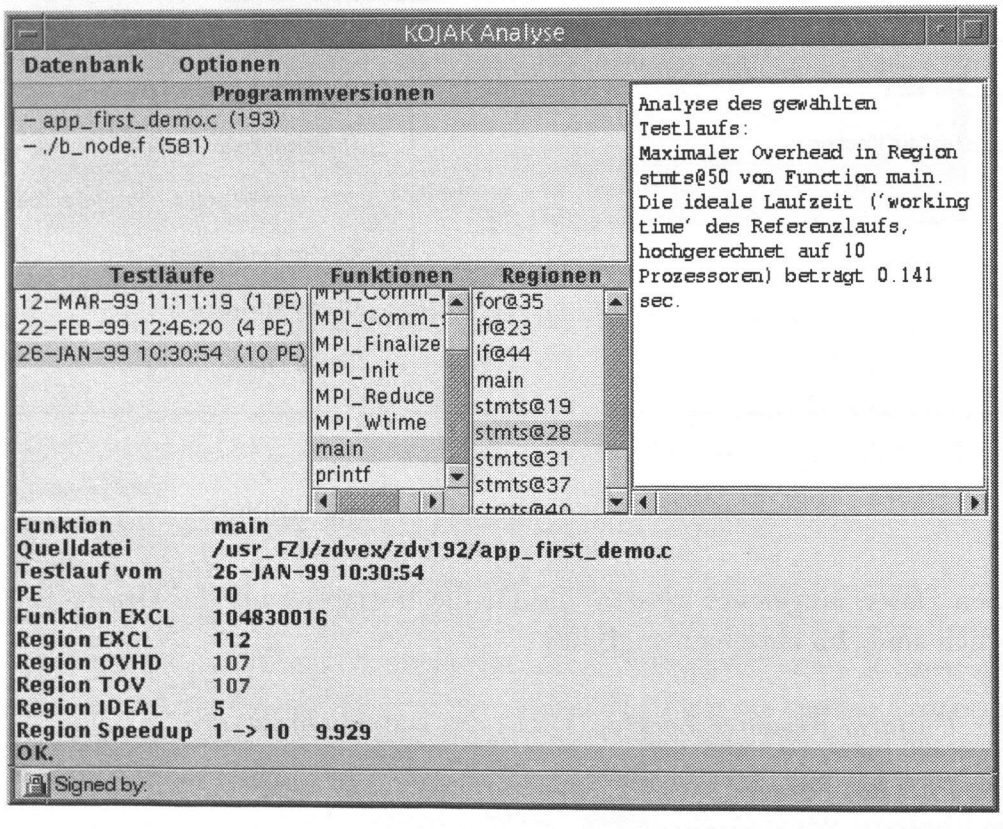

Abbildung 6.4: Erweitertes Hauptfenster des Java-Applets

Durch den Menüpunkt `Funktionen und Regionen ausblenden` des `Optionen`-Menüs können einige ergänzende Anzeigen des Hauptfensters zu- bzw. abgeschaltet werden. Zu diesem Zweck wird das Hauptfenster um drei weitere Teile erweitert (vgl. Abbildung 6.4):

- Die Liste der **Funktionen** enthält alle im selektierten Programm vorhandenen Funktionen.
- Die Liste der **Programmregionen und Basisblöcke** beinhaltet nach Auswahl eines Elements der Funktionsliste alle Regionen dieser Funktion.
- Die Anzeige der **Zusatzdaten** am unteren Rand des Fensters enthält nähere Angaben zu den in den Listen selektierten Elementen. So werden zu informativen Zwecken nochmals der Name der ausgewählten Funktion, die Datei

des Quelltextes, das Datum des selektierten Testlaufs, die Anzahl der Prozessoren sowie die exklusive Rechenzeit der Funktion angezeigt. Zur gewählten Region werden ebenfalls die exklusive Rechenzeit, der Apprentice-Overhead, der *Total Overhead* (TOV), die bereits beschriebene ideale Rechenzeit sowie Informationen zur Skalierbarkeit ausgegeben.

Alle Werte werden hierbei in Taktzyklen angegeben, wobei die exklusiven Rechenzeiten und der Overhead unverändert aus Apprentice übernommen werden.

Der Total Overhead t_{TOV} ist eine von Apprentice abweichende, genauere Bestimmung des Overhead. Die Berechnung von t_{TOV} in bezug auf einen Testlauf mit P Prozessoren erfolgt durch

$$\begin{aligned}(t_{\mathrm{TOV}})_P &= (t_{\mathrm{excl}})_P - (t_{\mathrm{ideal}})_P \\ &= (t_{\mathrm{ovhd}})_P + (t_{\mathrm{ovhd\_uninstr}})_P + (t_{\mathrm{I/O}})_P\end{aligned}$$

und stellt genau die Rechenzeit dar, die über die bereits berechnete ideale Rechenzeit hinausgeht. Der TOV entspricht somit dem Apprentice-Overhead, der um die Zeiten für Ein- und Ausgabe sowie für nicht instrumentierte Funktionen erweitert wird.

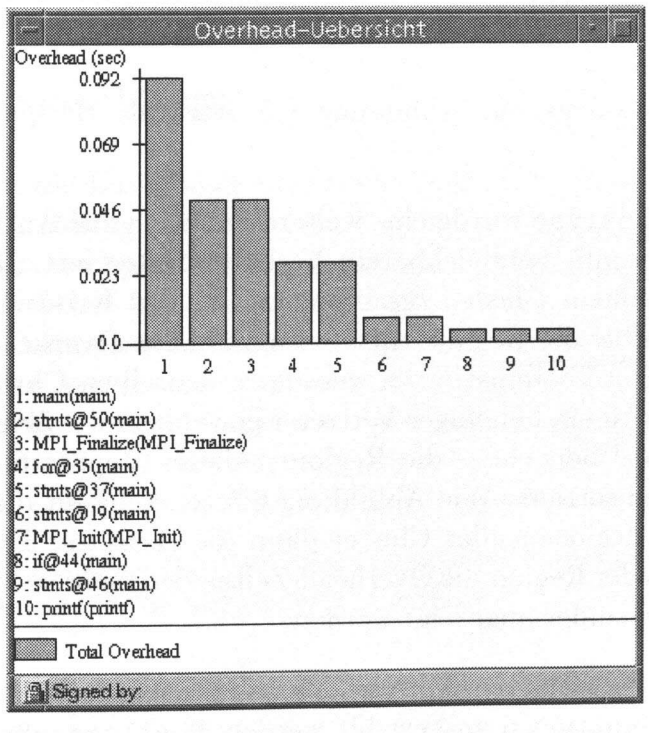

Abbildung 6.5: Overhead-Übersichtsfenster

Die Funktionen `Übersicht Apprentice Overhead`, `Übersicht Total Overhead` sowie `Overhead-Differenz TOV-Apprentice` öffnen jeweils ein Fenster, das in einem Balkendiagramm die Regionen mit ihren maximalen Overhead-Daten graphisch anzeigt. Die Meßwerte werden dabei nicht bewertet, sondern ihrem Betrag nach

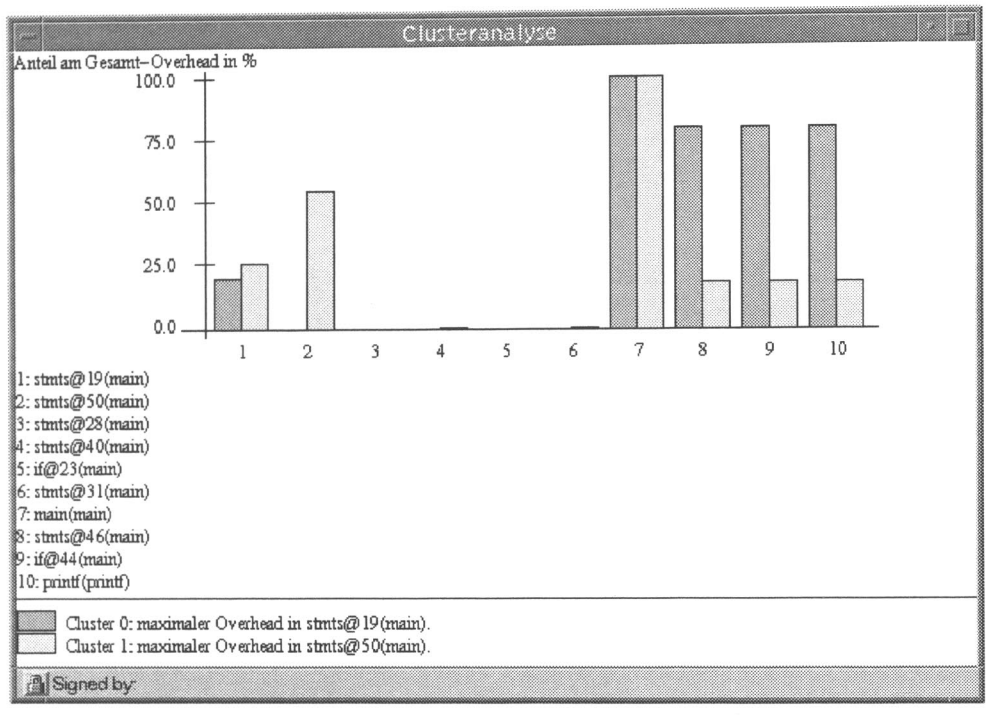

Abbildung 6.6: Cluster-Vergleich

absteigend sortiert ausgegeben. Abbildung 6.5 zeigt als Beispiel die Grafik des Apprentice-Overhead.

Durch die `Cluster-Analyse` wurde eine weitere Funktion zur Auswertung verschiedener Programmläufe mit vergleichbarem Verhalten integriert. Dabei werden alle ähnlichen Läufe zu einem *Cluster* zusammengefaßt. Als Kriterium für „ähnliches Verhalten" dient hierbei die Region mit dem maximalen Overhead: Alle Testläufe, bei denen diese Region übereinstimmt, werden in denselben Cluster übernommen. Aus jedem Cluster wird ein beliebiger Vertreter gewählt und – ähnlich zu den Funktionen zur Overhead-Übersicht – die Regionen dieses Repräsentanten absteigend nach ihrem Overhead sortiert. Wie Abbildung 6.6 zeigt, bildet die Vereinigung der zehn „kritischsten" Regionen aller Cluster dann die Basis zur Erstellung des Diagramms, in dem zu jeder Region die Overhead-Zeiten der Repräsentanten der Cluster vergleichend nebeneinander angezeigt werden.

Bei Aktivierung der erweiterten Anzeige des Hauptfensters können im `Optionen`-Menü zwei weitere Funktionen ausgewählt werden: `Quellcode-Referenz` öffnet ein Fenster, in dem der Quelltext der selektierten Funktion angezeigt wird, falls dieser verfügbar ist. Bei Auswahl einer Programmregion werden die entsprechenden Zeilen in diesem Fenster farbig markiert (vgl. Abbildung 6.7).

Die `Timing Types`-Option wurde zu Debugging- und Entwicklungszwecken implementiert und zeigt die gemessenen Werte der verschiedenen Apprentice-Zeittypen (vgl. Tabelle 3.2) ergänzend zu den Zeitdaten des Hauptfensters an.

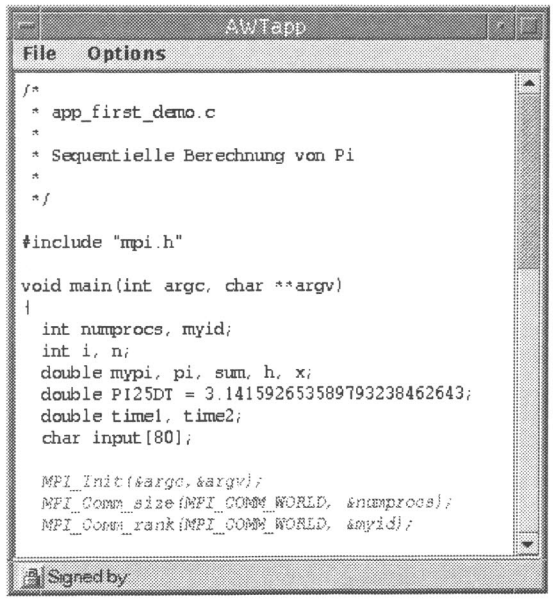

Abbildung 6.7: Anzeige des Quelltextes

6.3 Implementierung

6.3.1 Datenbankabfrage

Für den Zugriff auf die Datenbank wurde der JDBC-Thin-Treiber gewählt, da das Applet durch die Unabhängigkeit von einer Client-Installation auch von Anwendern genutzt werden kann, die keine direkte Datenbankanbindung haben. Zur Vermeidung der dennoch erforderlichen Installation der JDBC-Klassen wurden diese sogar in das Applet integriert. Dadurch vergrößert sich zwar das Applet nicht unerheblich, eine Analyse kann jedoch von jedem beliebigen Rechner mit Netzanbindung gestartet werden. Die Datenbank wurde als Objekt der Klasse `Database` implementiert, die die erforderlichen Funktionen zur Anmeldung sowie eine Instanz der JDBC-Klasse `Connection` enthält, über die die SQL-Anfragen erfolgen.

Da es nicht sinnvoll ist, den gesamten verfügbaren Datenbestand einzulesen, werden zum Programmstart durch die simple SQL-Anweisung

```
SELECT PROGID, NAME FROM PROGRAMVERSION;
```

zunächst nur Name und ID-Nummer aller Programmversionen ermittelt. Diese Anfrage reicht zur Anzeige in der Liste der Programmversionen aus. Erst nach Selektion eines Programms werden durch

```
SELECT TO_CHAR(TESTTIME, 'DD-MON-YY HH24:MI:SS'), PE, CLOCKSPEED
FROM TESTRUN
```

```
WHERE PROGID = <ID des gewählten Programms>
ORDER BY PE;
```

die verfügbaren Testläufe eingelesen und angezeigt. Die TO_CHAR-Konstruktion dient dabei der Transformierung des datenbankinternen Datentyps DATE in eine Zeichenkette. Gleichzeitig erfolgt die Abfrage aller Funktions-, Regions- sowie Zeitdaten des gewählten Programms. Diese Abfrage gliedert sich in die folgenden drei SQL-Anweisungen:

```
SELECT FUNCTION_NAME, NAME, EXCL, OVHD
FROM TOTAL_REGION_TIMING
WHERE PROGID = <ID des gewählten Programms>
AND TESTTIME = <Datum und Zeit des gewählten Testlaufs>

SELECT FUNCTION_NAME, NAME, BEGINRANGE, ENDRANGE
FROM SOURCE_CODE_REFERENCE
WHERE PROGID = <ID des gewählten Programms>

SELECT FUNCTION_NAME, NAME, TYPE, TIMING
FROM TYPED_REGION_TIMING
WHERE PROGID = <ID des gewählten Programms>
AND TESTTIME = <Datum und Zeit des gewählten Testlaufs>
```

Die Ergebnisse werden durch Ergänzung der Anweisung

```
ORDER BY FUNCTION_NAME, NAME;
```

jeweils nach Funktions- und Regionsnamen sortiert, so daß die Rückgabelisten parallel durchlaufen und die internen Datenstrukturen hieraus zusammengesetzt werden können.

6.3.2 Klassen des Java-Applets

Das Applet baut auf vier Basisklassen auf, die von einer zusätzlichen, zentralen Klasse `Analyse` gesteuert werden. Diese Klasse realisiert auch die Strukturen des Hauptfensters sowie des Menüaufbaus und behandelt alle Ereignisse, die durch unterschiedliche Anwenderaktionen ausgelöst werden. Der Start des Applets erfolgt durch die Klasse `Kojak`, deren Aufgabe sich darauf beschränkt, ein Objekt von `Analyse` zu bilden. Danach wird die Kontrolle vollständig an dieses Objekt übergeben.

Von den Basisklassen `ProgramVersion`, `TestRun`, `Function` und `ProgramRegion` wird zu Beginn jeweils genau eine Instanz gebildet. Diese repräsentieren in Form von

Funktion	Beschreibung
`tt`	Apprentice-Zeittypen
`excl, incl, ovhd`	Aufsummierte Apprentice-Laufzeitdaten
`tov, ideal`	Vom Applet berechnete Laufzeitdaten
`getRegionName`	Name der Region
`numRanges`	Anzahl der Quellcodereferenzen
`rangeBegin, rangeEnd`	Erste und letzte Zeile einer Quellcodereferenz
`numRegions`	Anzahl der Regionen einer Funktion
`getMainRegion`	Index der Region, die die Funktion repräsentiert

Tabelle 6.1: Funktionen zum Zugriff auf Programm- und Laufzeitdaten

Listen genau die vier Objekttypen, die schon im ER-Diagramm in 4.3 das Grundgerüst des Datenbankentwurfs bilden.

Die jeweiligen Klassen stellen in ihren Schnittstellen Funktionen zur Verfügung, durch die von außen ein einfaches Auslesen von Programm- und Laufzeitinformationen der Programmregionen möglich ist. Eine Identifikation wird dabei über die Positionen in den einzelnen Listen realisiert. So werden Testläufe und Funktionen durch die jeweilige Position in ihrer Liste, Programmregionen zusätzlich durch den Index der umgebenden Funktion eindeutig bestimmt. Zusätzlich wurde eine Klasse `RegionIdentifier` implementiert, die alle drei Indizes einer Region in einem Objekt zusammenfaßt und so den Zugriff vereinfacht.

Das Auslesen der Apprentice-Laufzeitdaten erfolgt durch die in Tabelle 6.1 dargestellten Funktionen, die in der Klasse `ProgramRegion` bereitgestellt werden. Die Aufrufparameter dieser Funktionen orientieren sich am Schema

```
(int i, int pe, int region, int function)   bzw.
(int i, RegionIdentifier ri).
```

Die einzelnen Parameter sind jedoch nur implementiert, sofern es für die betreffende Funktion sinnvoll ist. So muß der Testlauf nur im Falle von laufzeitabhängigen Funktionen wie z. B. `excl` angegeben werden. Der Wert `i` wird nur bei solchen Funktionen angewendet, die mehrere Rückgabewerte je Programmregion zurückliefern können und dient zur Ansteuerung des i-ten Wertes. Mit Ausnahme des Zählers `i` sind die Argumente beginnend mit dem letzten sukzessiv optional. Sollten ein oder mehrere Indizes nicht angegeben sein, so wird der Wert des aktuell selektierten Listenelements eingesetzt.

Darüber hinaus wurden in `ProgramRegion` Funktionen implementiert, die die Basis für die im `Optionen`-Menü zuschaltbaren Graphiken zur Clusteranalyse und zum Overhead-Vergleich bilden. Auch die im Textbereich des Hauptfensters angezeigte Region mit maximalem Overhead wird hier bestimmt. Da jedoch der Overhead jeder Region den ihrer Subregionen beinhaltet, liegen die maximalen Overhead-Werte in den Funktionsknoten des Apprentice-Programmbaums. Eine Suche nach

Klasse	Beschreibung	
`Kojak`	Aufrufklasse zur Integration in ein HTML-Dokument	Hauptklassen
`Analyse`	Steuerung der Benutzerschnittstelle und Aufruf der Analysefunktionen	
`Database`	Verwaltung der Datenbankverbindung	
`ProgramVersion`	Verwaltung Programmversionen	Programm- und Laufzeitdaten
`TestRun`	Verwaltung der Testläufe	
`Function`	Verwaltung der Funktionen	
`ProgramRegion`	Verwaltung der Programmregionen	
`RegionIdentifier`	Identifizierung von Programmregionen	
`Browser`	Fenster für die Anzeige des Quellcodes	Analysefunktionen
`OvhdReport`	Fenster für die Overhead-Übersicht	
`ClusterAnalyse`	Fenster für die Cluster-Analyse	
`ClusterMaxRegion`	Hilfsklasse zur Bildung der Cluster	
`BarChart`	Zeichnen des Balkendiagramms für die Overhead-Übersicht und Cluster-Analyse	
`AreYouSure`	Fenster für die Sicherheitsabfrage beim Löschen von Testläufen	Hilfsfunktionen
`PWDialog`	Fenster zur Abfrage der Datenbank-Benutzerdaten	

Tabelle 6.2: Klassen des Java-Applets

dem Programmteil mit dem betragsmäßig maximalen Overhead würde daher eine ganze Funktion ermitteln und wäre für einige Auswertungen zu ungenau. Aus diesem Grund erfolgt das Aufspüren der kritischen Programmteile durch einen relativen Overhead $t_{\text{rel\_ovhd}}$, der als Quotient von Anteil an Gesamtoverhead und Anteil an der Gesamtrechenzeit berechnet wird:

$$(t_{\text{rel\_ovhd}})_P = \frac{(t_{\text{ovhd}})_P \cdot (t_{\text{sum\_excl}})_P}{(t_{\text{excl}})_P \cdot (t_{\text{sum\_ovhd}})_P}$$

Hierdurch werden Regionen mit hohem Overheadanteil, aber nur geringem Rechenzeitanteil bevorzugt. Dies bildet zwar keine Garantie dafür, daß mit der Suche nach der Region mit maximalem Overhead gleichzeitig der kritischste Programmteil aufgespürt wird, dennoch ist die Bestimmung genauer als bei der Verwendung des betragsmäßigen Overhead.

Der relative Overhead findet Anwendung bei der Ermittlung der Region mit maximalem Overhead im Textfenster sowie bei der Bestimmung der Cluster, da die Verwendung des unbewerteten Overhead bzw. TOV in diesen Fällen triviale Ergebnisse zur Folge hätte. Bei der Anzeige der Overhead-Übersicht wird hingegen auf den betragsmäßigen Overhead zurückgegriffen, da zur vergleichenden Darstellung mehrerer Regionen auch die Funktionen von Interesse sind.

Die Steuerung der verschiedenen Auswertungen wird durch die Klassen `OvhdReport` sowie `ClusterAnalyse` übernommen. Beim Aufruf der Konstruktoren dieser Klasse werden Verweise auf die aktuellen Objekte von `TestRun`, `Function` sowie `ProgramRegion` übergeben. Dadurch können nach der Erstellung eines neuen Fensters die jeweiligen Laufzeitdaten ausgelesen und mit Hilfe der allgemeinen Klasse `BarChart` angezeigt werden. Der Konstruktor von `OvhdReport` enthält ein weiteres Argument, durch das ausgewählt werden kann, welche Art von Overhead ausgewertet werden soll.

Die Klasse `Browser` übernimmt die Anzeige des Quelltextes. Die Implementierung der Grundstrukturen dieser Klasse stand im Zentralinstitut für Angewandte Mathematik bereits zur Verfügung und mußte nur geringfügig modifiziert werden. Auf die genauen Strukturen sowie die Hilfsklassen von `Browser` soll hier nicht näher eingegangen werden.

Tabelle 6.2 zeigt eine Übersicht aller im Java-Applet implementierten Klassen.

6.4 Probleme bei der Implementierung

Leider ist die Realisierung als Java-Applet von einigen Problemen geprägt, die im Hinblick auf die Implementierungsaspekte einige Einschränkungen zur Folge haben. Vor allem aus Gründen der Laufzeiteffizienz konnten nicht alle Vorteile der objektorientierten Programmierung sowie der relationalen DBMS ausgenutzt werden.

Alleine das Java-Laufzeitsystem – die *Java Virtual Machine* – benötigt mit dem Netscape Communicator eine Initialisierungszeit von etwa einer Minute. Erst nach dieser Wartezeit wird das Analysefenster geöffnet und der Anwender kann mit dem Erstellen der Datenbankverbindung beginnen. Auch nach dieser Startphase sind Einschränkungen in bezug auf die Geschwindigkeit deutlich spürbar. So kann es vorkommen, daß das Applet nach dem Anklicken eines Menüs erst nach einer Verzögerung von 1 – 2 Sekunden reagiert.

Aus diesem Grund beinhalten die Listen `ProgramVersion`, `TestRun`, `Function` sowie `ProgramRegion` keine Instanzen eigenständiger Klassen, wie es im Sinne der objektorientierten Programmierung naheläge. Diese Implementierung würde bei großen Datenbeständen die Erzeugung mehrerer tausend Objekte einer komplexen Klasse zur Folge haben, die zumindest die folgenden Funktionen bzw. Variablen umfassen sollte:

- Speicherung und Zugriff auf die Timing-Daten aller Testläufe
- Modellierung der Apprentice-Baumstruktur
- Bereitstellung aller Informationen und Verweise zu Objekten anderer Klassen, zu denen eine Verbindung besteht (z. B. zugeordnete Testläufe)

Art der Datenbankanfrage		JDBC		Pro*C	
		Query	Fetch	Query	Fetch
60 Datensätze	unsortiert	19 ms	40 – 50 ms	5 – 8 ms	35 – 40 ms
	sortiert	31 ms			
780 Datensätze	unsortiert	20 ms	800 ms		340 ms
	sortiert	129 ms			515 ms
5551 Datensätze	unsortiert	65 ms	5 – 6 s		2.0 s
	sortiert	1.2 s			3.1 s

Tabelle 6.3: Zugriffszeiten für Datenbankabfragen

- Ermittlung der zur Anzeige in der Liste erforderlichen Angaben

Eine Generierung dieser Objekte würde jedoch die Laufzeit des Applets auf ein nicht mehr vertretbares Maß steigern. Stattdessen werden die Elemente der jeweiligen Liste, wie in 6.3.2 beschrieben, konventionell in einem Array gesichert. Aus diesem Grund wurde gleichzeitig auf eine weitere Verwendung der von Apprentice in die Datenbank übernommenen Baumstruktur verzichtet.

Als weitere Hürde erweist sich – wie schon bei der Entwicklung des Importskripts in Perl – der Datenzugriff über JDBC. Kritisch sind hierbei sowohl die Generierung bzw. Ausführung des SQL-Statements als auch das Abrufen der Ergebnis-Datensätze. Vor allem die Ausführung der Anfrage erforderte in Testmessungen Laufzeiten in der Größenordnung von Sekunden.

Tabelle 6.3 zeigt die Zugriffszeiten einiger Testmessungen mit JDBC, getrennt nach Zeiten für die Ausführung der Anweisung („Query") und für das Abrufen der Ergebnisse („Fetch"). Zum Vergleich wurden die gleichen Abfragen unter C mit Hilfe des Oracle-Precompilers *Pro*C* implementiert. Alle Abfragen wurden mit der bei JDBC voreingestellten Prefetching-Größe von zehn Datensätzen durchgeführt. Grundlage dieser Messungen waren reale Datenbestände aus den in Kapitel 4 beschriebenen Relationen. Dadurch ergeben sich auch die für Testmessungen sicherlich unüblichen Größen der einzelnen Abfragen.

Die Messungen zeigen, daß die Performance von JDBC vor allem im Bereich der Ergebnisabfrage deutlich unter der von C liegt. Bei Werten im Bereich weniger Millisekunden sind die Messungen jedoch sehr ungenau, da der allgemeine Netzverkehr auch die Verbindung zum Datenbank-Server beeinflußt. Bei starker Netzlast werden die angegebenen Zeiten nicht mehr erreicht und können leicht um ein Vielfaches überschritten werden.

Die Zeiten für die Ausführung der SQL-Anweisung sind bei JDBC im Gegensatz zur interaktiven Oracle-Schnittstelle SQL*Plus oder zu klassischen Programmiersprachen wie z. B. C/C++ stark von der Komplexität der Anfrage abhängig. So wirkt sich bereits das Sortieren des Ergebnisses durch die Anweisung `ORDER BY...` sehr negativ auf die Performance aus. Dies ist sicherlich auf die in 5.3 bereits erwähnte nur rudimentäre Implementierung der JDBC-Treiber zurückzuführen.

Ebenso sollte eine zu intensive Nutzung des Join-Operators, der einem Kartesischen Produkt gleichkommt, vermieden werden. Ein Join von mehr als zwei Tabellen ist meist nicht praktikabel, da hieraus meist extrem hohe Zugriffszeiten resultieren.

Ein JDBC-Datenbankzugriff „je nach Bedarf" ist in der Implementierung des Applets aus diesen Gründen nicht sinnvoll. Jede Benutzeraktion würde einige SQL-Statements zur Abfrage der jeweiligen Laufzeitdaten erzeugen, wodurch die Laufzeit erheblich gesteigert würde. Darüber hinaus wäre eine gezielte Abfrage von Maxima oder Durchschnittswerten, die bei der Suche nach kritischen Programmteilen sicherlich zweckmäßig wäre, nur über die Verknüpfung von Tabellen zu realisieren. Da dies eine starke Verwendung des Join-Operators zur Folge hat, ist diese Vorgehensweise in Anbetracht der hohen Laufzeiten nicht sinnvoll. Der ursprüngliche Wunsch, die Datenbank auch als Datenbestand des Applets zu betrachten und dadurch die Verwaltung eigener Datenstrukturen weitestgehend zu vermeiden, kann dadurch nicht umgesetzt werden. Stattdessen wird versucht, alle erforderlichen Daten mit möglichst wenigen SQL-Anweisungen aus der Datenbank in eigene Strukturen zu übertragen.

Ein weiteres Problem entsteht in der Praxis durch die starke Vernetzung der an der Analyse beteiligten Komponenten. Durch die Implementierung als Applet kommt neben dem Datenbank-Server und der lokalen Workstation mit dem Webserver eine weitere Netzwerkkomponente zum Einsatz. Bei zusätzlicher Betrachtung des CRAY-Rechners, der die Leistungsdaten ursprünglich liefert, müssen zur Speedup-Analyse mindestens vier in ihrer Architektur teilweise unterschiedliche Rechner zusammenarbeiten und miteinander harmonieren. Dies kann jedoch durch Ausfälle im Netzwerk die Anwendung einschränken: Ist nur eine Komponente in diesem System nicht zugänglich, so ist eine Analyse nicht möglich.

6.5 Integration in ein HTML-Dokument

Zur Integration des Applets in ein HTML-Dokument, das im World Wide Web veröffentlicht werden kann, sind durch das Java-Sicherheitskonzept einige zusätzliche Schritte erforderlich. Unter der Kontrolle des anzeigenden Web-Browsers verhindern spezielle Sicherungen[1] unerwünschte Nebenwirkungen auf dem Client-Rechner. So können Applets im allgemeinen nicht auf lokale Dateien, Systemkomponenten oder Programme zugreifen, und auch Internet-Verbindungen können ausschließlich zu dem Server, von dem sie geladen wurden, hergestellt werden. Erst ein vertrauenswürdiges Applet (*Trusted Applet*), das durch eine digitale Unterschrift signiert wurde, kann nach einer zusätzlichen Zustimmung des Anwenders diese Dienste ausführen.

Diese Signierung beruht auf dem Prinzip der *Public Key*-Kryptographie. Dabei erfolgt eine Chiffrierung von Daten durch ein Schlüsselpaar – dem öffentlichen (*public*)

[1] In der Java-Terminologie werden diese Sicherungen als „Sandkasten" (engl.: sandbox) bezeichnet, da das Applet eingeschlossen ist und nicht aus seiner Umgebung ausbrechen kann.

und dem privaten (*private*) Schlüssel. Der öffentliche Schlüssel wird der Außenwelt zugänglich gemacht, der private Schlüssel bleibt geheim. Informationen, die mit dem öffentlichen Schlüssel chiffriert wurden, können nur mit dem privaten Schlüssel dechiffriert werden. Ebenso gilt der umgekehrte Fall, der zunächst wenig zweckmäßig erscheinen mag, weil es kaum sinnvoll ist, wertvolle Daten durch den privaten Schlüssel zu schützen, da der Schlüssel zur Dechiffrierung frei zugänglich ist. Genau dies macht sich jedoch die digitale Unterschrift zunutze, indem die Daten (in diesem Fall also das Applet) durch den privaten Schlüssel unterschrieben werden. Jeder kann dadurch mit Hilfe des öffentlichen Schlüssels die Herkunft des Applets verifizieren. Voraussetzung hierfür ist jedoch, daß der öffentliche Schlüssel auf vertrauenswürdigem Weg zum Anwender übermittelt wurde.

Da die Sicherungen Bestandteil der Web-Browser sind, existieren leider keine standardisierten Java-Anweisungen zur Steuerung der Zugriffsprivilegien. Zur Zeit ist die Entwicklung von signierten Applets lediglich mit dem *Netscape Communicator* sowie dem *Microsoft Internet Explorer* möglich. Im Rahmen dieser Arbeit wurde das Applet für den Communicator implementiert und signiert (vgl. [21]). Eine Erweiterung für den Internet Explorer verhält sich jedoch ähnlich und sollte keine Einschränkung für das Analyse-Werkzeug darstellen.

Die Kontrolle von Privilegien erfolgt für den Communicator durch den *Privilege Manager*, der die verschiedenen möglichen Zugriffsrechte überwacht. So wird z. B. zwischen Dateizugriff, Netzwerkzugriff oder Zugriff auf die Systemdaten unterschieden. Auf den Privilege Manager wird durch die Bildung einer Instanz der Klasse `netscape.security.PrivilegeManager` zugegriffen. Das im Falle des Datenbankzugriffs erforderliche Privileg, eine Verbindung zu entfernten Rechnern herzustellen, wird durch

```
if (System.getProperty("java.vendor").startsWith("Netscape"))
{
    PrivilegeManager.enablePrivilege("UniversalConnect");
}
```

angefordert. Befindet sich der entsprechende öffentliche Schlüssel zur Signierung in der Schlüsseldatenbank des Communicators[2], öffnet sich durch diese Anweisung ein Fenster, in dem der Anwender dem gewünschten Privileg zustimmen kann. Eine Verbindung zur Datenbank kann dann ohne weitere Einschränkungen erfolgen.

Zur Generierung der Schlüssel und zum Signieren stellt Netscape das Programm `signtool` bereit, das für die gängigsten Rechnerplattformen über [22] bezogen werden kann. Durch den einfachen Aufruf von `signtool -G <Name des Schlüssels>` kann ein neues Schlüsselpaar erzeugt werden, dessen privater Schlüssel in der eigenen Netscape-Datenbank und dessen öffentlicher Schlüssel in den Dateien `x509.cacert` sowie `x509.raw` im ASCII- bzw. Binärformat gesichert werden.[3] Dieses Schlüsselpaar

[2] Der Schlüssel muß in der Datenbank zusätzlich als vertrauenswürdig gekennzeichnet sein. Dies ist durch Änderung der Einstellungen im Security-Dialog möglich.

[3] Vor Erzeugung dieses Schlüsselpaares muß die Netscape-Datenbank mit einem Paßwort gegen unbefugten Zugriff gesichert werden. Bei fehlendem Paßwort werden keine Schlüssel generiert.

ist mit einer Gültigkeit von drei Monaten zu Test- und Entwicklungszwecken geeignet. Für einen professionellen Einsatz kann von offiziellen Institutionen ein länger gültiges, zertifiziertes Schlüsselpaar erworben werden. Ein Verbreiten des öffentlichen Schlüssels erfolgt über die Integration in ein HTML-Dokument. Der Binärschlüssel `x509.raw` besitzt den MIME-Typ `application/x-x509-ca-cert`, so daß bei entsprechend konfiguriertem Web-Server bei Anklicken eines Verweises auf diese Datei der Schlüssel automatisch in die Datenbank des Anwenders übernommen wird.

Ein Signieren des Applets erfolgt durch den Aufruf von

```
signtool -k <Name des Schlüssels> -Z <Name des Archivs> <Verzeichnis>.
```

Dadurch werden alle im angegebenen Verzeichnis befindlichen Klassen in einem Archiv zusammengefaßt, das anschließend signiert wird. Das Archiv wird im *Java Archive*-Format (JAR) angelegt, so daß alle Elemente des Applets in einem Paket veröffentlicht werden können. Der Netscape-Client ist in der Lage, dieses Archiv zu lesen und die benötigten Klassen zu extrahieren.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Die Entwicklung effizienter Anwendungen auf Parallelrechnern stellt durch die komplexe Architektur der Rechner eine große Herausforderung bei der Softwareerstellung dar. Die klassischen Werkzeuge zur Unterstützung der Implementierung und Optimierung von parallelen Programmen beschränken sich meist auf die Visualisierung von Laufzeitdaten und Ereignisspuren. Zur Interpretation dieser Informationen muß ein Entwickler daher neben problemspezifischem Wissen auch über Erfahrung im Umgang mit den gegebenen Analysewerkzeugen verfügen. Zusätzlich sind Kenntnisse über potentielle Engpässe erforderlich.

Die im Rahmen dieser Arbeit entworfene Datenbank zur Aufnahme von Leistungsdaten sowie die entwickelten Basiswerkzeuge zur Bearbeitung und Auswertung dieser Daten bilden ein Instrumentarium, auf dessen Basis eine Umgebung zur automatischen Performanceanalyse realisiert werden kann.

Als Grundlage dienen die Leistungsdaten des auf der CRAY T3E verfügbaren Werkzeugs Apprentice. Apprentice besitzt jedoch keine Möglichkeit zum Export der gemessenen Daten. Da ein direktes Einlesen der von Apprentice verwendeten Runtime Information Files angesichts ihrer Komplexität nicht sinnvoll ist, wurde Apprentice um eine Funktion erweitert, durch die die graphisch angezeigten Werte in textueller Form in eine Datei ausgegeben werden können, so daß eine Weiterverarbeitung ermöglicht wird.

Diese Daten werden in dem im Forschungszentrum Jülich verfügbaren relationalen Datenbanksystem Oracle 7 zusammengeführt und verwaltet. Zu diesem Zweck ist eine Datenbank auf Basis eines Entity-Relationship-Modells entworfen worden, die die Apprentice-Strukturen zur Darstellung eines Programms, Informationen zu verschiedenen Testszenarien sowie die zugehörigen Quelltexte aufnehmen kann.

Die Übernahme der Daten erfolgt durch ein Perl-Skript, das in der Lage ist, die auf der CRAY T3E bereitliegenden Exportdateien und Quelltexte auf den lokalen

Arbeitsplatzrechner zu übertragen, auszuwerten und an die Datenbank zu übergeben. Dabei können durch die große Anzahl einzufügender Daten unangemessen hohe Laufzeiten entstehen. Aus diesem Grund werden zur optimalen Ausnutzung der Performance sowohl das Oracle-Importwerkzeug SQL*Loader als auch die Perl-Datenbankschnittstelle DBI eingesetzt.

Durch ein Java Applet wird der Zugriff auf die Datenbank demonstriert und eine erste Analyse der Leistungsdaten realisiert. Des weiteren können Testläufe mit unterschiedlicher Prozessoranzahl miteinander verglichen werden. Darüber hinaus umfaßt das Applet die folgenden Funktionen:

- Anzeige der Apprentice-Leistungsdaten
- Suche der Programmregion mit maximalem Overhead
- Berechnung des Speedup in bezug auf einen Referenzlauf
- Berechnung der idealen Laufzeit
- Graphische Anzeige einer Cluster-Analyse
- Berechnung des Total Overhead
- Graphische Übersicht der Programmregionen mit maximalem Overhead
- Anzeige der Quellcode-Referenz einzelner Programmregionen

Der Hinweis auf die Region mit maximalem Overhead sowie die ideale Laufzeit der Testläufe werden dem Anwender in textueller, ausformulierter Form angezeigt.

Die Implementierung als Java-Applet paßt sich dem Trend der modernen Programmierung an und ermöglicht eine plattformunabhängige Realisierung. Durch Integration in ein HTML-Dokument und Veröffentlichung im World Wide Web kann dem Anwender ohne größeren Aufwand stets eine aktuelle Version des Werkzeugs bereitgestellt werden.

Leider wird dieses hohe Maß an Flexibilität und Portabilität mit langen Laufzeiten erkauft, die eine Anwendung der Analyseumgebung zur Zeit stark einschränken. Der Datenbankzugriff über die in Java verfügbaren JDBC-Treiber bildet dabei, wie auch beim Import der Daten, den weitaus größten Engpaß bei der Verarbeitung und Auswertung der Leistungsdaten.

Verschärft wird dieses Problem durch die Anzahl der am Analyseprozeß beteiligten Netzwerkkomponenten, die sich durch den Einsatz der CRAY T3E, des Oracle-Servers, des HTTP-Servers und des Arbeitsplatzrechners ergibt. Durch hohe Resourcenauslastungen sind Ausfälle, durch die der Einsatz des Systems verhindert wird, nicht auszuschließen.

Sowohl die verfügbaren Java-Laufzeitumgebungen als auch die Implementierung der verwendeten JDBC-Klassen befinden sich noch nicht in einem völlig ausgereiften

Entwicklungsstadium. Dies betrifft insbesondere die JDBC-Treiber für das verwendete Datenbanksystem Oracle 7, da diese durch Oracle weder unterstützt noch weiterentwickelt werden. Sämtliche JDBC-Entwicklungen seitens Oracle konzentrieren sich auf das neueste Produkt Oracle 8.

Ebenso befinden sich die verwendeten Java-Klassen zur Darstellung der graphischen Benutzeroberfläche „Swing“ noch in der Beta-Phase und werden von den anzeigenden Browsern nicht direkt unterstützt. Stattdessen ist zur Zeit eine zusätzliche lokale Installation der Swing-Klassen erforderlich.

Es ist zu erwarten, daß mit Fortschreiten der Arbeiten auf diesen Gebieten eine erhebliche Leistungssteigerung des Applets verbunden ist. Einige Messungen mit einer Testinstallation eines Oracle 8-Servers bestätigen diese Annahmen. Ebenso werden die Laufzeiten mit der Fertigstellung des Java Just-In-Time Compilers (JIT) weiter optimiert werden.

Diese Arbeit zeigt, daß eine automatische Performanceanalyse auf Basis der hier implementierten und vorgestellten Techniken realisiert werden und bei hoher Ausfallsicherheit der Komponenten sowie bei künftigen, schnelleren Rechnersystemen durchaus sinnvoll zum Einsatz kommen kann. Auch die Wahl einer relationalen Datenbank als zentrale Verwaltungsinstanz erscheint trotz der Engpässe beim Datenimport sowie beim Zugriff aus dem Analysewerkzeug heraus zweckmäßig. Bei Betrachtung der geplanten Erweiterungen, die sicherlich ein breiteres Spektrum verschiedener Teilkomponenten sowie eine noch stärkere Nutzung verschiedener Netzwerkkomponenten mit sich bringen, schafft der Einsatz einer solchen Datenbank zusätzliche Flexibilität und erleichtert das Zusammenführen von Leistungsdaten aus verschiedenen Quellen.

7.2 Ausblick

Ein Ansatzpunkt für weitergehende Arbeiten kann zunächst der Ausbau des Applets in bezug auf die Kompatibilität zum Microsoft Internet Explorer bilden. Obwohl Oracle sich zum Standard entwickelt hat, kann im Hinblick auf die Portabilität ebenso eine Unterstützung weiterer relationaler Datenbanksysteme integriert werden.

Im Rahmen des am Zentralinstitut für Angewandte Mathematik entwickelten KOJAK-Projektes sollen auf Basis dieser Arbeit weitere Verfahren zur automatischen Erkennung von Leistungsengpässen in parallelen oder verteilten Anwendungen entwickelt werden.

Die implementierten Basiswerkzeuge beschränken sich gegenwärtig auf die exemplarische Auswertung der Leistungsdaten des Werkzeugs Apprentice auf der CRAY T3E. Daher ist eine Erweiterung sowohl der Datenbank als auch des Applets zur Aufnahme und Verarbeitung von Leistungsdaten weiterer Werkzeuge geplant.

Darüber hinaus kann die Instrumentierung, die Ausführung sowie die Übernahme der Leistungsdaten der zu untersuchenden Anwendung automatisch durch die Ana-

lyseumgebung gesteuert werden. Dies ist eine wichtige Erweiterung, die aufgrund ihres Umfangs im Rahmen dieser Arbeit nicht implementiert wurde.

Da Performanceprobleme erfahrungsgemäß in den meisten Fällen nur aus einer kleinen Menge von typischen Fehlern resultieren, wird eine Erweiterung der Analyse der gesammelten Leistungsdaten angestrebt. Hierzu soll eine Reihe von häufig auftretenden Engpässen anhand ihrer definierten Merkmale identifiziert und ausgewiesen werden. So kann die Analyseumgebung nicht nur von Experten, sondern auch von unerfahrenen Anwendern eingesetzt werden.

Anhang A

Apprentice-Modifikationen

Datei: `app_env.C`
Funktion: `validate_options()`
Aufgabe: Prüfung der Kommandozeilenparameter
Änderung: Ergänzung einer zusätzlichen Prüfung des neuen Schalters `-k`.

```
...
(strcmp( argv[i], "-r" ) != 0) &&
(strcmp( argv[i], "-k" ) != 0) && }
(strcmp( argv[i], "-V" ) != 0) )
...
```

Datei: `app_env.C`
Funktion: `process_command_options()`
Aufgabe: Auswertung der Kommandozeilenparameter
Änderung: Bei Auswahl des bisherigen Reports durch den Parameter `-r` wurde die Variable `Disp_sup.report` mit Hilfe der Funktion `set_report()` auf den Wert 1 gesetzt. Die Anwahl der Export-Datei soll durch den Wert 2 repräsentiert werden.

```
...
else if (strcmp( argv[i], "-k" ) == 0) {
Disp_sup.set_report( 2 );
continue;
...
```

Datei: app_env.C
Funktion: process_command_options()
Aufgabe: Prüfung und Auswertung des Dateinamens für die Reportdatei
Änderung: Sollte die Generierung eines Reports ausgewählt sein, so ist die Angabe eines Dateinamens obligatorisch. Dies soll analog auch für die Export-Datei geprüft werden.

```
...
if  (( Disp_sup.get_report()) &&
    (( main_win.get_filename( ) == NULL) ||
     ( strncmp( main_win.get_filename(), "\0", 1 ) == 0 )))
     usage();
...

if  (( Disp_sup.get_report_filename() != NULL ) &&
     ( !Disp_sup.get_report( )) )
     usage();
...
```

Datei: app_env.C
Funktion: usage()
Aufgabe: Ausgabe eines Benutzerhinweises
Änderung: Erweiterung der Ausgabe, so daß der Schalter `-k` ausgegeben wird.

```
...
if  (( Disp_sup.get_report()) &&
fprintf(stderr, " -k     Sends a kojak report to stdout. \n");
fprintf(stderr, "        Requires specification of a file name.\n");
fprintf(stderr, "        X options are not valid in this mode.\n");
...
```

Datei: main.C
Funktion: apprentice()
Aufgabe: Aufruf der Funktion zur Generierung des Reports
Änderung: In Abhängigkeit des Wertes von `Disp_sup.report` soll unterschieden werden, ob ein Report oder eine Export-Datei generiert wird.

```
...
if ( Disp_sup.get_report( ) == 1 ) {
    make_report( );
    return ( 1 );
}
else if ( Disp_sup.get_report( ) == 2) {
    do_kojak( );
    return ( 0 );
}
...
```

Anhang B

Apprentice-Exportschnittstelle

Datensatz: *
Beschreibung: Programm- und Testlaufinformationen
Relation(en): `ProgramVersion`, `Test_Run`

Position	Datenfeld	Format
1	Programmname	String
2	Programmdatum	`DD-MON-YY HH:MM:SS`
3	Datum und Zeit des Testlaufs	`DD-MON-YY HH:MM:SS`
4	Anzahl der Prozessoren	Long
5	Taktzyklus	Long

Datensatz: `S`
Beschreibung: Quellcode
Relation(en): `SourceCode`

Position	Datenfeld	Format
1	Dateiname mit Pfadangabe	String
2	Datum und Zeit der Datei	`DD-MON-YY HH:MM:SS`

Datensatz: `F`
Beschreibung: Funktionsdaten
Relation(en): `Function`, `Function_Timing`

Position	Datenfeld	Format
1	Name der Funktion	String
2	Quelldatei	String
2	Datum und Zeit der Quelldatei	`DD-MON-YY HH:MM:SS`
3	Instrumentierungs-Kennzeichen	Long
4	Anzahl der Durchläufe	Long
5	Anzahl der Aufrufpunkte	Long

Datensatz: C
Beschreibung: Funktionsaufrufe
Relation(en): Function_Call, Call_Timing

Position	Datenfeld	Format
1	Name der aufgerufenen Funktion	String
2	Name der aufrufenden Funktion	String
3	Name der aufrufenden Programmregion	String
4	ID-Nummer des Aufrufs	Long
5	Minimale Anzahl von Funktionsaufrufen	Long
6	Nummer des Prozessors mit der minimalen Anzahl von Funktionsaufrufen	Long
7	Maximale Anzahl von Funktionsaufrufen	Long
8	Nummer des Prozessors mit der maximalen Anzahl von Funktionsaufrufen	Long
9	Mittelwert der Anzahl von Funktionsaufrufen	Double
10	Standardabweichung der Anzahl von Funktionsaufrufen	Double
11	Minimale gemessene Zeit für Funktionsaufrufe	Long
12	Nummer des Prozessors mit der minimal gemessenen Zeit	Long
13	Maximale gemessene Zeit für Funktionsaufrufe	Long
14	Nummer des Prozessors mit der maximal gemessenen Zeit	Long
15	Mittelwert der gemessenen Zeit für Funktionsaufrufe	Double
16	Standardabweichung der Zeit für Funktionsaufrufe	Double

Datensatz: N
Beschreibung: Programmregion
Relation(en): ProgRegion

Position	Datenfeld	Format
1	Umgebende Funktion	String
2	Name der Region	String

Datensatz: T
Beschreibung: Summierte Laufzeiten
Relation(en): Total_Region_Timing

Position	Datenfeld	Format
1	Name der Funktion	String
2	Name der Programmregion	String
3	Exklusive Zeit	Long
4	Inklusive Zeit	Long
5	Overhead	Long
6	Geschätzte Zeit	Long

Datensatz: Y
Beschreibung: Laufzeiten nach Apprentice-Zeittypen
Relation(en): Typed_Region_Timing

Position	Datenfeld	Format
1	Name der Funktion	String
2	Name der Programmregion	String
3	Apprentice-Zeittyp	Long
4	Wert der Messung	Long

Datensatz: L
Beschreibung: Quellcode-Referenz
Relation(en): Source_Code_Reference

Position	Datenfeld	Format
1	Name der Funktion	String
2	Name der Programmregion	String
3	Quelldatei	String
4	Datum und Zeit der Quelldatei	DD-MON-YY HH:MM:SS
5	Erste Zeile des Programmblocks	Long
6	Letzte Zeile des Programmblocks	Long

Anhang C

Einrichten der Datenbank

Das zur Erstellung des ER-Diagramms eingesetzte Werkzeug ERDRAW generiert zwei Skripte mit SQL-Anweisungen, die ohne umfangreiche Veränderungen von der interaktiven Schnittstelle des verwendeten DBMS ausgeführt werden können.

Durch das erste Skript werden alle zur Aufnahme der Apprentice-Leistungsdaten erforderlichen Tabellen in der Datenbank angelegt. Dies erfolgt durch `CREATE TABLE`-Anweisungen. Vor der Ausführung muß lediglich die erste Zeile angepaßt werden, indem der Text `your_user_name` durch den eigenen Benutzernamen ersetzt wird. Im einzelnen umfaßt das Skript die folgenden Kommandos:

```
CREATE SCHEMA AUTHORIZATION your_user_name

create table ProgramVersion (
Compilation date NOT NULL,
ProgId varchar2(10) NOT NULL,
Name varchar2(255) NOT NULL)

create table Function (
ProgId varchar2(10) NOT NULL,
instrumented number(1) NOT NULL,
Name varchar2(255) NOT NULL)

create table ProgRegion (
Parent_Name varchar2(255) NULL,
Parent_Function_Name varchar2(255) NULL,
Parent_ProgId varchar2(10) NULL,
ProgId varchar2(10) NOT NULL,
Function_Name varchar2(255) NOT NULL,
Name varchar2(255) NOT NULL)

create table Test_Run (
ProgId varchar2(10) NOT NULL,
testtime date NULL,
pe number NULL,
clockspeed number NULL)

create table Function_Call (
```

```
ProgId varchar2(10) NULL,
Name varchar2(255) NULL,
Caller_ProgId varchar2(10) NOT NULL,
Function_Name varchar2(255) NOT NULL,
Caller_Name varchar2(255) NOT NULL,
Call_ID number NOT NULL)

create table Function_Timing (
testtime date NULL,
ProgId varchar2(10) NOT NULL,
Name varchar2(255) NOT NULL,
num_called_from number NOT NULL,
passes number NOT NULL)

create table SourceCode (
FileName varchar2(255) NOT NULL,
FileDate date NOT NULL,
Code long NULL)

create table Total_Region_Timing (
ProgId varchar2(10) NOT NULL,
Function_Name varchar2(255) NOT NULL,
Name varchar2(255) NOT NULL,
testtime date NULL,
excl number NOT NULL,
incl number NOT NULL,
ovhd number NOT NULL,
est_clocks number NOT NULL)

create table Source_Code_Reference (
FileName varchar2(255) NOT NULL,
FileDate date NOT NULL,
ProgId varchar2(10) NOT NULL,
Function_Name varchar2(255) NOT NULL,
Name varchar2(255) NOT NULL,
beginrange number NOT NULL,
endrange number NOT NULL)

create table Call_Timing (
testtime date NULL,
Caller_ProgId varchar2(10) NOT NULL,
Function_Name varchar2(255) NOT NULL,
Caller_Name varchar2(255) NOT NULL,
Call_ID number NOT NULL,
ncall_min number NOT NULL,
ncall_min_pe number NOT NULL,
ncall_max number NOT NULL,
ncall_max_pe number NOT NULL,
ncall_mean number NOT NULL,
ncall_stdev number NOT NULL,
time_min number NOT NULL,
time_min_pe number NOT NULL,
time_max number NOT NULL,
time_max_pe number NOT NULL,
```

```
time_mean number NOT NULL,
time_stdev number NOT NULL)

create table Typed_Region_Timing (
testtime date NULL,
ProgId varchar2(10) NOT NULL,
Function_Name varchar2(255) NOT NULL,
Name varchar2(255) NOT NULL,
type number NOT NULL,
timing number NULL);
```

Mit Hilfe des zweiten Skriptes werden in den zuvor angelegten Tabellen durch ALTER TABLE-Anweisungen die jeweiligen Primärschlüssel sowie die Verknüpfungen zwischen den Tabellen definiert. Der Zusatz ON DELETE CASCADE bewirkt dabei, daß beim Löschen eines Datensatzes alle anderen Sätze, die diesen über ihren Fremdschlüssel referenzieren, ebenfalls gelöscht werden. Dieses Skript enthält die folgenden Anweisungen:

```
ALTER TABLE ProgramVersion ADD (
PRIMARY KEY (ProgId) );

ALTER TABLE Function ADD (
PRIMARY KEY (ProgId, Name) );

ALTER TABLE ProgRegion ADD (
PRIMARY KEY (ProgId, Function_Name, Name) );

ALTER TABLE Test_Run ADD (
PRIMARY KEY (testtime) );

ALTER TABLE Function_Call ADD (
PRIMARY KEY (Caller_ProgId, Function_Name, Caller_Name, Call_ID) );

ALTER TABLE Function_Timing ADD (
PRIMARY KEY (testtime, ProgId, Name) );

ALTER TABLE SourceCode ADD (
PRIMARY KEY (FileName, FileDate) );

ALTER TABLE Total_Region_Timing ADD (
PRIMARY KEY (ProgId, Function_Name, Name, testtime) );

ALTER TABLE Source_Code_Reference ADD (
PRIMARY KEY (beginrange, endrange, FileName, FileDate, ProgId,
             Function_Name, Name) );

ALTER TABLE Call_Timing ADD (
PRIMARY KEY (testtime, Caller_ProgId, Function_Name, Caller_Name,
             Call_ID) );

ALTER TABLE Typed_Region_Timing ADD (
PRIMARY KEY (type, testtime, ProgId, Function_Name, Name) );

ALTER TABLE Function ADD (
```

```
CONSTRAINT Function_ri_1 FOREIGN KEY (ProgId)
        REFERENCES ProgramVersion ON DELETE CASCADE  );

ALTER TABLE ProgRegion ADD (
CONSTRAINT ProgRegion_ri_1 FOREIGN KEY (Parent_ProgId,
                                        Parent_Function_Name,
                                        Parent_Name)
        REFERENCES ProgRegion ON DELETE CASCADE ,
CONSTRAINT ProgRegion_ri_2 FOREIGN KEY (ProgId, Function_Name)
        REFERENCES Function ON DELETE CASCADE  );

ALTER TABLE Test_Run ADD (
CONSTRAINT Test_Run_ri_1 FOREIGN KEY (ProgId)
        REFERENCES ProgramVersion ON DELETE CASCADE  );

ALTER TABLE Function_Call ADD (
CONSTRAINT Function_Call_ri_1 FOREIGN KEY (ProgId, Name)
        REFERENCES Function ON DELETE CASCADE ,
CONSTRAINT Function_Call_ri_2 FOREIGN KEY (Caller_ProgId, Function_Name,
                                           Caller_Name)
        REFERENCES ProgRegion ON DELETE CASCADE  );

ALTER TABLE Function_Timing ADD (
CONSTRAINT Function_Timing_ri_1 FOREIGN KEY (testtime)
        REFERENCES Test_Run ON DELETE CASCADE ,
CONSTRAINT Function_Timing_ri_2 FOREIGN KEY (ProgId, Name)
        REFERENCES Function ON DELETE CASCADE  );

ALTER TABLE Total_Region_Timing ADD (
CONSTRAINT Total_Region_Timing_ri_1 FOREIGN KEY (ProgId, Function_Name,
                                                 Name)
        REFERENCES ProgRegion ON DELETE CASCADE ,
CONSTRAINT Total_Region_Timing_ri_2 FOREIGN KEY (testtime)
        REFERENCES Test_Run ON DELETE CASCADE  );

ALTER TABLE Source_Code_Reference ADD (
CONSTRAINT Source_Code_Reference_ri_1 FOREIGN KEY (FileName, FileDate)
        REFERENCES SourceCode ON DELETE CASCADE ,
CONSTRAINT Source_Code_Reference_ri_2 FOREIGN KEY (ProgId, Function_Name,
                                                   Name)
        REFERENCES ProgRegion ON DELETE CASCADE  );

ALTER TABLE Call_Timing ADD (
CONSTRAINT Call_Timing_ri_1 FOREIGN KEY (testtime)
        REFERENCES Test_Run ON DELETE CASCADE ,
CONSTRAINT Call_Timing_ri_2 FOREIGN KEY (Caller_ProgId, Function_Name,
                                         Caller_Name, Call_ID)
        REFERENCES Function_Call ON DELETE CASCADE  );

ALTER TABLE Typed_Region_Timing ADD (
CONSTRAINT Typed_Region_Timing_ri_1 FOREIGN KEY (testtime)
        REFERENCES Test_Run ON DELETE CASCADE ,
CONSTRAINT Typed_Region_Timing_ri_2 FOREIGN KEY (ProgId, Function_Name,
                                                 Name)
```

```
REFERENCES ProgRegion ON DELETE CASCADE );
```

Zur automatischen Einrichtung der Datenbank-Tabellen existiert das Perl-Skript `install_kojak_database`, das nach Abfrage der Benutzerdaten durch

- Ergänzung des Benutzernamens,
- Starten von SQL*Plus,
- Ausführen der beiden Skripte und
- Beenden von SQL*Plus

alle zur Installation erforderlichen Schritte automatisch ausführt. Nach dem Ausführen von `install_kojak_database` ist die Datenbank einsatzbereit und kann die vom Import-Skript übergebenen Daten aufnehmen.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] R. Borgeest, B. Dimke, O. Hansen: *A trace based performance evaluation tool for parallel real time systems.* Parallel Computing 21, Nr. 4, 1995, S. 551-564

[2] R. Borgeest, Ch. Rödel: *Trace Analysis with a Relational Database System.* Fourth Euromicro Workshop on Parallel and Distributed Processing, 1996, S. 243-250

[3] P. Chen: *The Entity-Relationship-Model – Towards a Unified View of Data.* ACM Transactions on Database Systems 1, Nr. 1, 1976, S. 9-36

[4] W. Clancey: *Heuristical classification.* Artificial Intelligence 27, 1985, S. 289-350

[5] E. Codd: *A relational model of data for large shared data banks.* Comm. ACM 13, No. 6, 1970, S. 377-387

[6] CRAY Research, Inc.: *Compiler Information File (CIF) Reference Manual.* CRAY Manual SR-2401, 1994

[7] CRAY Research, Inc.: *Introducing the MPP Apprentice Tool.* CRAY Manual IN-2511, 1994

[8] A. Espinosa, T. Margalef, E. Luque: *Automatic Performance Evaluation of Parallel Programs.* Sixth Euromicro Workshop on Parallel and Distributed Processing, 1998

[9] A. Espinosa, T. Margalef, E. Luque: *Automatic detection of parallel program performance problems.* Universitat Autònoma de Barcelona, Computer Science Department, 1998

[10] M. Gerndt, B. Mohr, M. Pantano, F. Wolf: *Automatic Performance Analysis for Cray T3E.* Proceedings of the Seventh Workshop on Compilers for Parallel Computers (CPC'98), University of Linköping, 1998, S. 69-78

[11] W. Gropp, E. Lusk, A. Skjellum: *Using MPI : Portable Parallel Programming With the Message-Passing Interface (Scientific and Engineering Computation).* MIT Press, 1995

[12] B. Helm, A. Malony: *Automating Performance Diagnosis: a Theory and Architecture.* International Workshop on Computer Performance Measurement and Analysis (PERMEAN '95), 1995

[13] J. Hollingsworth: *Finding Bottlenecks in Large Scale Parallel Programs.* Ph. D. Thesis, University of Wisconsin – Madinson, 1994

[14] C. Kilpatrick, K. Schwan: *ChaosMON – Application-Specific Monitoring and Display of Performance Information for Parallel and Distributed Systems.* Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, ACM SIGPLAN Notices 26, Nr. 12, 1991, S. 57-67

[15] P. C. Lockemann, J. W. Schmidt (Hrsg.): *Datenbank-Handbuch.* Springer, Berlin, 1987

[16] E. Maillet: *TAPE/PVM an efficient performance monitor for PVM applications – user guide.* LMC-IMAG Grenoble, 1995

[17] Message Passing Interface Forum: *MPI: Extensions to the Message Passing Interface.* http://www.mpi-forum.org/docs/docs.html, Juli 1997

[18] B. Miller et al.: *The Paradyn Parallel Performance Measurement Tool.* IEEE Computer 28, No. 11, 1995, S. 37-46

[19] J. Nesheiwat, B. Szymanski: *Instrumentation Database for Performance Analysis of Parallel Scientific Applications.* Fourth International Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers (LCR '98), Pittsburgh, Ed. David O'Hallaron, 1998, S. 229-242

[20] Netscape Communications Corp.: *CORBA: Catching the Next Wave.* http://developer.netscape.com/docs/wpapers/corba/index.html

[21] Netscape Communications Corp.: *Netscape Object Signing – Establishing Trust for Downloaded Software.* http://developer.netscape.com/docs/manuals/signedobj/trust/index.htm

[22] Netscape Communications Corp.: http://developer.netscape.com/software/signedobj/index.html

[23] Oracle Corp.: *Programmer's Guide to the Oracle Call Interface, Release 7.3.* Oracle Part No. A32546-1, 1996

[24] Oracle Corp.: *Programmer's Guide to the Oracle Precompilers, Release 1.8.* Oracle Part No. A42525-1, 1996

[25] R. Orfali, D. Harkey: *Client/Server Programming with JAVA and CORBA.* John Wiley & Sons, Inc., New York, 1997

[26] R. Snodgrass: *A Relational Approach to Monitoring Complex Systems.* ACM Transactions on Computer Systems 6, Nr. 2, 1988, S. 157-196

[27] E. Szeto, V. Markowitz: *ERDRAW 5.3 – A Graphical Editor for Extended Entity-Relationship Schemas.* Technical Report LBL-PUB-3048, Lawrence Berkeley Laboratory, 1993

[28] G. Vossen: *Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme.* Addison-Wesley, Bonn, 1987

[29] L. Wall, T. Christiansen, R. L. Schwartz: *Programming in Perl.* 2nd edition, O'Reilly & Associates Inc., 1996

[30] F. Wolf, B. Mohr: *EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs.* Technischer Bericht FZJ-ZAM-IB-9803, Forschungszentrum Jülich, 1998

Danksagung

Die vorliegende Arbeit wurde als Diplomarbeit in Fach Informatik am Lehrstuhl für Technische Informatik und Computerwissenschaften der Rheinisch-Westfälischen Technischen Hochschule (RWTH) Aachen erstellt.

Dem Lehrstuhlinhaber Herrn Prof. Dr. Friedel Hoßfeld danke ich für die Möglichkeit, die Arbeit am Zentralinstitut für Angewandte Mathematik (ZAM) des Forschungszentrums Jülich (FZJ) anfertigen zu können. Herrn Prof. Dr. Thomas Bemmerl danke ich für die Übernahme des Zweitgutachtens.

Außerordentlicher Dank gilt Herrn Dr. Michael Gerndt für die ausgezeichnete Betreuung, für seine stete Ansprechbarkeit und für die vielen interessanten Diskussionen, die die Arbeit positiv beeinflußt haben.

Herrn Walter Elmenhorst und Herrn Bernd von Studnitz danke ich für ihre Hilfsbereitschaft in Datenbankfragen. Ebenso bedanke ich mich bei Frau Anke Häming, die mir bei Problemen mit der Programmierung in Java half und mir ihre Klassen zur Quelltext-Anzeige überlassen hat. Weiterhin danke ich Frau Astrid Göke und Herrn Dietmar Jansen, die für alle Fragen und Probleme während der Erstellung dieser Arbeit stets ein offenes Ohr hatten.

Besonderers bedanken möchte ich mich außerdem bei meiner Freundin und Kollegin Tamara Knödler, die während der Abschlußphase dieser Arbeit viel Verständnis aufgebracht hat und mir durch wertvolle Anregungen zur Seite stand.

Vor allem möchte ich mich bei meiner Familie bedanken, die mir durch ihre Unterstützung mein Studium ermöglicht hat und deren Hilfe wesentlich zum Gelingen dieser Arbeit beigetragen hat.