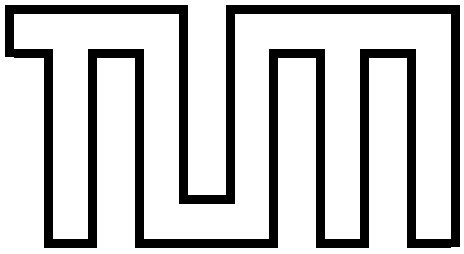

Technische Universität München

Fakultät für Informatik

Entwicklung eines komponentenbasierten Designs zur interaktiven Visualisierung und Steuerung von Profiling-Messungen

Diplomarbeit

Stefan Hamerl

Aufgabensteller: Prof. Dr. Arndt Bode

Betreuer: Dr. Josef Weidendorfer

Abgabedatum: 15. Juli 2004

Erklärung

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

15.07.2004

Stefan Hamerl

Inhaltsverzeichnis

1 Einleitung **4**

2 Grundlagen des Profilings **6**
2.1 Methoden 7
2.2 Profiling-Daten 8
2.3 Profiling-Werkzeuge 9
2.3.1 gprof 9
2.3.2 OProfile 10
2.3.3 PerfCtr 10
2.3.4 pfmon 10
2.3.5 PAPI 10
2.3.6 DynInst/DPCL/Dynaprof 11
2.3.7 VTune 11
2.4 Valgrind 11
2.4.1 Cachegrind 12
2.4.2 Calltree 12
2.5 Visualisierungswerkzeuge 12
2.5.1 HPCView 12
2.5.2 VProf 12
2.5.3 KCachegrind 13

3 Analyse der Aufgabenstellung **14**
3.1 Ausgangssituation 14
3.1.1 Hintergrund 14
3.1.2 Funktionsumfang 15
3.1.3 Anwendungsbereich 15
3.2 Anforderungen 16
3.2.1 Ziele 16
3.2.2 Benutzerfreundlichkeit 16
3.2.3 Zusammenspiel mit anderen Anwendungen 17
3.2.4 Mehrbenutzerfähigkeit 17
3.2.5 Sicherheitsanforderungen 18
3.2.6 Portierbarkeit, Standards 18

3.3 UML . . . 18
3.4 Szenarien . . . 19
3.5 Use Cases . . . 21
3.5.1 Aktoren . . . 21
3.5.2 System: KCachegrind . . . 21
3.6 Architektur . . . 24
3.6.1 Datenimport . . . 25
3.6.2 Datenmodell . . . 25
3.6.3 Visualisierung . . . 28

4 Konzept **31**
4.1 Grobarchitektur . . . 31
4.2 Datenmodell . . . 32
4.3 Datenimport . . . 35
4.4 Profiling-Steuerung . . . 35
4.5 Visualisierungserweiterung . . . 36

5 Realisierung **38**
5.1 Plugin-System . . . 38
5.2 Schnittstellen . . . 39
5.2.1 Schnittstelle: Daten-Import . . . 39
5.2.2 Schnittstelle: Interaktives Profiling . . . 40
5.2.3 Schnittstelle: Visualisierung . . . 41

6 Fazit **42**
6.1 Zusammenfassung . . . 42
6.2 Ausblick . . . 43

Anhang **44**

A TraceCost-Klassen **44**

B Datenmodell-Klassen **46**

Literaturverzeichnis **51**

Abbildungsverzeichnis **54**

Kapitel 1

Einleitung

Im Jahre 1969 begann Intel im Rahmen eines Projektes für den japanischen Rechner-Hersteller Busicom mit der Entwicklung des ersten Mikroprozessors. Der Auftrag bestand in der Entwicklung von Chips für eine Produktreihe programmierbarer Rechenmaschinen, wie man sie heute hinter dem Kassenschalter eines jeden Supermarktes finden kann. Ergebnis war der Intel 4004, der erste in Serie gefertigte Mikroprozessor mit dem die ganze Prozessorszenerie ihren Anfang nahm. Dieser besaß eine Datenbreite von 4-Bit, bestand aus 2.300 Transistoren und lief anfänglich bei einer Taktfrequenz von 108kHz[1].
Schnell wurden die unbegrenzten Möglichkeiten und zahlreichen Anwendungsgebiete des Mikroprozessors erkannt. So wurde der Intel 4004 zum Beispiel in den Raumsonden Pioneer 10 und Pioneer 11 verwendet.
Bereits 1965 sagte Gordon E. Moore voraus, daß sich die Anzahl der Transistoren auf einem Chip jährlich etwa verdoppeln wird, also exponentiell wächst. Nachzulesen in [1]. Damit ginge natürlich auch eine Erhöhung der Rechenleistung einher. Die Entwicklung der Halbleitertechnologie folgte bis heute dieser Regel, und wird es voraussichtlich in den nächsten Jahren auch weiterhin tun.
Für die Leistungsfähigkeit eines Rechners ist aber nicht nur der Prozessor relevant, sondern auch das Speichersystem. Um nicht als *„Bottleneck“* zu fungieren müsste der Rechner einen einheitlichen Speicher mit ausreichender Kapazität und so kurzer Zugriffszeit haben, daß der Prozessor, um mit seiner Verarbeitungsgeschwindigkeit schrittzuhalten, mit Befehlen und Daten aus dem Speicher versorgt werden kann. Dies ist aus technischen und wirtschaftlichen Gründen allerdings nicht realisierbar.
Deshalb wird eine mehrstufige Speicherhierarchie eingesetzt, bei der jede Stufe kleiner, schneller und dadurch auch teurer als die nächste Stufe ist. Die Hierarchie besteht dabei aus drei Stufen, dem Prozessor am nächsten sind Register- und Pufferspeicher, gefolgt vom Hauptspeicher bis hin zu langsamen Hintergrundspeichern, die über eine große Ka-

[1] als er auf den Markt kam lief er bereits mit 500kHz, später mit 740kHz

pazität verfügen.
Caches sind die schnellsten aber eben auch kleinsten Pufferspeicher die zur Verfügung stehen. Sie befinden sich zwischen der CPU und dem Hauptspeicher des Systems. Ihre Aufgabe ist es, den Prozessor mit den am häufigsten verwendeten Anweisungen und Daten zu versorgen.
Damit Caches effizient genutzt werden, sind Programme im Hauptspeicher so abzulegen, daß möglichst viele Code- und Datenanteile mit Blockzugriffen vorausgreifend in den Cache geladen und darin möglichst lange gehalten und wiederverwendet werden können. Dies gewinnt immer mehr an Bedeutung, da die Entwicklung im Speicherbereich nicht mit der im Bereich der Prozessoren mithalten kann. Der Geschwindigkeitsunterschied wird also immer größer und somit ein effizientes Speicherzugriffsverhalten für die Leistungsbewertung von Anwendungen immer wichtiger.
Dies berücksichtigen sogenannte „Profiling“-Werkzeuge. Sie messen beispielsweise die von einer Funktion beanspruchte Prozessorzeit oder die Anzahl der verursachten Cache-Misses (Fehlzugriffe). Als Ergebnis liefern die meisten dieser Programme allerdings nur eine Textdatei, welche eine nach Kosten sortierte Liste der Funktionen enthält. Die Ausgabedatei wird jedoch schnell sehr unübersichtlich. Deshalb gibt es Werkzeuge, die eine grafische Oberfläche zur Verfügung stellen und die gemessenen Daten graphisch aufbereiten.
Ein solches, noch in der Entwicklung befindliches Werkzeug ist KCachegrind, welches im Rahmen der vorliegenden Diplomarbeit überarbeitet und ein Stück verbessert werden soll. Es wurden einige Nachteile am aktuellen Design festgestellt, welche die Weiterentwicklung beeinträchtigen. Das betrifft speziell die Erweiterbarkeit des Programms, was durch die Entwicklung eines Plugin-Systems beseitigt werden soll.
Dazu wird in Kapitel 2 beschrieben, was man als Profiling bezeichnet, welche Möglichkeiten es bietet und wofür es eingesetzt wird. Darüberhinaus wird eine Übersicht über eine Auswahl der zur Verfügung stehenden Profiling- und Visualisierungswerkzeuge gegeben.
Im 3. Kapitel wird analysiert, welche Anforderungen an KCachegrind generell gestellt werden und welche es momentan erfüllt, beziehungsweise welche noch nicht. Dies beinhaltet eine nähere Betrachtung der Architektur.
Basierend auf den durch die Analyse gewonnenen Erkenntnissen, wird in Kapitel 4 ein prinzipielles Konzept erarbeitet, welche Änderungen an der Architektur möglich und sinnvoll sind, um die gegebenen Anforderungen zu erfüllen.
In Kapitel 5 wird das Konzept speziell für KCachegrind ausgearbeitet. So wird ein Erweiterungsystem mit den zugehörigen Schnittstellen detailliert vorgestellt.
Neben einer kurzen Zusammenfassung findet sich in Kapitel 6 dann noch ein Ausblick in Bezug auf die mögliche Weiterentwicklung von KCachegrind.

Kapitel 2

Grundlagen des Profilings

Profiling ist das Sammeln Quellcode-bezogener Statistikdaten zum Ablaufverhalten eines Programmes zur Laufzeit mit dem Ziel, Zeitbedarf und Resourcenanforderungen von Programmbereichen zu erfassen. Als *Profiler* bezeichnet man ein Werkzeug, welches diese Informationen über das Laufzeitverhalten sammelt.

Die Entwicklung einer Software kann in verschiedene Phasen aufgeteilt werden. Eine davon ist der Optimierungsprozeß, ein iterativer Prozeß, dessen Ziel es unter anderem ist, die *„Hot Spots“* und *„Bottlenecks“* des Programmes zu finden, da dort Codeoptimierungen am effektivsten sind:

- „Hot Spots“ sind Programmabschnitte in einem Programm, die übermäßig viel Rechenzeit in Anspruch nehmen.
- Als „Bottleneck“ bezeichnet man einen Abschnitt im Programm, der die zur Verfügung stehenden Betriebsmittel ungleichmäßig nutzt und dadurch unnötige Verzögerungen verursacht.

Um diese Engpässe zu finden, setzt man einen Profiler ein. Dieser nimmt zur Laufzeit des Programmes verschiedene Messungen vor, anhand derer ein Programmprofil erstellt wird. Moderne Prozessoren stellen dazu verschiedene Ereigniszähler für die Ermittlung der Hardware-Auslastung zur Verfügung, die ausgelesen werden können. Ereigniszähler sind Register, die das Auftreten bestimmter Signale zählen, die mit der Funktionsweise des Prozessors zusammenhängen. So können benötigte Rechenzeit und Ausführungshäufigkeit für bestimmte Programmabschnitte ermittelt und die aufgetretenen Hardware-Ereignisse, wie beispielsweise Cache-Fehlzugriffe, zugewiesen werden. Mehr dazu in Kapitel 2.2.
Messungen können mit unterschiedlicher Granularität erfolgen. Vom Programm über Funktionen bis hin zu einzelnen Maschinenbefehlen. Um die Zähler auszulesen, muß allerdings ein entsprechender Code in das eigentliche Programm eingefügt werden. Ideal wäre eine eigene Hardware mit direkten Signalleitungen für den Abtransport der Daten. Da

dies aber viel zu kostspielig ist, geht man den Kompromiß ein, daß die Meßdaten durch das Meßverfahren beeinflußt werden. Je detaillierter eine Messung sein soll, desto größer ist der zusätzliche Aufwand und somit auch die Verfälschung der Messung.

2.1 Methoden

Es gibt folgende Profiling-Techniken, die sich in Genauigkeit und Aufwand unterscheiden:

- *Exakte Messung*: Bei dieser Methode wird der zum Auslesen der Zähler benötigte Code in den Quellcode des eigentlichen Programmes integriert. Dies bezeichnet man auch als Instrumentierung. Die Messungen werden also von der Anwendung selbst durchgeführt und erhöhen die benötigte Rechenzeit. Selbst bei Messungen geringer Granularität kann sich dadurch der Gesamtaufwand verdoppeln.

- *Statistische Messung*: Eine Art der statistischen Messung ist instrumentierendes Sampling. Die dieser Annahme zugrundeliegende Methode ist, daß die Verteilung der Ereignisse eines Typs über den Programmcode dieselbe ist, wenn man statt jeden Ereignisses nur jedes n-te mißt (für n ¡¡ N mit N Anzahl aller Ereignisse dieses Typs über eine Programmausführung). Nach dem Zählen von n Ereignissen durch einen Ereigniszähler wird der Programmzähler (PC) vermerkt. Zusammen mit den Informationen des Compilers[1] kann ein Ereignis einem bestimmten Programmabschnitt zugeordnet werden. Dieser Ansatz bedeutet kaum zusätzlichen Aufwand, dafür ist allerdings die Genauigkeit geringer als bei einer Messung durch Intstrumentierung. Ein anderer Ansatz ist „Ephemeral Instrumentation“. Hierbei wird die instrumentierte Version einer bestimmten Funktion nur in festlegbaren, periodischen Zeitintervallen ausgeführt, wodurch sich der zusätzliche Aufwand regulieren lässt. Mehr dazu ist in [17] nachzulesen.

- *Simulation*: Der Programmcode wird in einer durch Software simulierten CPU[2] ausgeführt, die auftretende Ereignisse selbst generiert. Es können neue Ereignistypen eingeführt werden, weshalb bei der Beobachtung keine Grenzen gesetzt sind. Durch den Instrumentierungsaufwand beim Meßverfahren der Laufzeitinstrumentierung wird das reale Zeitverhalten eines Programmes verfälscht. Da eine Simulation nicht auf tatsächliche Zeiten angewiesen ist, eignet sie sich gerade für dieses Verfahren.

Ausführlichere Erklärungen der Profiling-Methoden sind in [3] und [4] zu finden.

[1] Ein Programm, das einen Quelltext einer Programmiersprache in die vom Prozessor ausführbare Maschinensprache, auch nativer Code genannt, übersetzt.

[2] Central Processing Unit, (Haupt-)Prozessor eines Systems

2.2 Profiling-Daten

Profiling-Daten können eine ganze Reihe verschiedener Daten-Typen sein, zum Beispiel Ereignis-Logs, Ausführungshäufigkeiten, Betriebsmittelzuteilung und noch vieles mehr. Es gibt viele Wege, Performanz-relevante Daten zu gewinnen. Da aber laut Definition Profiling-Daten zur Laufzeit gesammelt werden müssen, sind die zur Verfügung stehenden Möglichkeiten begrenzt.
Der einfachste Weg ist das Auslesen von *Ereigniszählern*. Diese stehen erst seit ein paar Jahren zur Verfügung und stellen eine Hardware-Unterstützung für Software-basierte Profiler da.
Das einfachste Mittel ist ein hochauflösender Zeit-Zähler, beispielsweise der TSC[3] des Pentium. Dies ermöglicht eine genaue Bestimmung der für eine Operation notwendigen Zeit.
Weitaus detailliertere Informationen bieten Ereigniszähler. Das sind Register, die performanz-relevante Ereignisse, wie zum Beispiel Cache-Fehlzugriffe oder Fließkommaberechnungen mitzählen. Die zur Verfügung stehenden Zähler unterscheiden sich allerdings je nach Prozessor-Hersteller, weshalb hier auch auf eine detaillierte Beschreibung verzichtet wird.

Bezüglich der Ereignisse, die während der Ausführung eines bestimmten Programmabschnitts auftreten, unterscheidet man zweierlei Arten:

- *Inklusivkosten* sind die Anzahl der Ereignisse, die während der Ausführung des Programmabschnitts sowie während aller von ihm aufgerufenen Funktionen auftreten.
- *Exklusivkosten* sind die Anzahl der Ereignisse, die während der Ausführung des Programmabschnitts selbst auftreten.

Eine sinnvolle Hierarchie an Programmabschnitten besteht aus Programm, Funktionen, Anweisungen und Maschinenbefehlen. Je nachdem, welchen Typ man wählt, erhält man unterschiedlich detaillierte Informationen. Der Aufwand für das Profiling nimmt mit jeder Detailstufe zu.

Prinzipiell können von einem Profiler folgende Daten ermittelt werden: Die einfachste Art von Information ist die *Summe der aufgetretenen Ereignisse*, sie gibt einen Hinweis auf die Arbeitslast.
Dazu verwandt sind *Tracefiles*. Sie enthalten eine mit Zeitstempel versehene Liste der aufgetretenen Ereignisse. Dies ermöglicht es, Ereignisse bestimmten Programmabschnitten zuzuordnen und somit die *Kosten eines Programmabschnitts* zu ermitteln. Ferner ist denkbar, Ereignisse auch Datenstrukturen zuzuordnen.
Besonders interessant sind *Zeit-basierte Werte*, da der Anwender an ihnen sehr schnell

[3] Time Stamp Counter

sieht, welche Programmabschnitte sich zu betrachten lohnen. Bei einem Programmabschnitt, der kaum Zeit verbraucht, ist eine Optimierung nicht lohnend, die dafür aufzuwendende Zeit ist an anderer Stelle effektiver einsetzbar.
Der Programmablauf wird durch Aufrufgraphen dargestellt, für die Informationen über die Aufruffolge der Funktionen notwendig sind. Also welche Funktionen von einer Funktion aufgerufen werden.

Fassen wir noch einmal kurz zusammen, welche Informationen ein Profiler liefern kann:

- Anzahl der Ereignisse eines Typs für einen Programmabschnitt beziehungsweise eine Datenstruktur
- Ausführungshäufigkeiten eines Programmabschnitts
- Ausführungszeiten eines Programmabschnitts
- Aufruffolgen von Funktionen

2.3 Profiling-Werkzeuge

Im folgenden soll ein kurzer Überblick über eine Auswahl an Profiling-Werkzeugen gegeben werden. Die Liste ist natürlich bei weitem nicht vollständig, umfasst allerdings die bekanntesten Werkzeuge. Alle Programme sind von einer Shell aus startbar, wobei verschiedene Parameter übergeben werden können oder müssen. Von besonderem Interesse sind die Informationen die der jeweilige Profiler liefert.

2.3.1 gprof

gprof[4] ist ein Profiler, der auf jedem Unix/Linux-System zu finden ist. Zur Gewinnung der Profiling-Daten wird statistisches Sampling verwendet. Es können 2 verschiedene Ausgaben erzeugt werden. Zum einen das „Flat Profile“. Dies ist eine Tabelle, welche die Namen der Funktionen mit ihren Ausführungszeiten und -Häufigkeiten beinhaltet. Zum anderen einen Aufrufgraphen, der die Aufruffolgen in Form von Aufrufer und aufgerufener Funktion beschreibt. Auch hier wird Zeitverbrauch und Aufrufhäufigkeit einer Funktion beziehungsweise der von ihr aufgerufenen Funktionen dargestellt. Außerdem werden vorhandene Zyklen nach beteiligten Funktionen, der verbrauchten Zeit und der Ausführungshäufigkeit des Zyklus aufgeschlüsselt. Weitere Informationen sind in der Anleitung zu gprof zu finden, Shellbefehl: `man gprof`.

[4]GNU Profiler

2.3.2 OProfile

OProfile läuft unter Linux und unterstützt mehrere Architekturen, darunter Pentium, Itanium und die AMD Athlon Familie. Als Meßmethode wird statistisches Sampling verwendet. Dabei kann ein systemweites Profiling durchgeführt werden, also Profiling Daten für alle Prozesse eines Systems gesammelt werden. OProfile kann annotierten Assemblercode und zu gprof kompatible Daten liefern, die dem „Flat Profile“ entsprechen. Außerdem können noch Aufrufgraph-Daten erzeugt werden, die sich jedoch die Aufruffolge und die Anzahl der in einer Funktion durchgeführten Samples beschränkt. Mehr Informationen zu OProfile siehe [19] und [20].

2.3.3 PerfCtr

PerfCtr[5] ist ein Kernelpatch, der mehrere Register zum Auszählen der Hardwarezähler eines Prozessors bietet.

2.3.4 pfmon

pfmon ist ein von Hewlett-Packard für IA-64 entwickeltes Profiling-Werkzeug. Es bietet neben dem Auszählen einfacher Ereignisse auch die Möglichkeit statistisches Sampling durchzuführen. Die Profiling-Daten können für Binärcode, als auch für ein komlettes System gewonnen werden. Mehr dazu ist auf der Homepage [21] zu finden.

2.3.5 PAPI

PAPI[6] ist der Versuch, eine einheitliche Schnittstelle zum Auslesen von Ereigniszählern für die verschiedenen Plattformen und Prozessoren zur Verfügung zu stellen. Dabei wird auf PerfCtr aufgesetzt und zwei Schnittstellen zur Verfügung gestellt.
Die untere Schnittstelle, auch als „Low Level Interface“ bezeichnet, erlaubt das Gruppieren von Hardwareereignissen. Dadurch können verschiedene Ereignisse gemessen und zueinander in Beziehung gesetzt werden. Die Gruppen sind durch den Anwender programmierbar. Das „High Level Interface“ ermöglicht einfache Aktionen, wie das Starten, Stoppen und Auslesen für eine bestimmte Menge an Ereignissen. PAPI unterstützt also statistisches Profiling. Für eine Liste der unterstützten Plattformen und weiteren Details, siehe [24].

[5] Performance Counter
[6] Performance Application Programming Interface

2.3.6 DynInst/DPCL/Dynaprof

DynInst ist eine API[7]zur Intsrumentierung von Code. Unterstützt werden sowohl IA32 als auch IA64. Siehe auch [25].

DPCL[8] ist eine Objekt-basierte C++-Klassen-Bibliothek, die für Profiling-Werkzeuge die Technik des dynamischen Instrumentierens zur Verfügung stellt. Dabei kann der Instrumentierungs-Code zur Laufzeit der Anwendung hinzugefügt oder entfernt werden. Details siehe [26].

DynaProf ist ein Profiling-Werkzeug, das auf DynInst und DPCL aufsetzt und die Instrumentierung zur Laufzeit der Anwendung vornimmt. Zur Messung wird dynamische Instrumentierung verwendet. Meßbar sind sowohl Ausführungszeiten in Echtzeit, als auch alle von PAPI zur Verfügung gestellten Hardwareereignisse.
Dynamische Instrumentierung hat unter anderem den Vorteil, daß der Code nicht bei jeder Änderung der Instrumentierung neu übersetzt werden muß und daß der instrumentierende Code die Übersetzung nicht beeinflußt. Mehr dazu siehe [27].

2.3.7 VTune

VTune wird von Intel entwickelt und steht sowohl für Linux als auch Windows zur Verfügung. Es ist ein Werkzeug zur Performanz Analyse von Programmen und bietet neben einer Profiler- auch eine Visualisierungskomponente.
Möglich ist zeit- und Ereignis-basiertes, systemweites Sampling. An Visualisierungen werden ein „Counter Monitor“ zur Anzeige der gemessenen Ereignisse und ein Aufrufgraph geboten. Für weitere Informationen siehe [28].

2.4 Valgrind

Valgrind ist eine Software, die eine x86 CPU simuliert. Die Architektur ist modular, so daß verschiedene Komponenten für Messungen verwendet werden können, die unterschiedliche Schwerpunkte legen. Eine Komponente ist beispielsweise „memcheck“, die sich mit der Erfassung von Speicherverwaltugnsproblemen beschäftigt. Weitere Komponenten sind unter anderem cachegrind, addrcheck und helgrind. Alle Messungen basieren auf der Methode der Simulation. Die gelieferten Profiling-Daten sind je nach der verwendeten Komponente unterschiedlich, deshalb siehe [13] für Details.

[7]Application Program Interface
[8]Dynamic Probe Class Library

2.4.1 Cachegrind

Cachegrind ist ein Kommandozeilen-basierter Profiler, der Valgrind erweitert und sich auf Cache-Misses spezialisiert. Es werden unter anderem die L1- und L2-Cache-Misses gemessen. Weitere Informationen siehe [29].

2.4.2 Calltree

Calltree ist eine weitere Komponente für Valgrind, die auf Cachegrind basiert. Es erweitert die Ausgabe von Cachegrind um Daten für Aufrufgraphen, wie beispielsweise Aufrufer und aufgerufene Funktion. Dabei werden Zyklen, rekursive Aufrufe hervorgehoben.

2.5 Visualisierungswerkzeuge

Sofern das Profiling-Werkzeug keine eigene Visualisierung besitzt, wird eine Text-Datei erzeugt, welche die Ergebnisse der Messung enthält. Dies sind üblicherweise tabellarisch dargestellte Informationen zu den Kosten der einzelnen Programmabschnitte und die Aufrufbeziehungen zwischen den Funktionen. Je komplexer das Programm, desto unübersichtlicher wird diese Darstellungsform. Deshalb gibt es spezielle Visualisierungs-Werkzeuge, die dem Benutzer das Analysieren der Profiling-Daten erheblich erleichtern. Für codebezogene Darstellung der Ereignisse werden zum Beispiel Funktionslisten, attributierte Aufrufgraphen und annotierter Sourcecode verwendet. Ausführlichere Erklärungen der Visualisierungs-Techniken sind in [3] und [4] zu finden.

2.5.1 HPCView

HPCView ist ein Visualisierungswerkzeug, das Profiling-Daten und zugehörigen Quellcode darstellen kann. Für die Messungen der Profiling-Daten werden *ssrun*, *uprofile* und *PAPI* verwendet. Der Anwender kann die Metriken, die berechent werden sollen, selbst definieren. Ausführlichere Informationen siehe [30] und [31].

2.5.2 VProf

Der „Visual Profiler“, *VProf*, ist ein Werkzeug Peformanz-Analyse. Es bietet verschiedene Möglichkeiten Profiling-Daten zu gewinnen. Standardmäßig wird die Unix-Subroutine `profil(3)` zum statistischen Sampling des Programmzählers verwendet. Statistiken über die Ereigniszähler des Prozessors werden durch *PAPI* gewonnen. Alternativ kann auch

PerfCtr direkt eingesetzt werden.
Die Visualisierung der Profiling-Daten beschränkt sich auf eine Funktionsliste, die nach Datei, Funktion und Zeile sortiert werden kann und wahlweise in einer grafischen Oberfläche oder auf der Konsole ausgegeben wird. Mehr dazu siehe [32].

2.5.3 KCachegrind

KCachegrind[5] ist das Visualisierungswerkzeug mit dem sich die vorliegende Diplomarbeit beschäftigt. Die zugrundeliegende Architektur zur Darstellung von Profiling-Daten ist Profiler-unabhängig. Bisher werden nur wenige Formate unterstützt, darunter das von Calltree. An Visualisierungsformen bietet es nach Ereigniszahl sortierte Funktionlisten, Aufrufgraphen sowie annotierten Source- und Assemblercode. Im nächsten Kapitel wird KCachegrind detaillierter beschrieben.

Kapitel 3

Analyse der Aufgabenstellung

Gegenstand dieses Kapitels ist die Analyse von KCachegrind. Es erfolgt eine Analyse der Anforderungen und der aktuellen Architektur. Dabei werden auch diverse nichtfunktionale Nebenbedingungen, wie Benutzerfreundlichkeit und Sicherheitsanforderungen diskutiert. Die aufgezeigten Schwachstellen sind Ansatzpunkt für die im nächsten Kapitel erarbeiteten Verbesserungsvorschläge.

3.1 Ausgangssituation

3.1.1 Hintergrund

KCachegrind wird im Rahmen des Projekts DiME[1] eingesetzt, welches die Möglichkeiten von Cache-Optimierung bei iterativen Verfahren, zum Beispiel Multigrid-Verfahren (effiziente Gleichungslöser) untersucht. Solche Verfahren sind besonders speicherlastig und hängen wesentlich von der Geschwindigkeit des Speicherzugriffs ab.
Zur Untersuchung des Cachezugriffsverhaltens von Programmen wird Calltree verwendet, ein Cachesimulator für Linux/IA-32, der auf Valgrind[13] basiert. Dieses Werkzeug soll im Rahmen von DiME erweitert werden.

KCachegrind ist eine in C++ geschriebene KDE/QT-Anwendung und als solche auf Linux, Unix, Mac-OS-X und dank Cygwin[14] auch unter Windows lauffähig. Es entstand als Visualisierungswerkzeug für die von Calltree erzeugten Daten und soll ebenfalls im Rahmen von DiME weiterentwickelt werden. Bisher wird nur das Datenformat von Calltree unterstützt.
Vom Funktionsumfang her ist die codebezogene Darstellung von Ereignissen bereits recht

[1] Data Local Iterative Methods For The Efficient Solution of Partial Differential Equations, [22]

gut realisiert. Aufrufgraphen, nach Ereigniszahl sortierte Funktionlisten sowie annotierter Source- und Assemblercode können bereits dargestellt werden. Es fehlt allerdings noch die Darstellung des Kontrollflusses beziehungsweise der Schleifenstruktur von Funktionen. Ebenfalls noch nicht möglich ist die Darstellung von Kosten für Datenstrukturen. Es gibt zwar derzeit noch kaum ein Profiling-Werkzeug, das diese Daten liefern kann, aber da datenstrukturbezogene Ereignisse interessant für das Profiling sind, kann sich dies in naher Zukunft ändern.
Problematisch ist dabei der interne Aufbau von KCachegrind. Es gibt weder für den Datenimport, noch für die Visualisierung eine klare Schnittstelle, über welche diese erweiterbar wären. Dadurch werden Erweiterungen erheblich erschwert. Aufgabe der vorliegenden Diplomarbeit ist es, ein Lösungskonzept zu entwickeln, so daß man relativ einfach Komponenten hinzufügen kann. Dabei kann es sich um Importfilter für neue Formate, interaktives Steuern von Profiling-Werkzeugen oder Erweiterungen der Visualisierung handeln.

3.1.2 Funktionsumfang

An und für sich ist KCachegrind unabhängig, also nicht auf ein spezielles Profiling-Werkzeug zugeschnitten. Es erlaubt die Visualisierung beliebiger Ereignistypen. Momentan wird nur das Dateiformat von Calltree unterstützt, es gibt allerdings Konverterskripte für OProfile, Perl und PHP. Die Visualisierung umfasst verschiedene Detailebenen. Bei der Erzeugung des Aufrufgraphen kann durch Angabe eines Wertes eingestellt werden, welchen Mindestanteil eine Funktion an der Gesamtausführungszeit haben muß, um im Aufrufgraphen dargestellt zu werden. Dadurch bleibt der Graph auch bei komplexen Programmen übersichtlich und bietet schnellen Zugriff auf die verschiedenen Elemente der Aufrufhierarchie. Die Funktionslisten lassen sich nach Ereignistyp und Anzahl sortieren. Unterste Ebene bildet die Visualisierung von annotiertem Source- und Assemblercode.

3.1.3 Anwendungsbereich

Aktuell:
Als Visualisierungstool für Profiling-Daten unterstützt KCachegrind den Programmierer bei der Auswertung der gewonnenen Ergebnisse. An und für sich ist es auf kein spezielles Profiling-Werkzeug zugeschnitten, unterstützt momentan aber nur das Datenformat von Calltree. Mittlerweile gibt es Konvertierungsskripte für andere Profiling-Werkzeuge, welche die von ihnen erzeugten Profiling-Daten ins Calltree-Format übersetzen und sich so in KCachegrind betrachten lassen.

3.2 Anforderungen

3.2.1 Ziele

Das momentane Anwendungsspektrum von KCachegrind ist noch recht begrenzt. Die vorliegende Diplomarbeit soll Änderungen am Design vorschlagen, so daß die Grundlagen für die Anbindung von Importfiltern und die Ansteuerung von Profiling-Werkzeugen zur Verfügung stehen. KCachegrind entwickelt sich also immer mehr zu einer Art Schaltzentrale für den Optimierungsprozeß.

Das Design von KCachegrind soll so überarbeitet werden, daß folgende Eigenschaften erfüllt werden:

- Es soll ein komponentenbasiertes Design entstehen, d.h. es gibt Schnittstellen über die das System erweiterbar ist.
- Das interne Datenmodell muß dahingehend erweitert werden, daß
  - mehrere Ansichten derselben Profiling-Daten möglich sind,
  - datenstrukturbezogene Kosten verarbeitet werden können,
  - große Mengen an Profilingdaten ohne lange Wartezeiten geladen werden.
- Grundlagen für die Interaktive Steuerung von Profiling-Tools sind zu schaffen, dies beinhaltet:
  - die Schnittstelle zum Anwender (Oberflächenelemente),
  - sowie die Schnittstelle zum Profiling-Tool (Steuerung).

3.2.2 Benutzerfreundlichkeit

3.2.2.1 Performance und Antwortzeiten

Ein wesentliches Kriterium, das über die Akzeptanz einer Anwendung entscheidet, sind die Antwortzeiten der einzelnen Teilfunktionalitäten. Falls Wartezeiten über einer Sekunde bei der Verarbeitung entstehen, sollte der Anwender über die aktuellen Fortschritte informiert werden.
KCachegrind ermöglicht ein zügiges Arbeiten. Es gibt jedoch zwei kritische Punkte die Wartezeiten verursachen können.
Dies ist zum einen das Einlesen einer sehr großen Profiling-Datei. Die Größe dieser Datei hängt vom Umfang des Codes ab und bewegt sich üblicherweise im Rahmen von 50 Kilobyte bis 1 Megabyte. Für eine KDE-Anwendung, die QT- und KDE-Bibliotheken

verwendet, kann diese aber etwa 10 Megabyte betragen. Es kommt in einem solchen Fall zu einer längeren Wartezeit, da alle Informationen auf einmal eingelesen werden. Das soll durch eine spezielle Ladetechnik behoben werden, bei der zunächst nur die Grobstruktur importiert und Detailinformationen bei Bedarf nachgeladen werden.
Zum Anderen kann eine Vielzahl an Funktionen und Funktionsaufrufen zu enormen Wartezeiten bei der Erzeugung des graphischen Layouts des Aufrufgraphen führen. Hierfür wird die Funktionalität eines Fremdprogrammes benutzt, weshalb derzeit keine Verbesserungsmöglichkeit denkbar ist.

3.2.2.2 Benutzeroberfläche

Die Benutzeroberfläche von KCachegrind genügt bereits allen Ansprüchen im Bezug auf Benutzerfreundlichkeit. Das Layout ist durch den Anwender konfigurierbar und die Einstellungen werden gespeichert. Wenn im Rahmen von Erweiterungen neue grafische Elemente hinzukommen, soll dies weiterhin gegeben sein.

3.2.3 Zusammenspiel mit anderen Anwendungen

Es werden zwei Fremdprogramme benutzt, „dot“ (GraphViz) für die Erzeugung des Aufrufgraphen und „objdump“ (BinUtils) bei der Darstellung von annotiertem Assemblercode.
Mit der Realisierung der interaktiven Steuerung kommt noch der Profiler hinzu. Genauer gesagt, findet diese Interaktion zwischen der Erweiterung und dem Profiling-Werkzeug statt.

3.2.4 Mehrbenutzerfähigkeit

KCachegrind ist nicht für die Verwendung durch mehrere Benutzer vorgesehen. Jeder Benutzer arbeitet mit seiner eigenen Instanz. Derzeit sind keinerlei kritische Zugriffe auf gemeinsam verwendete Daten denkbar, weil Daten nur betrachtet und nicht erzeugt werden. Ein gemeinsamer Zugriff ist nur möglich, wenn zwei Anwender auf das gleiche Tracefile zugreifen. Da es sich dabei aber um einen Lesezugriff handelt, besteht auch hier kein Problem.
Wird der Funktionsumfang von KCachegrind um interaktives Profiling erweitert, ändert sich die Situation. Führen zwei Anwender zur gleichen Zeit einen Profiling-Lauf mit dem selben Profiler auf dem selben System aus, so können Komplikationen auftreten, falls die Ausgabedateien in das selbe Verzeichnis gespeichert werden. Ob und in wie weit dieses Problem auftreten kann, hängt allerdings vom jeweiligen Profiler ab.

3.2.5 Sicherheitsanforderungen

Es gibt keinerlei Sicherheitsanforderungen im Bezug auf Authentifizerung gegenüber dem Programm. Jeder, der das Programm ausführen kann, ist auch dazu berechtigt.
Als Visualisierungswerkzeug ist KCachegrind selbst nicht dazu in der Lage Daten zu manipulieren oder zu zerstören. Problematisch ist allerdings die Verwendung der beiden bereits angesprochenen Fremdprogramme. Es ist theoretisch denkbar, Sicherheitslöcher in diesen Programmen auszunutzen und so Fremdcode zur Ausführung zu bringen. Auch eine Art „Denial of Service“-Angriff ist denkbar, sei es durch das Einlesen einer riesigen Datei oder der Erzeugung eines sehr komplexen Aufrufgraphen. Dieses Thema ist so komplex, daß hier auf eine detaillierte Analyse verzichtet werden muß.
Generell bestehen Sicherheitsrisiken. Es kann aber davon ausgegangen werden, daß der Zugriff auf die Maschinen, wo KCachegrind läuft, begrenzt ist.

3.2.6 Portierbarkeit, Standards

KCachegrind ist momentan ausschließlich auf Linux lauffähig, beziehungsweise sollte sich über cygwin[14] auch auf Windows ausführen lassen. Um die bestehende Portabilität beizubehalten, ist eine strikte Einhaltung der folgenden Standards, mit denen KCachegrind entwickelt wurde, notwendig:

- C++
- QT3
- KDE3

3.3 UML

UML[2] wird im folgenden als Modellbeschreibungssprache eingesetzt. Die groben Anforderungen werden mit Hilfe von UML „Use Case“-Diagrammen genauer erläutert. Dabei werden alle dem Anwender des Programmes sichtbaren Anwendungsfälle diskutiert. UML-Klassendiagramme werden für die Beschreibung des Systems eingesetzt. Szenarien sind beispielhafte Abläufe, die eine sinnvolle Nutzung der Software aus Sicht des Anwenders wiederspiegeln. Detaillierte Beschreibungen zu UML sind in [7], [8], [9] und [10] zu finden.

[2]Unified Modeling Language

3.4 Szenarien

Ein Optimierungsprozeß läßt sich grob in folgende Einzelschritte unterteilen:

1. Profiling der Anwendung
2. Analyse der gewonnenen Daten
3. Verbesserung des Programmes
4. Profiling des verbesserten Codes
5. Überprüfen der Performanzsteigerung

Momentan wird der Programmierer von KCachegrind nur beim zweiten Schritt unterstützt. Im fünften Schritt müssen die Ergebnisse der beiden Profiling-Läufe manuell verglichen werden. Wie der Optimierungsvorgang in Zukunft aussehen soll, ist in Abb. 3.1 zu sehen.

Der Erste Schritt des Optimierungsprozesses besteht darin, eine Performanzanalyse der Software durchzuführen. Dies kann der Anwender über KCachegrind steuern. Dazu wählt er ein Profiling-Werkzeug aus, das auf seinem System installiert ist und von KCachegrind unterstützt wird. Anschließend muß er die notwendigen Konfigurationen, wie beispielsweise Pfade und Startoptionen, für das jeweilige Werkzeug und das zu profilende Programm, vornehmen.

Das Profiling läßt sich über die grafische Oberfläche starten. Die Aktionen, die dem Anwender währenddessen zur Verfügung stehen, hängen vom verwendeten Profiling-Werkzeug ab. Denkbar ist einerseits, daß man den aktuellen Zustand des Profiling sieht, also die Daten die bisher gemessen wurden sowie die gerade ausgeführte Funktion. Unterstützt das Profiling-Werkzeug dies nicht, so kann über die grafische Oberfläche ein Zwischenstand angefordert werden. Für den Fall, daß das Profiling wegen einer komplexen Programmanalyse sehr lange dauert, bieten einige Profiling-Werkzeuge die Möglichkeit eine Datei mit einem Zwischenresultat zu erzeugen.

Ist der Profilingprozess abgeschlossen, so wird die erzeugte Datei in KCachegrind geladen. Aus den gemessenen Daten erstellt Kcachegrind verschiedene Graphen und Tabellen, anhand derer der Anwender jene Codesegmente identifizieren kann, die unverhältnismäßig viel Rechenzeit verbrauchen. Aufgrund der Ereignistypen lassen sich Rückschlüsse auf die Ursachen des Performanzverlustes ziehen. So kann der Anwender recht schnell Schwachstellen und die besten Ansatzpunkte für eine Optimierung identifizieren.

Wurden die Änderungen vorgenommen, so führt der Anwender ein erneutes Profiling durch. Nun können das neue, als auch das alte Tracefile geöffnet und miteinander vergli-

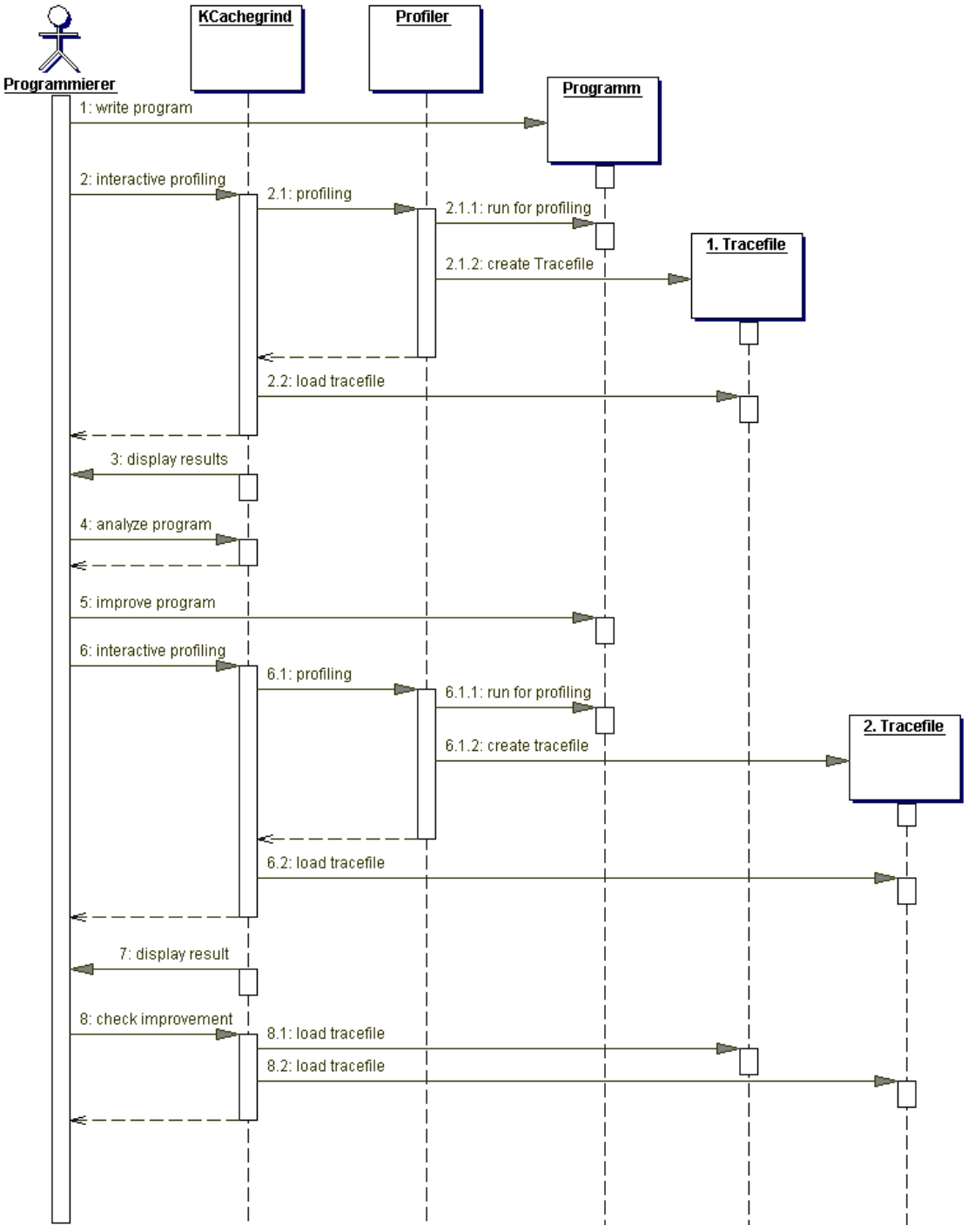

Abbildung 3.1: Ablauf eines Optimierungsvorgangs

chen werden. Jeder Funktion werden dabei die Kosten aus beiden Meßergebnissen zugeordnet, Änderungen sind somit sofort ersichtlich.

3.5 Use Cases

3.5.1 Aktoren

3.5.1.1 Actor: Programmierer

Einer der Software-Entwickler ist für die Optimierung der Performanz des Programmes zuständig. Seine Aufgabe ist es, mittels eines Profiling-Tools eine Performanzanalyse der Software durchzuführen, Optimierungsmöglichkeiten zu finden, Änderungen am Code vorzunehmen und eine erneute Analyse, zur Überprüfung der Performanzsteigerung, durchzuführen.

3.5.1.2 Actor: Profiling-Werkzeug

Das Profiling-Werkzeug zeichnet das Laufzeit-Verhalten eines Programmes auf und speichert die Daten in einem Tracefile. Welche Art von Daten das Werkzeug liefert wurde bereits in Kapitel 2.2 erläutert.

3.5.2 System: KCachegrind

KCachegrind versucht, die vom Profiling-Tool gesammelten Daten dem Anwender in einer übersichtlichen und klaren Form zu präsentieren. Dies beschleunigt das Identifizieren der Codestellen mit dem größten Optimierungspotential.

Die „Use Cases“ für KCachegrind sind in Abbildung 3.2 in einem UML „Use Case“-Diagramm im Überblick dargestellt.

Der Programmierer kann mit KCachegrind prinzipiell zwei verschiedene Tätigkeiten ausführen. Einerseits ist der Profiling-Vorgang steuerbar und kann „live“ mitverfolgt werden, das heißt, daß nicht erst das endgültige Ergebnis des Proflings dargestellt wird, sondern während des Profiling-Vorgangs mehrere Zwischenstände angezeigt werden.
Andererseits werden die Ergebnisse des Profilings nach wie vor grafisch dargestellt und so wird der Anwender bei seiner Analyse unterstützt. Hierbei sind zwei Fälle zu unterscheiden: Das Analysieren der Ergebnisse eines Profiling-Laufes und das Vergleichen zweier Ergebnisse. Der erste Fall tritt ein, wenn der Programmierer zum ersten Mal eine Performanzanalyse für sein Programm durchführt. Der zweite Fall entsteht, wenn der Anwender

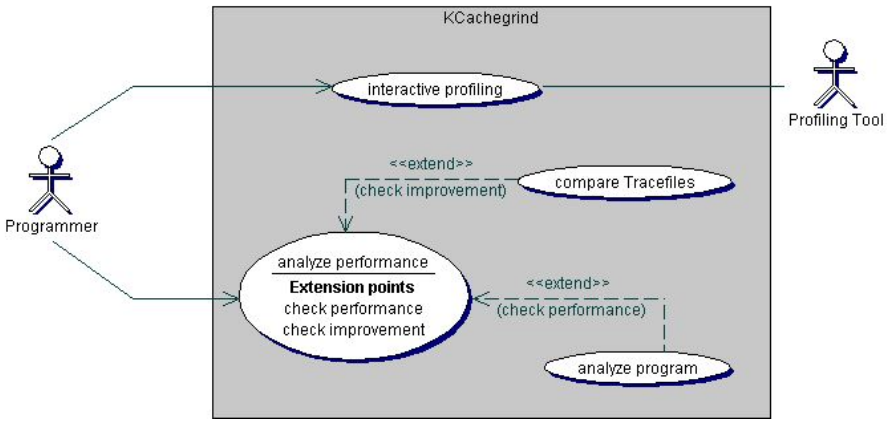

Abbildung 3.2: Use Case Diagramm

Änderungen an seinem Programm durchgeführt hat und nun die erhoffte Performanzsteigerung bestätigt haben möchte.

3.5.2.1 Use Case: Interactive Profiling

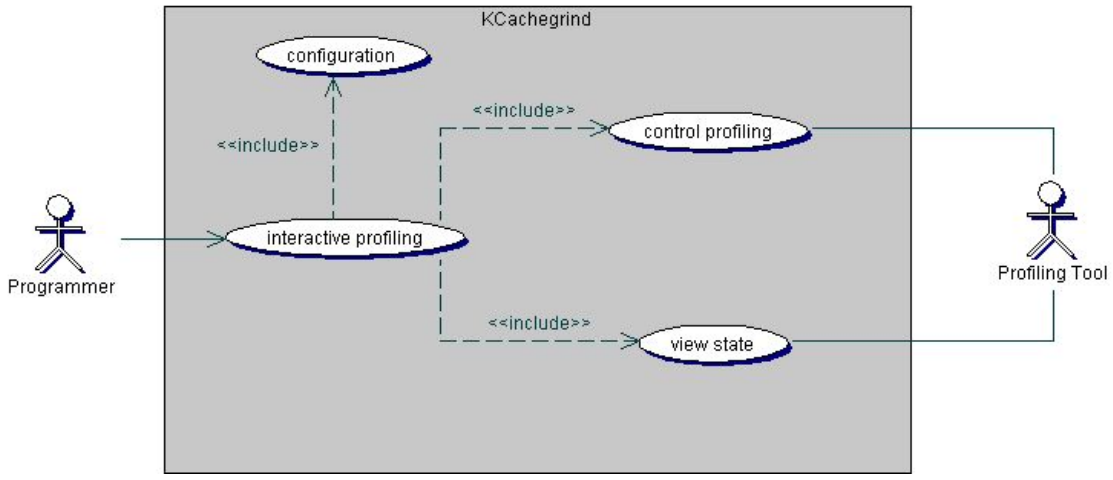

Abbildung 3.3: Use Case: Interactive Profiling

Dieser Use Case ermöglicht die Beschleunigung und Vereinfachung des Profile-Prozesses, indem aus KCachegrind heraus ein Profiling-Tool angesteuert werden kann.
Zunächst müssen einige Einstellungen vorgenommen werden, zum Beispiel welches Programm analysiert und welche Optionen des Profilers genutzt werden sollen (**Use Case:**

Configure). Ist dies geschehen, so kann das Profiling gestartet (**Use Case: Start Profiling**) beziehungsweise notfalls abgebrochen werden (**Use Case: Stop Profiling**).
Dabei kann während des Profiling-Vorgangs der Fortschritt beobachtet werden (**Use Case: View State**), es wird ein Zwischenergebnis der Messungen erzeugt und automatisch geladen. Die aktuelle Position (Funktion) die gerade ausgeführt wird, ist dabei hervorgehoben.

3.5.2.2 Use Case: Analyze Performance

KCachegrind unterstützt den Programmierer bei der Analyse seiner Meßergebnisse, indem die vom Profiler erzeugten Daten grafisch aufbereitet werden (**Use Case: Analyze Performance**). Hierbei kann man zwei verschiedene Situationen differieren:

Use Case: Analyze Program

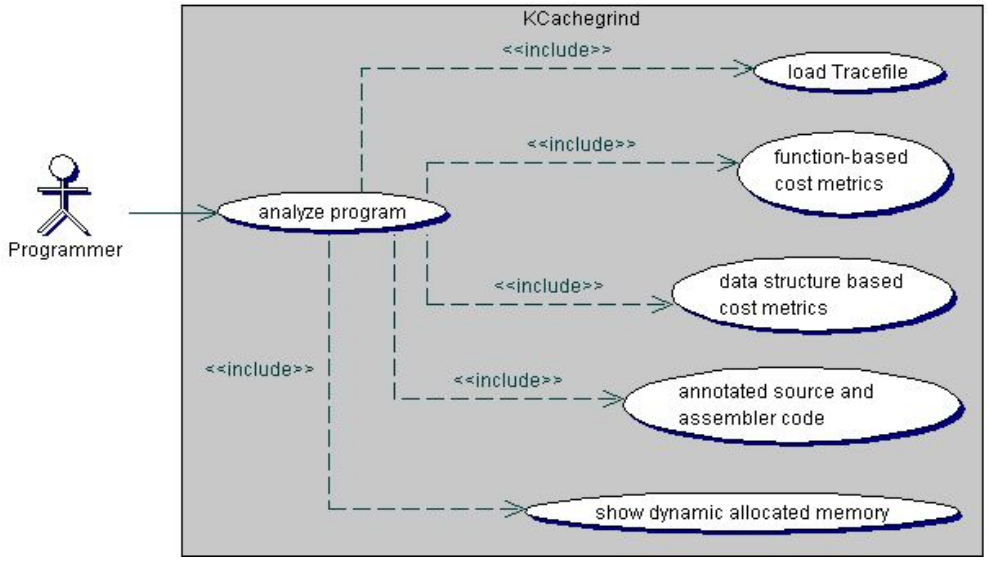

Abbildung 3.4: Use Case: Analyze Program

Der Programmierer möchte sein Programm optimieren. Dazu erzeugt er mittels eines Profiling-Tools ein Tracefile von seinem Programm und lädt dieses in KCachegrind.
Der Use Case **Analyze Program** bezieht sich auf die Auswertung eines Tracefiles. Zunächst wird das Tracefile geladen (**Use Case: load Tracefile**) anhand dessen dann verschiedene Darstellungen erzeugt werden:

- Eine Liste der aufgerufenen Funktionen mit inklusiven und exklusiven Kosten, ein Callgraph sowie eine Call-TreeMap (**Use case: Function based Cost Metrics**).

- Zusätzlich können annotierter Source- und Assemblercode inklusive Sprünge angezeigt werden (**Use Case: Annotated Source and Assembler Code**).
- Es kann eine Liste der Datenstrukturen mit zugehörigen Kosten erzeugt werden (**Use Case: Data Structure based Cost Metrics**).

Dabei wird dynamisch allokierter Speicher berücksichtigt und fließt korrekt in die Speicherberechnung mit ein (**Use Case: Dynamic allocated Memory**).

Use Case: Compare Tracefiles

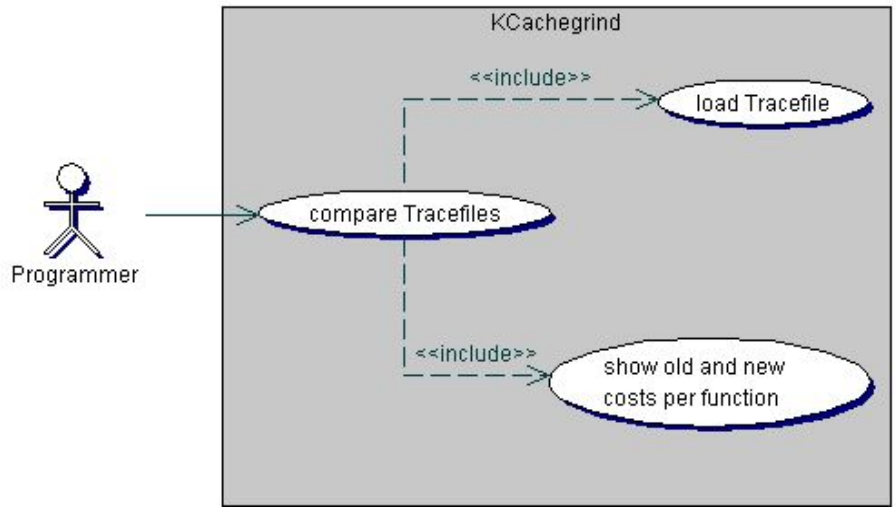

Abbildung 3.5: Use Case: Compare Tracefiles

Angenommen, ein Programmierer hat die Effizienz seines Programmes mittels eines Profiling Tools und KCachegrind analysiert und die Schwachstellen optimiert, so möchte er nun die verbesserte Performance überprüfen.
Hierzu bietet sich der Vergleich eines alten Tracefiles mit einem neuen an (**Use Case: Compare Tracefiles**). Dazu werden beide Tracefiles geladen (**Load Tracefile**) und eine Visualisierung erzeugt, in der sowohl die alten als auch die aktuellen Kosten der jeweiligen Funktion angezeigt werden (**Use Case: Show old and new Costs per Function**).

3.6 Architektur

KCachegrind umfasst derzeit über 100 Klassen. Eine komplette Beschreibung wäre nicht nur sehr groß und unübersichtlich, sondern ist auch für die vorliegende Arbeit nicht notwenig. Eine detaillierte Beschreibung ist normalerweise Aufgabe der Entwicklerdokumen-

tation, die im Falle von KCachegrind noch nicht zur Verfügung steht. Deshalb wird die interne Architektur im folgenden schematisch vorgestellt und nur wichtige Punkte beziehungsweise Schwachstellen detailliert herausgearbeitet.
Prinzipiell lassen sich die Klassen in drei funktionale Komponenten gliedern:
Datenimport, Datenmodell und Visualisierung.

3.6.1 Datenimport

Die Klasse LOADER ist die Basisklasse für den Datenimport und definiert die grundlegenden Funktionen für den Import von Daten. Wie in Abbildung 3.6 zu sehen, ist CACHEGRINDLOADER von ihr abgeleitet. Sie ist eine Importklasse für die Profile-Daten des Programmes Calltree, dessen Datenformat auf dem Format von Cachegrind basiert.

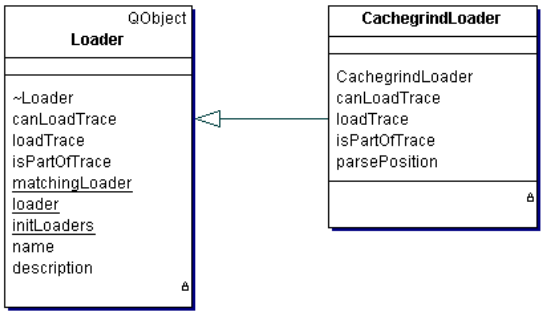

Abbildung 3.6: Datenimport

3.6.2 Datenmodell

Das von KCachegrind verwendete Datenmodell ist in Abb. 3.9 im Überblick zu sehen. Es spiegelt die Informationshierarchie der vom Profiler gelieferten Daten wieder. Dabei ist zu beachten, daß während eines Profiling-Laufes mehrere Dateien mit jeweils Teilinformationen entstehen können. Die Profiling-Daten können also auf mehrere Dateien verteilt sein, die wir im folgenden als Tracefiles bezeichnen.
Die „Trace“-Klassen enthalten jeweils die Informationen des gesamten Profiling-Laufes. Für jede Trace-Klasse gibt es auch eine „Tracepart“-Klasse, welche die Informationen eines bestimmten Files enthält. Abbildung 3.7 zeigt den Zusammenhang am Beispiel der Klassen TRACEFUNCTION und TRACEPARTFUNCTION.

Die Vererbungshierarchie der TraceCost-Klassen ist in Abbildung 3.8 zu sehen. Basisklasse ist TRACECOST. Sie enthält die grundlegenden Metriken für die kleinste und einfachste Einheit in einem Tracefile, beispielsweise Lese-, Schreibzugriffe und Cache-Misses. Alle anderen TraceCost-Klassen sind von ihr abgeleitet und um spezielle Metriken erweitert.

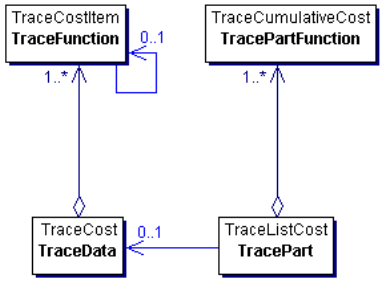

Abbildung 3.7: TracePart

Die „List“-Klassen bieten eine Basisfunktionalität für das automatische Aufaddieren von Kosten aus einer Liste von Kosten von Untereinheiten. „Cumulative“-Klassen stellen die Möglichkeit zur Verfügung Exklusiv- und Inklusivkosten zu speichern.
TraceCallCost erweitert die TraceCost-Klasse um Metriken für Aufrufe. Die Klasse TraceCostItem ist die Basisklasse für alle „interessanten“ Kosten wie TraceFunction, TraceClass, TraceFile und TraceObject.

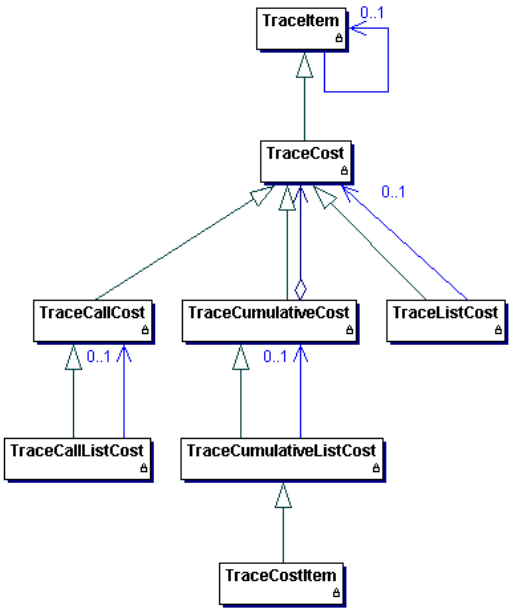

Abbildung 3.8: TraceCost-Klassen

Die Klasse TraceData bildet die oberste Ebene des Datenmodells. Sie ist von TraceCost abgeleitet und enthält die wichtigsten Informationen der eingelesenen Tracefiles eines Profiling-Laufes. Für jedes TraceData-Objekt existiert ein TraceCostMapping-Objekt. Dieses stellt die Beziehung zwischen den Werten und den zugehörigen Kostentypen her.
Die Meßdaten eines Profiling-Laufes können auf mehrere Tracefiles verteilt sein. Jede

Quelldatei wird durch ein TraceFile-Objekt repräsentiert. TraceObject enthält die Informationen eines bestimmten Programmabschnitts und den darin enthaltenen Funktionen. Die Klasse TraceClass repräsentiert eine C++-Klasse und besteht ihrerseits, wie auch TraceFile und TraceOject, aus beliebig vielen TraceFunction-Objekten.
TraceFuncionSource ist eine Hilfsklasse für TraceFunction und beschreibt in welchem TraceFile-Objekt sich eine Funktion befindet. Ein TraceFunction-Objekt enthält die Zuordnung aller TraceLines zu einem TraceFile-Objekt.
Ein TraceLine-Objekt enthält die Daten einer Zeile, welche mehrere Instruktionen beinhalten kann. TraceInstr repräsentiert die unterste Ebene des Datenmodells.
Die Klasse TraceCall stellt den Aufruf einer Funktion aus einer anderen heraus dar und lässt sich in TraceLineCall- und TraceInstrCall-Objekte unterteilen.

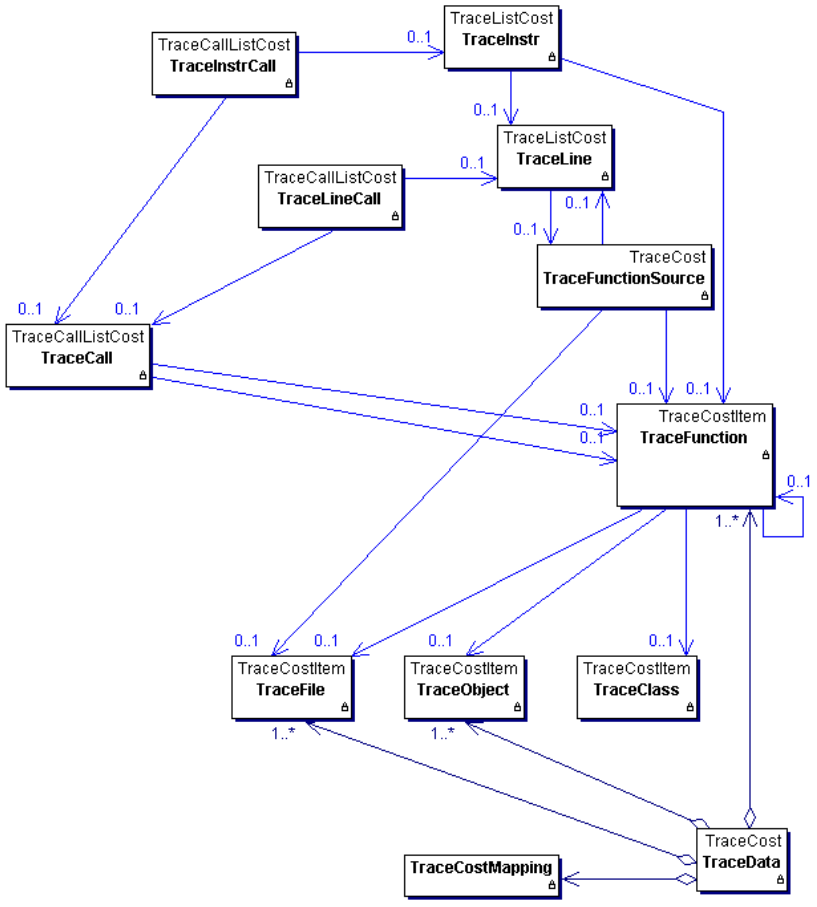

Abbildung 3.9: Datenmodell

3.6.3 Visualisierung

Die Visualisierung ist für die grafische Aufbereitung der Profiling-Daten zuständig. Sie stellt das Bindeglied zwischen dem Daten-Import und der grafischen Anzeige dar und bestimmt, welche Informationen wo und wie angezeigt werden.
Der Aufbau der grafischen Oberfläche ist in Abbildung 3.10 zu sehen.

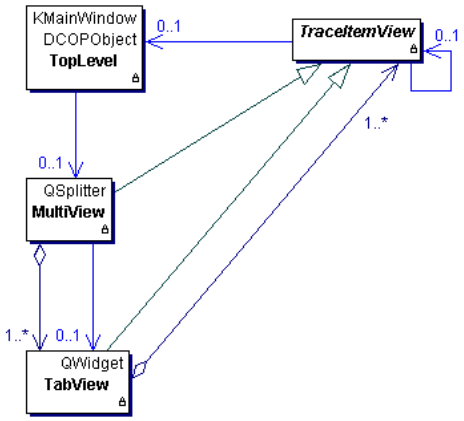

Abbildung 3.10: Fenster der grafischen Oberfläche

Widgets sind die kleinste Einheit, mit der ein GUI-Toolkit[3] umgehen kann, und üblicherweise bilden sie eine Hierarchie, da Behälter-Widgets andere Widgets enthalten können.
Die Klasse TopLevel repräsentiert das oberste Element in dieser Hierarchie von KCachegrind und stellt Methoden zur Verwaltung der graphischen Oberfläche zur Verfügung. Die Methode `newWindow` beispielsweise, erzeugt ein neues Toplevel-Fenster und `loadConfig/saveConfig` lädt, beziehungsweise speichert KDE-Konfigurationseinträge.
Ein Toplevel-Widget enthält Verweise auf die Unter-Widgets, wie zum Beispiel ein MultiView-Objekt. Die Klasse MultiView kann eines oder mehrere TabView-Objekte verwalten, sorgt beispielsweise für die Anordnung der einzelnen TabViews.
TraceItemView ist eine abstrakte Basisklasse, welche die gemeinsame Funktionalität aller KCachegrind-Visualisierungen für ein aktives TraceItem definiert , beispielsweise eine Funktion. Wichtig ist, daß zwischen aktiven und selektierten Elementen unterschieden werden muß. Bei allen GUI-Toolkits einheitlich, kann immer nur ein Element aktiv, aber meherere Elemente selektiert sein.
Ändert sich das aktive Element in einem TabView-Objekt, so hat dies Auswirkungen auf den Inhalt der anderen TabViews.

Die von TraceItemView abgeleiteten Klassen sind spezifische Visualisierungen und in Abbildung 3.11 im Überblick dargestellt.

[3]Bibliothek mit Methoden zur Erzeugung grafishcer Oberflächen, QT stellt zum Beispiel ein solches Toolkit dar

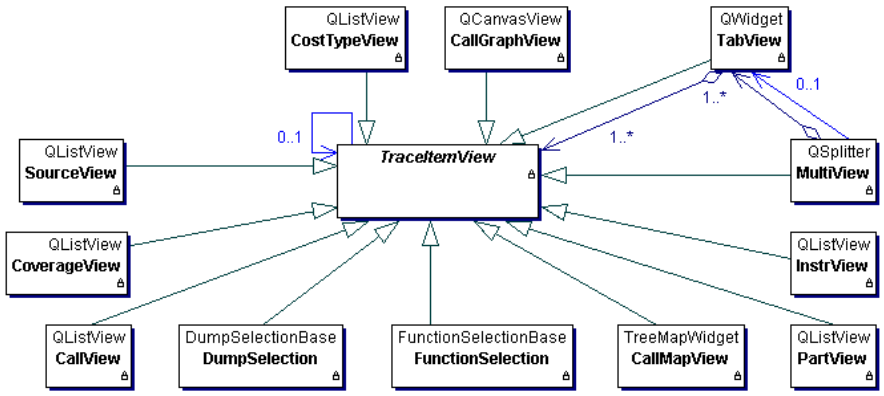

Abbildung 3.11: Visualisierungs-Klassen

Nachfolgend wird ein kurzer schematischer Überblick auf die wichtigsten Visualisierungsformen von KCachegrind geworfen. Die Klasse SourceView ist für die Anzeige des annotierten Quellcodes zuständig. CallGraphView repräsentiert den Aufrufgraphen. Die einzelnen grafischen Subkomponenten dieser Visualsierung sind in Abbildung 3.13 zu sehen.

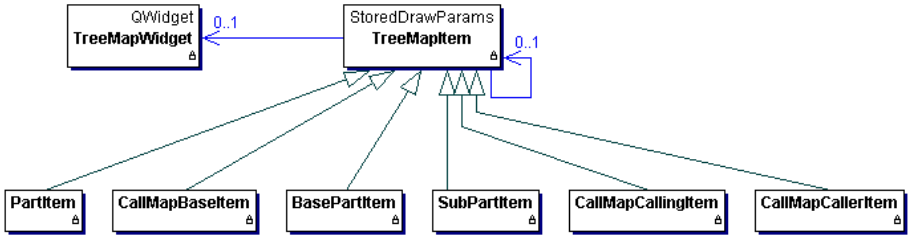

Abbildung 3.12: TreeMap-Klassen

Die an der „Treemap“-Visualisierung beteiligten Klassen zeigt Abbildung 3.12. TreeMap ist die Basisklasse der einzelnen Elemente der Treemap.

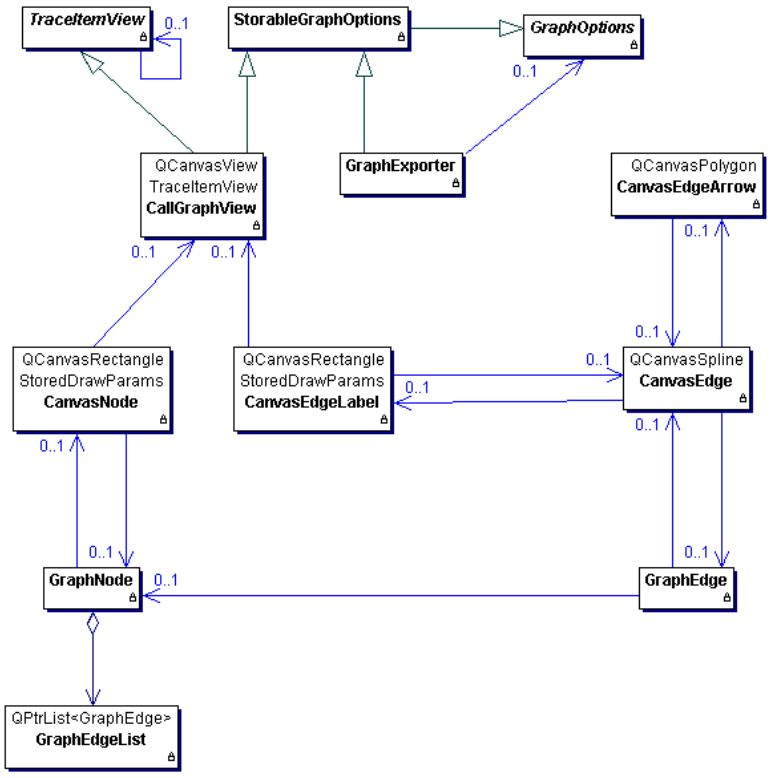

Abbildung 3.13: Callgraph-Klassen

Kapitel 4

Konzept

Wie schon in Kapitel 3.2.1 beschrieben, soll das Design von KCachegrind modifiziert werden. Das Programm soll einzelne Komponenten erweiterbar sein, wofür sich verschiedene Konzepte anbieten, die im folgenden diskutiert werden.

4.1 Grobarchitektur

Ein *Plugin* ist eine Code-Komponente, die einem Programm dynamisch hinzugefügt werden kann und eine neue oder erweiterte Funktionalität bietet. Sie bedient bestimmte Schnittstellen und wird vollständig dynamisch geladen. Ein Plugin ist nicht alleine lauffähig, kann aber einzeln verbreitet werden.

Für die Realisierung eines Plugins bieten sich unter Linux „Dynamically loaded libraries“, *dynamische Programmbibliotheken*, an. Diese werden nicht beim Start des Programmes, sondern erst bei Bedarf geladen, wodurch die „Startup“-Zeit[1] einer Anwendung gering bleibt. Über den Befehl `dlopen(3)` lassen sich dynamische Bibliotheken laden und durch den Befehl `dlclose()` wieder schließen.

Ein Programmierer kennt nicht alle Datenformate der verfügbaren Profiler, was angesichts der großen Zahl verschiedener Profiling-Werkzeuge auch nur schwer vorstellbar ist. So muß er sich dank eines Plugins nicht mit dem kompletten Programm auskennen, sondern entwickelt ein solches nur für eine gegebene Schnittstelle. Dies bedeutet einen wesentlich geringeren Einarbeitungsaufwand und vereinfacht die Entwicklung von Komponenten durch Dritte. Außerdem werden Plugins zur Laufzeit bei Bedarf geladen und wirken sich somit nicht negativ auf die Startzeit der Anwendung aus. Ein weiterer Vorteil ist, daß nach Auslieferung einer Software Anpassungen und Erweiterungen relativ einfach möglich

[1] die Zeit, die benötigt wird um ein Programm zu starten

sind. Beispielsweise muß wegen eines neuen Import-Filters nicht gleich eine neue Version veröffentlicht werden.
Ein solches Plugin-System ist in Abbildung 4.1 schematisch dargestellt. Das Hauptprogramm erhält Plugins durch den PlugInLoader. Dieser führt die Suche nach Plugins in Form einer Dateisystemsuche durch und instanziiert passende, gefundene Plugins. Die Kommunikation zwischen dem „Client“ und den Plugins läuft ausschließlich über die Schnittstelle.
Der PlugInLoader sucht dynamisch nach konkreten Plugins und liefert geladene Plugins zurück. Ein konkretes Plugin (ConcretePlugIn) implementiert die Plugin-Schnittstelle und kann auch noch von anderen Klassen abgeleitet sein, also mehrere Schnittstellen implementieren. Im Falle von KCachegrind wird es 3 Schnittstellen geben, eine für den Datenimport, eine für Visualisierungserweiterungen und eine für die interaktive Steuerung. Die Klasse PlugIn stellt die Schnittstellendefinition dar und legt die ausführbaren Funktionen und Rückgabewerte fest. Plugins werden in [23] ausführlich behandelt.

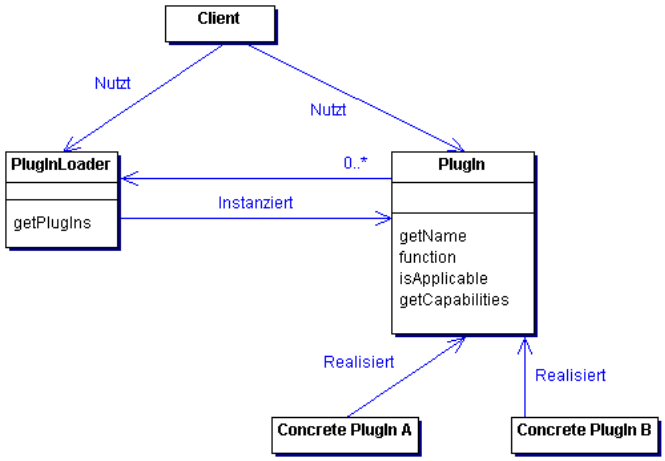

Abbildung 4.1: PlugInSystem

4.2 Datenmodell

Das von KCachegrind verwendete Datenmodell war bisher statisch, also fest implementiert. Aufgrund der neuen Anforderungen sollen nun auch Kosten für Datenstrukturen verwaltet werden können. Dafür bieten sich 3 Möglichkeiten an:

Statisches Modell:
Die naheliegenste Lösung ist es, wie Abbildung 4.2 illustriert, das statische Datenmodell um die notwendigen Klassen zu erweitern.

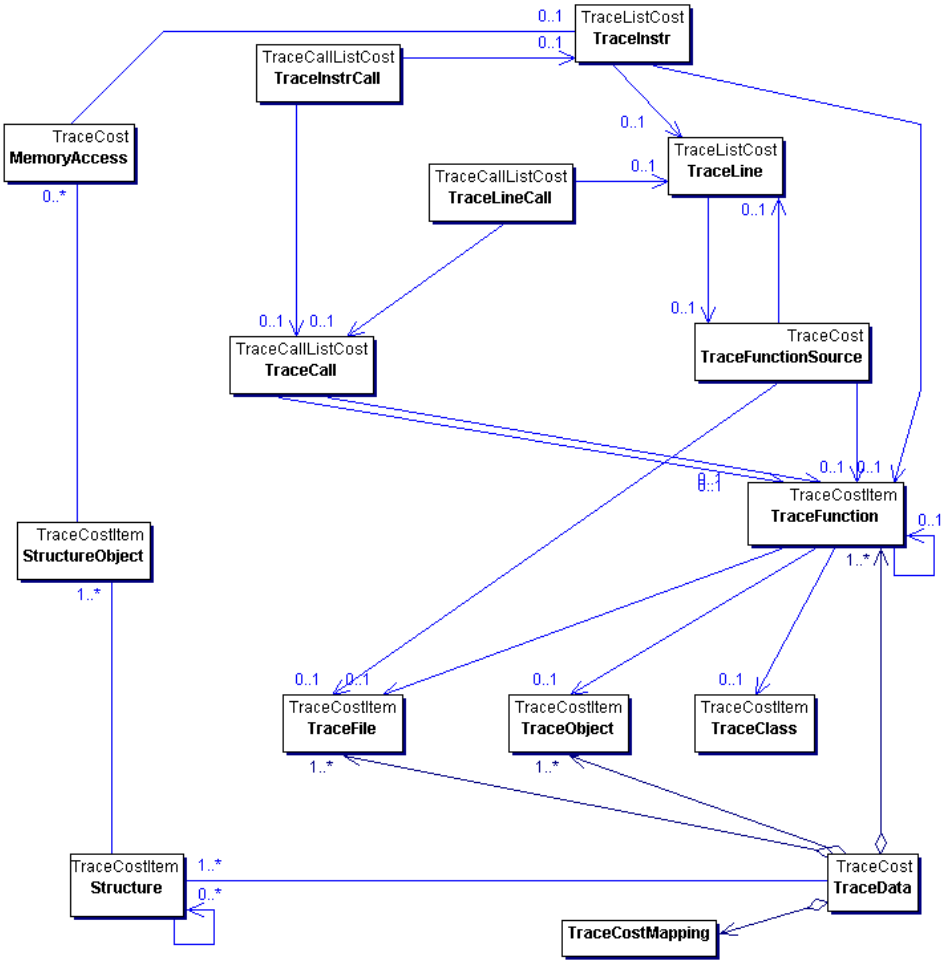

Abbildung 4.2: Datenmodell

Die Klasse STRUCTURE verwaltet die Profile-Daten aller Datenstruktur-Objekte der geladenen Dateien. STRUCTUREOBJECT repräsentiert die Informationen für ein bestimmtes Datenstruktur-Objekt. Der einzelne Speicherzugriff auf eine Datenstruktur wird durch ein Objekt der Klasse MEMORYACCESS symbolisiert. Vorteil dieser Methode ist der geringe Aufwand der Umsetzung. Das Datenmodell bleibt allerdings statisch und damit unflexibel gegenüber Änderungen/Erweiterungen, die in Zukunft notwendig sein könnten.
Die verschiedenen Profiler, liefern unterschiedliche Arten von Informationen und unterscheiden sich folglich auch in den verwendeten Datenmodellen. Als unabhängiges Visualisierungswerkzeug, versucht KCachegrind möglichst alle Profiler prinzipiell zu unterstützen. Große Differenzen zwischen den Datenmodellen der Profiler sind zwar unwahrscheinlich aber theoretisch möglich. Es ist also denkbar, daß ein Datenmodell nicht komplett von KCachegrind abgedeckt wird, da ein statisches Modell eine gewisse Inflexibilität mit sich bringt. Dies ist beispielsweise dann der Fall, wenn ein Profiler bereits datenstrukturbezogene Profiling-Daten liefern kann. Die Profiling-Daten könnten zwar

nach wie vor Importiert werden, allerdings würde ein Teil der Informationen verloren gehen.

Dynamisches Datenmodell:
Für jeden Profiler, dessen Daten in KCachegrind importiert werden sollen, ist es notwendig, ein eigenes Plugin zu schreiben. Dies muß zum Einlesen der Daten das Datenformat kennen. Das dynamische Datenmodell wird erst zur Laufzeit vom Plugin erzeugt. Dadurch wird eine Kompatibilität zu allen Profiling-Werkzeugen erreicht. Sollte sich das Datenmodell eines Profilers ändern, so muß nur das jeweilige Plugin bearbeitet werden. Die Datenmodelle der anderen Plugins bleiben dadurch unbeeinflußt. Ein dynamisches Datenmodell bietet also eine sehr hohe Flexibilität im Bezug auf Änderungen/Erweiterungen und schafft Unabhängigkeit zwischen den Datenmodellen der verschiedenen Profiler. Dafür ist aber der Implementierungsaufwand höher.
Eine *Factory* stellt Methoden bereit, um Objekte zu erzeugen. Der Sinn einer Factory besteht darin, Objekte auf einem sehr abstrakten Niveau zu erstellen und den Typ der erstellten Objekte zur Laufzeit eventuell auswechseln zu können.
Kernstück des dynamischen Datenmodells ist die Klasse DatamodelFactory, die die Methoden zur Erzeugung des Datenmodells zur Verfügung stellt. Das Plugin hat Zugriff auf diese Funktionen und erzeugt bei Bedarf das notwendige Datenmodell. Sollte sich mit der Version eines Profilers auch das Datenformat beziehungsweise das Datenmodell ändern, so kann das Plugin beide Datenmodelle zur Verfügung stellen und je nach verwendeter Version laden.

Semi-dynamisches Datenmodell:
Eine weitere Möglichkeit stellt ein Kompromiß zwischen statischem und dynamischen Datenmodell dar. Die bestehenden Klassen bleiben nach wie vor fest implementiert, das Datenmodell kann aber durch die Funktionen, die dem Plugin zur Verfügung stehen, um einzelne Klassen erweitert werden.
Dieses Semi-dynamische Datenmodell bietet eine gewisse Flexibilität und der bestehende Code des Datenmodells kann wiederverwendet werden, d.h. der Implementierungsaufwand fällt etwas geringer aus als der des dynamischen Datenmodells.

XML-Lösung:
Ein ganz anderer Ansatz wäre gewesen, das Datenmodell nicht in C++ zu beschreiben, sondern durch eine XML-Datei. Diese befindet sich im selben Verzeichnis wie das Plugin und wird bei Bedarf geladen. Für das Einlesen der XML-Datei benötigt man einen Parser. Dieser kann entweder selbst implementiert oder aber ein Bestehender eingebunden werden. Ersteres stellt einen erheblichen Programmieraufwand dar. Die Verwendung eines externen Parsers bedeutet, daß zur Verwendung von KCachegrind eine zusätzliche Software installiert werden muß. Somit entsteht ein Mehraufwand für den Anwender bei der Installation und eine Abhängigkeit vom Parser, was Plattformunabhängigkeit anbelangt.

Ferner müsste eine DTD[2] festgelegt werden, die die Regeln beinhaltet, wie eine solche XML-Datei aufgebaut sein darf.

4.3 Datenimport

Die Aufgabe des Datenimport-Plugins besteht darin, die Profiling-Daten eines Profilers einzulesen und KCachegrind zur Verfügung zu stellen. Die eingelesenen Imformationen werden nicht komplett über die Schnittstelle transferiert, sondern können gezielt angefordert werden, zum Beispiel alle Funktionen eines Tracefiles. Dadurch wird sichergestellt, daß die Verzögerung beim Laden sehr großer Dateien verringert werden kann. Das Import-Plugin liest zuerst nur die wichtigen Daten ein, also die oberen Elemente des Datenmodells. Die darunterliegenden Elemente können bei Bedarf, also wenn eine konkrete Anforderung über die Schnittstelle gesendet wird, nachgeladen werden.
Dabei kann davon ausgegangen werden, daß die Datei lokal zur Verfügung steht, auch wenn das Profiling auf einer anderen Maschine ausgeführt wird. Mit dem Thema *Remote Connection*[3] wird sich Kapitel 4.4 näher beschäftigen.
Eine weitere bedeutende Funktionatlität des Plugins ist es, überprüfen zu können ob das Format eines Tracefiles eingelesen werden kann, also ob es dem Format des vom Plugin unterstützten Profilers entspricht. Dies ist notwendig, weil nach der Anweisung des Benutzers eine bestimmte Datei zu öffnen, nicht klar ist, welchem Format die Datei entspricht. Es wird also ein Plugin nach dem anderen initialisiert und abgefragt, ob das Format der Datei unterstützt wird. Sollten mehrere Plugin das Datenformat verstehen können, so kann der Anwender auswählen.

4.4 Profiling-Steuerung

Der Profiler, den der Anwender benutzen möchte, befindet sich nicht zwangsweise auf der selben Maschine, auf der KCachegrind installiert ist. Um die Einsatzmöglichkeiten von KCachegrind nicht deutlich zu beschränken, sollten also „Remote Connections“ unterstützt werden. Alle Profiler können über eine Shell gesteuert werden, es ist also ausreichend, wenn das Plugin eine Möglichkeit hat Shellbefehle auf der Remote-Maschine“ abzusetzen.
Der Aufbau der Verbindung und die Kommunikation mit der Remote-Maschine könnte dem Plugin überlassen werden. Dies würde aber einen deutlichen Mehraufwand für die

[2] DocTypeDefinition

[3] Als Remote Connection bezeichnet man die Verbindung mit einem anderen Rechner, beispielsweise über ein Netzwerk

Programmierung eines Plugins bedeuten und zwangsweise Redundanz erzeugen. Am sinnvollsten ist wohl, daß der Verbindungsaufbau von KCachegrind übernommen wird und das Plugin über die Schnittstelle Befehle absetzen kann, die dann auf der Remote-Maschine zur Ausführung kommen. Wie die Verbindung im Endeffekt realisiert wird, bleibt damit für das Plugin transparent.

Die Schaltflächenelemente die zur Steuerung durch den Benutzer notwendig sind, können entweder vom Hauptprogramm zur Verfügung gestellt werden, oder aber durch das Plugin definiert werden. Ersteres würde dann eine allgemeine Funktionalität bieten, wie *Starten* beziehungsweise *Stoppen* eines Profiling-Laufes und *Konfiguration* des Profilers, worüber sich die verschiedenen Profiling-Optionen, die ein Profiler bietet, einstellen lassen. Letzteres würde es ermöglichen Profiler-spezifische Aktionen zur Verfügung zu stellen, wie etwa die Ausgabe eines Zwischenstands[4].
Da die Optionen Profiler-spezifisch sind, muß dieses grafische Element vom Plugin zur Verfügung gestellt werden.

Das interaktive Steuern von Profiling-Läufen beinhaltet auch, daß ein Zwischenresultat angezeigt und die aktuell ausführende Funktion markiert wird, wie etwa durch eine farbige Hervorhebung. Das Plugin muß über die Schnittstelle den Namen der Funktion übergeben können, die hervozuheben ist. Wie das Zwischenresultat erzeugt und die aktuelle Funktion ermittelt wird, ist Profiler-spezifisch und muß deshalb vom Plugin realisiert werden. Man kann dabei entweder auf Anweisung des Anwenders warten (durch klicken auf die Schaltfläche „erzeuge Zwischenresultat“), oder aber diesen Schritt immer in einem bestimmten Intervall ausführen lassen, welches über das Konfigurationsmenü eingestellt werden kann.

4.5 Visualisierungserweiterung

Ein Visualisierungsplugin bietet eine neue grafische Darstellungform für Profiling-Daten. Dabei können unter Umständen nur bestimmte Datentypen verarbeitet werden. Es muß also abgeglichen werden, welche Datentypen das Plugin visualisieren kann.
Eine Möglichkeit ist es, die Namen der Datentypen, die vom Plugin visualisiert und die vom Datenmodell zur Verfügung gestellt werden, zu vergleichen.
Die andere Möglichkeit besteht darin, daß man die denkbaren Daten-Typen in feste Gruppen einteilt. Diese Methode würde einen gewissen Abstand zwischen der Daten- und der Visualisierungsebene schaffen und so für eine größere Flexibilität zwischen Import- und Visualisierungsplugin sorgen. Sind die Datenmodelle zweier Import-Plugins an und für sich identisch, unterscheiden sich aber in den Bezeichnungen der einzelnen Datentypen, so kann nicht das selbe Visualisierungsplugin verwendet werden, sofern dieses nicht beide

[4]Manche Profiler bieten die Möglichkeit über einen Shellbefehl manuell die Erzeugung einer Profiling-Datei zu veranlassen, die die bisherigen Meßergebnisse beinhaltet

Namen bei den visualisierbaren Datentypen angibt. Eine kleine Änderung im Code eines der Plugins kann diese Inkompatibilität ohne größeren Aufwand beheben.
Eine dritte Möglichkeit ist, sowohl Namen als auch Gruppe eines Datentyps anzugeben und dies für den Abgleich zu verwenden. Dadurch entsteht ein geringfügig höherer Verwaltungsaufwand, dafür bleibt aber die hohe Kompatibilität der einzelnen Plugins erhalten. Ob sich der höhere Verwaltungsaufwand lohnt, hängt von der Zahl der Plugins ab die entwickelt werden. Im Falle von KCachegrind sollten einfache Namen ausreichend sein.

Da es um grafische Darstellung geht, muß die Schnittstelle eine Anbindung an die Benutzeroberfläche bieten. Die einzig sinnvolle Realisierung besteht darin, daß das Plugin ein Widget beschreibt, das in ein Widget der bestehenden Oberfläche integriert werden kann, beispielsweise eine „TabView“.

Zur Visualisierung des interaktiven Profilings, sollte die Schnittstelle eine Möglichkeit bereitstellen der Visualisierung mitzuteilen, welches die gerade aktive Funktion ist.

Es können mehrere Visualisierungen gleichzeitig aktiv sein. Diese arbeiten entweder unabhängig voneinander, oder reagieren auf gegenseitige Änderungen. Letzteres ist benutzerfreundlicher, denn wenn der Anwender das aktive Element in einer Visualisierung ändert, so ist es ideal wenn sich die Darstellung in den anderen Visualisierungen anpasst und nicht von Hand vorgenommen werden muß.

Die Daten, welche die Visualisierung für ihre Darstellung benötigt müssen vom Hauptprogramm angefordert werden. Die Antwort kann dabei nicht immer sofort erfolgen, beispielsweise wenn die Daten momentan gar nicht zur Verfügung stehen sondern erst über das Import-Plugin nachgeladen werden müssen. Die grafische Oberfläche darf dadurch aber nicht „einfrieren“, sprich sie muß weiterhin auf Benutzereingaben reagieren können. Dies läßt sich verhindern in dem man zwei Interfaces benutzt. Ein Interface beschreibt die Methoden die dem Plugin zur Verfügung stehen, wie etwa das Anfordern bestimmter Profiling-Daten. Das andere Interface wird vom Plugin implementiert und beinhaltet Methoden, über die Nachrichten und Antworten ans Plugin übergeben werden.

Kapitel 5

Realisierung

In diesem Kapitel wird ein konkretes Erweiterungs-Konzept mit den notwendigen Schnittstellen für KCachegrind vorgestellt.

5.1 Plugin-System

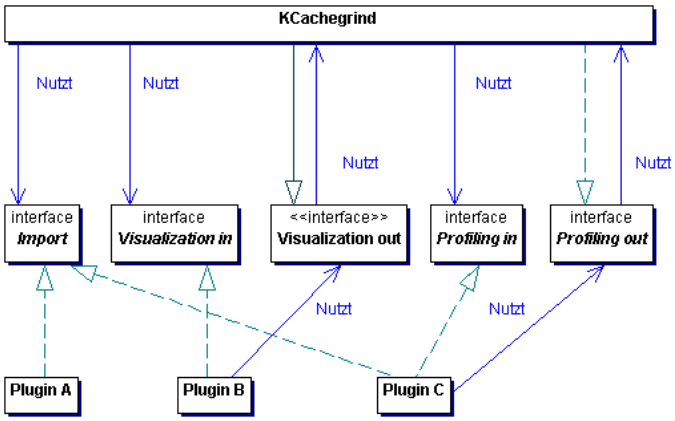

Abbildung 5.1: Plugin-Konzept

Abbildung 5.1 zeigt das Schema des Plugin-Konzepts für KCachegrind. Auf die Darstellung des PlugInLoaders wurde der Übersichtlichkeit halber verzichtet. Es gibt insgesamt fünf Schnittstellen, auch *Interface* genannt, die aber in drei Schnittstellenklassen eingeteilt werden können, weil zwei von ihnen bidirektional sind. Die Interfaces *Import*, Visualization in und Profiling in werden von KCachegrind genutzt und durch ein oder mehrerer Plugins realisiert. Die Interfaces Visualization out und Profiling out werden durch Klassen in KCachegrind realisiert und stellen den Plugins Methoden zur Verfügung.

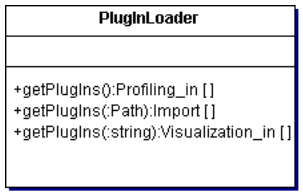

Abbildung 5.2: PlugInLoader

Die Klasse PlugInLoader, zu sehen in Abbildung 5.2 stellt die Methode `getPlugins` zur Verfügung, die von KCachegrind aufgerufen werden kann. Wird kein Parameter übergeben, so werden die Plugins zum interaktiven Profiling initialisiert und an KCachegrind zurückgegeben. Bei Übergabe eines Pfads zu einer Datei werden die Import-Plugin initialisiert und dasjenige an KCachegrind zurückgeliefert, welches das Dateiformat unterstützt. Ist der Übergabeparameter ein String so handelt es sich hierbei um den Namen eines Datentyp. Es wird überprüft, welches Visualisierungs-Plugin diesen Datentyp unterstützt und das entsprechende geladen.

5.2 Schnittstellen

Im folgenden werden die Schnittstellen und ihre Funktionsweise detailliert beschrieben.

5.2.1 Schnittstelle: Daten-Import

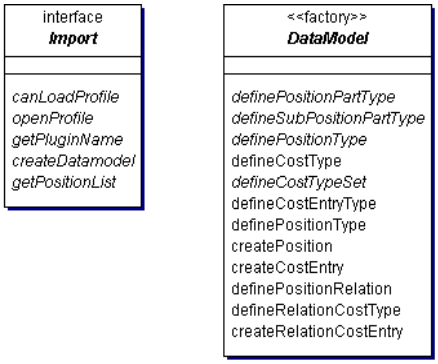

Abbildung 5.3: Interfaces für Import-Filter

In Abbildung 5.3 ist das Interface für den Datenimport, sowie die Factory-Klasse zur Erzeugung des Datenmodells dargestellt:

Die Methode `canLoadProfile` übergibt eine Datei um zu testen, ob das Plugin das Format unterstützt.

`openProfile` weist das Plugin an, eine Datei einzulesen.
`getPluginName` liefert den Namen des Plugins zurück, dieser ist notwendig falls dem Anwender eine Liste der zur Auswahl stehenden Plugins angezeigt werden muß.
`createDatamodel` weist das Plugin an, ein bestimmtes Datenmodell zu erzeugen, die Identifikation des Datenmodells erfolgt über einen String.
`getPositionsList` fordert explizite Informationen für einen bestimmten Datentyp an.

5.2.2 Schnittstelle: Interaktives Profiling

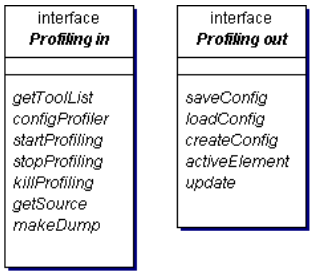

Abbildung 5.4: Interfaces für interaktives Profiling

Abbildung 5.4 zeigt die beiden Interface-Klassen für das interaktive Steuern von Profilern. Das *Profiling_in-Interface* wird vom Plugin realisiert und stellt folgende Funktionen zur Verfügung:

`configprofiler` legt fest, mit welchen Optionen der Profiler gestartet werden soll.
`startProfiling` setzt das Profiling in Gang.
`stopProfiling` weist das Plugin an, das Profiling kontrolliert abzubrechen.
`killProfiling` fordert die unmittelbare Unterbrechung des Profiling-Vorgangs.
`makeDump` fordert eine Zwischenausgabe an.
`getToolList` liefert die Namen der vom Plugin unterstützen Profiler beziehungsweise ihrer Versionen.
`getSource` liefert die Ausgabedatei.

KCachegrind realisiert das *Profiling_out*-Interface:

Die Funktion `activeElement` wird dazu benutzt, den Namen der gerade aktiven Funktion zu übergeben, damit diese in der Visualisierung hervorgehoben werden kann.
`saveConfig` und `loadConfig` dienen zum Laden und Speichern einer Profiler-spezifischen Konfiguration.
`createConfig` beschreibt, welche Konfigurationsoptionen der Profiler zur Verfügung stellt.
`update` informiert KCachegrind darüber, daß eine neue Zwischenausgabedatei vorliegt und geladen werden kann.

5.2.3 Schnittstelle: Visualisierung

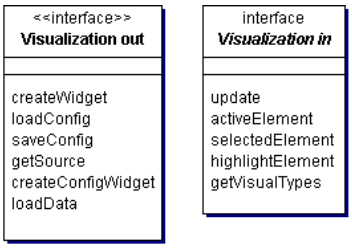

Abbildung 5.5: Interfaces für Visualisierungs-Plugins

Die Interface-Klassen des Visualisierungsplugins sind in Abbildung 5.5 zu sehen. Das *Visualization_out*-Interface wird von KCachgrind realisiert:

`createWidget` erzeugt das Visualisierungsfenster.
`saveConfig` und `loadConfig` Laden beziehungsweise Speichern die Visualisierungsspezifische Konfiguration.
`getSource` fordert von KCachegrind den Quellcode des profilierten Programmes an.
`cerateConfigWidget` übergibt das Widget zur Konfiguration der Visualisierung.
`loadData` liefert die Informationen für einen bestimmten Datentyp.

Das *Visualization_in*-Interface wird vom Pugin realisiert und definiert folgende Funktionen:

`update` teilt der Visualisierung mit, daß die angeforderten oder neue Profiling-Daten zur Verfügung stehen.
`activeElement` übergibt das momentan aktive Element.
`selectedElement` legt ein oder mehrere Elemente als selektiert fest.
`highlightElement` teilt der Visualisierung mit, welches Element besonders hervorzuheben ist.
`getVisualTypes` liefert die Liste der durch die Visualisierung unterstützten Datentypen zurück.

Kapitel 6

Fazit

6.1 Zusammenfassung

Die Anforderungen die an eine Software gestellt werden, könne sich im Laufe der Zeit ändern. Dann kann es dazu kommen daß eine Überarbeitung oder Erweiterung der Architektur notwendig wird, insbesondere wenn bei der ursprünglichen Entwicklung keine geeignete gewählt wurde. Die richtige Architektur einer Software ist von Anfang an sehr wichtig, weil ein späteres „Redesign“[1] der Architektur mit großem Aufwand verbunden ist. Auf die verschiedenen Architekturkonzepte und deren Möglichkeiten wird ausführlich in [33] eingegangen.
Die vorliegende Arbeit hat sich mit dem Redesign von KCachegrind beschäftigt. KCachegrind ist ein Visualisierungswerkzeug für Profiling-Daten. Als eine solche Anwendung ist sie direkt von den Profilern abhängig. Ändern sich die von Profiling-Werkzeugen gelieferten Daten, muß auch KCachegrind angepaßt werden, sofern es nicht einen sehr flexiblen Aufbau aufweist.
Das für KCachegrind entwickelte Erweiterungskonzept versucht eine möglichst hohe Flexibilität zu bieten und für zukünftige Veränderungen bereit zu sein. Neben dem Grobkonzept wurden auch die notwendigen Schnittstellen ausgearbeitet. So etwas ist immer problematisch, da bei der Ausarbeitung einer Schnittstelle unmöglich alle denkbaren Situationen und Anwendungsfälle vorhergesehen werden können. Trotzdem ist dies das erstrebenswerte Ziel bei der Entwicklung von Interfaces.
Dem Redesign einer Anwendung geht immer eine Analyse voraus. Im Rahmen dieser Arbeit wurden Anforderungen und bestehende Architektur analysiert. Gerade hierbei sind eine gute Dokumentation und Kommentare im Quelltext sehr hilfreich. Viele Programmierer unterschätzen dies bei der Entwicklung von Anwendungen. Aufbauend auf der Analyse wurden verschiedene Lösungskonzepte erarbeitet und gegeneinander abgewogen,

[1] Änderungen an der Architektur

worin im Nachhinein betrachtet der eigentliche Schwerpunkt der Arbeit lag.
Anspruch an die neue Architektur war, daß verschiedene modulare Komponenten, wie Import-Filter und Visualisierungen, in KCachegrind eingebaut werden können. Außerdem sollte eine Möglichkeit zur interaktiven Steuerung von Profiling-Werkzeugen geschaffen werden. Zu diesem Zweck wurde ein Plugin-System, mit separaten Schnittstellen für die einzelnen Komponenten, entwickelt.

6.2 Ausblick

Die nächste Aufgabe besteht in der Umsetzung des erarbeiteten Konzeptes. Dabei ist nicht nur die Funktionalität der Schnittstelle auf Seiten von KCachegrind zu implementieren, sondern auch bestehenden, fest integrierten Importfilter und Visualisierungen zu extrahieren und in Form von Plugins anzubinden.
Mit der, durch das komponentenbasierte Design, gewonnenen Funktionalität wird KCachegrind zu mehr als nur einem Visualisierungs-Werkzeug. Es entwickelt sich in Richtung einer „Schaltzentrale“ für das Profiling von Programmen. Der Profiling-Vorgang wird in Zukunft aus der Oberfläche von KCachegrind ausgeführt werden können und die Visualisierung von Profiling-Daten ist bereits jetzt möglich. Im Gegensatz zu vergleichbaren Tools, hängt KCachegrind dabei aber nicht von einem bestimmten Profiler ab. Der Anwender kann das von ihm favorisierte Profiling-Werkzeug verwenden, ohne auf verschiedene Visualisierungsformen zu verzichten.
Im Rahmen des Optimierungsprozeßes einer Software stellt KCachegrind schon jetzt eine große Hilfe für den Programmierer dar. Einzig die Analyse der Daten muß der Anwender nach wie vor selbstständig durchführen. Hierbei wäre eine Art *Expertensystem* extrem hilfreich. Man könnte in KCachegrind eine Datenbank integrieren, die bei Performanzproblemen Hinweise und Lösungsvorschläge gibt. Ein erster, einfacher Ansatz könnte sein, vermutete Ursachen und grobe Hinweise auf die prinzipiellen Möglichkeiten zu geben, die beispielsweise bei einer sehr hohen Anzahl an Cache-Misses zur Verfügung stehen. Eine Vision die wohl weit in der Zukunft liegt, wäre, daß KCachegrind bei der Visualisierung von Daten bestimmte Algorithmen zur Analyse der *Bottlenecks* und *Hot Spots* anwendet und hierzu Informationen aus einer, über das Internet angebundenen, Datenbank zieht. Die Einträge in der Datenbank könnten ihrerseits von erfahrenen Anwendern stammen. Letzter Schritt des Optimierungsprozeßes ist die Überprüfung der Performanzsteigerung. Hierbei könnte der Programmierer, je nach Ergebnis (Verbesserung/Verschlechterung), durch KCachegrind einen entsprechenden Eintrag in die Datenbank tätigen. So würden die Anwender gegenseitig durch ihre Erfahrungen profitieren. Auf vielen Gebieten zeigt der Trend momentan auf „Knowledge Libraries“, deren effektiver Einsatz jedoch nicht ganz unproblematisch ist. Diese Thema ist sehr komplex, deshalb soll hier auch nur ein Denkanstoß gegeben werden.

Anhang A

TraceCost-Klassen

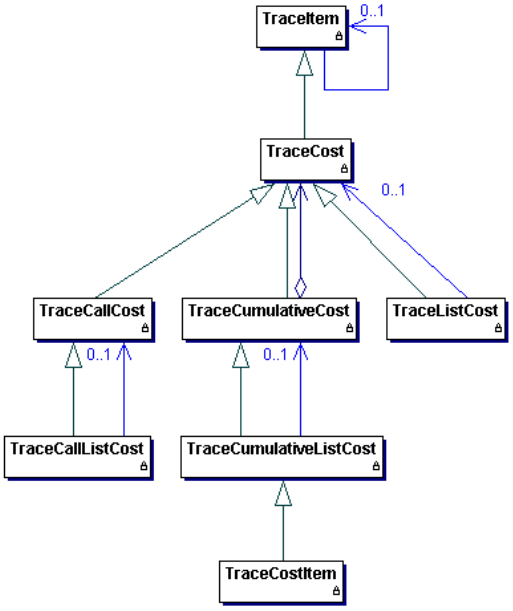

Abbildung A.1: Übersicht der TraceCost-Klassen

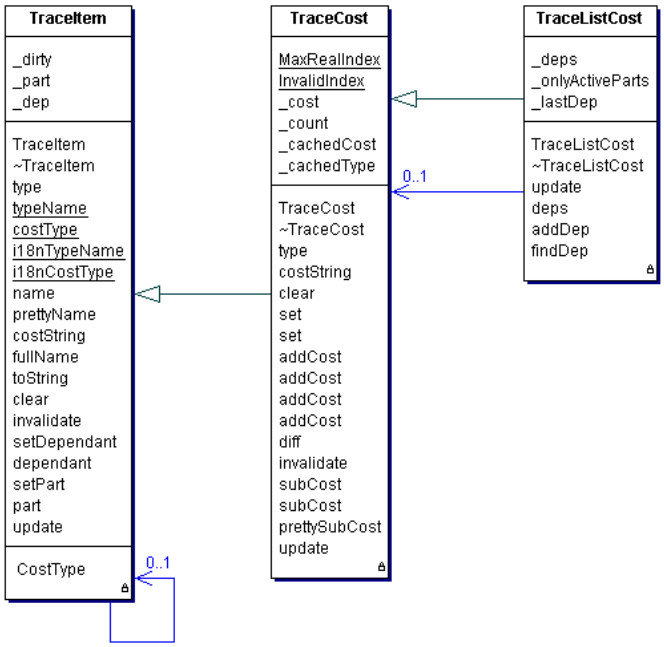

Abbildung A.2: Die Klassen TraceItem, TraceCost und TraceListCost

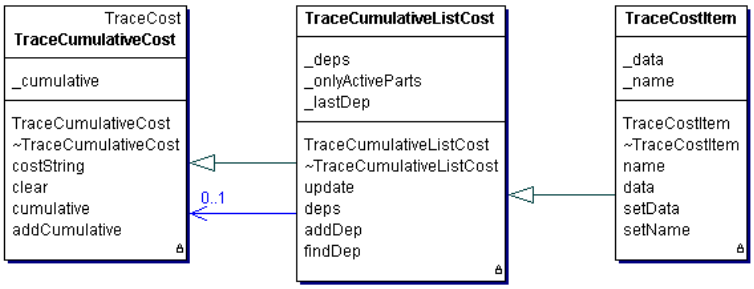

Abbildung A.3: Die Klassen TraceCumulativeCost, TraceCumulativeListCost und TraceCostItem

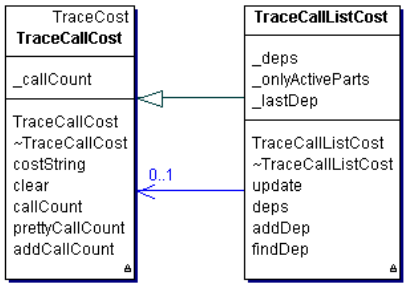

Abbildung A.4: Die Klassen TraceCallCost und TraceCallListCost

Anhang B

Datenmodell-Klassen

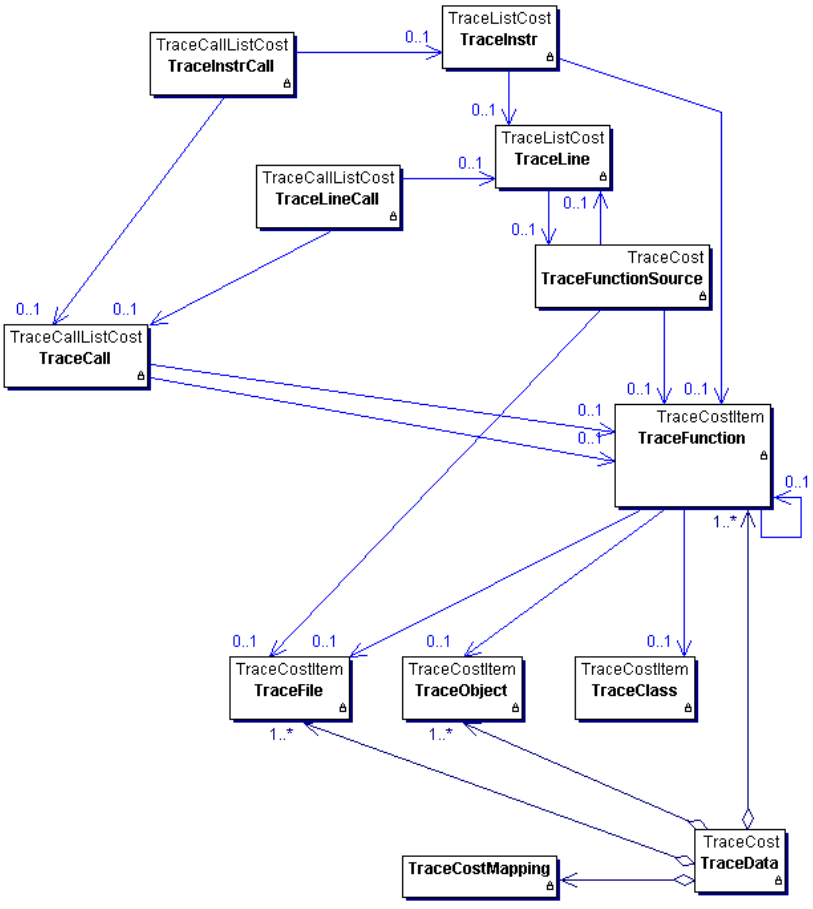

Abbildung B.1: Datenmodell

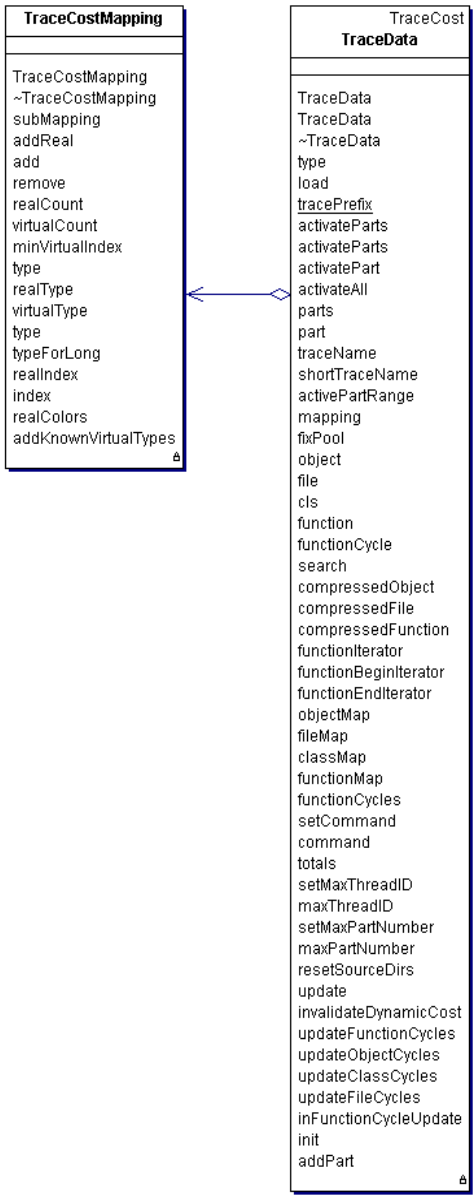

Abbildung B.2: Die Klassen TraceCostMapping und TraceData

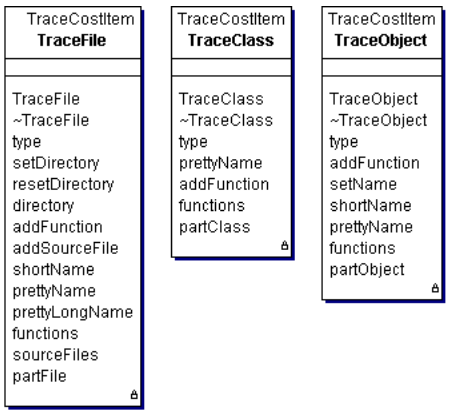

Abbildung B.3: Die Klassen TraceFile, TraceClass, TraceObject

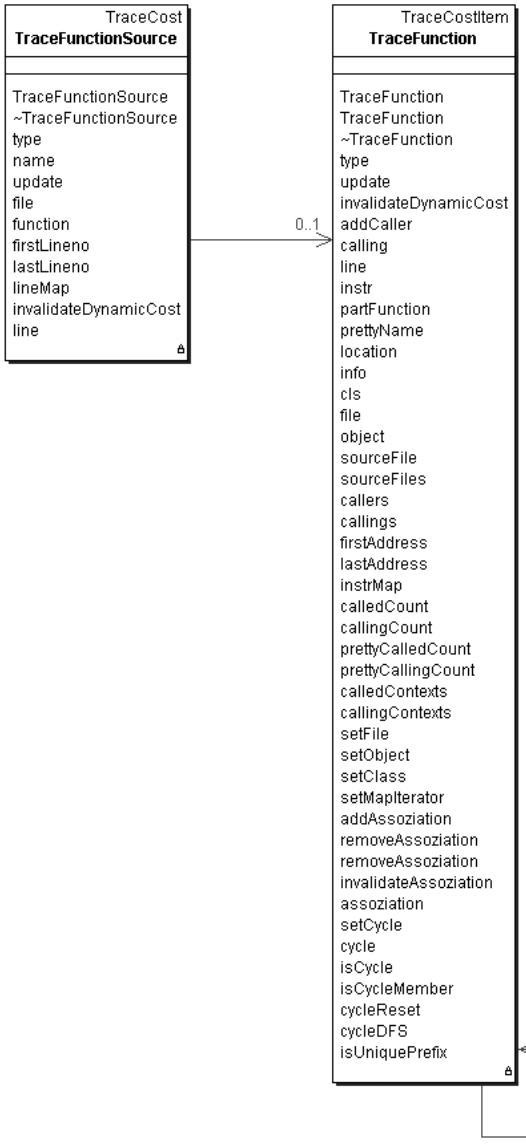

Abbildung B.4: Die Klassen TraceFunction und TraceFunctionSource

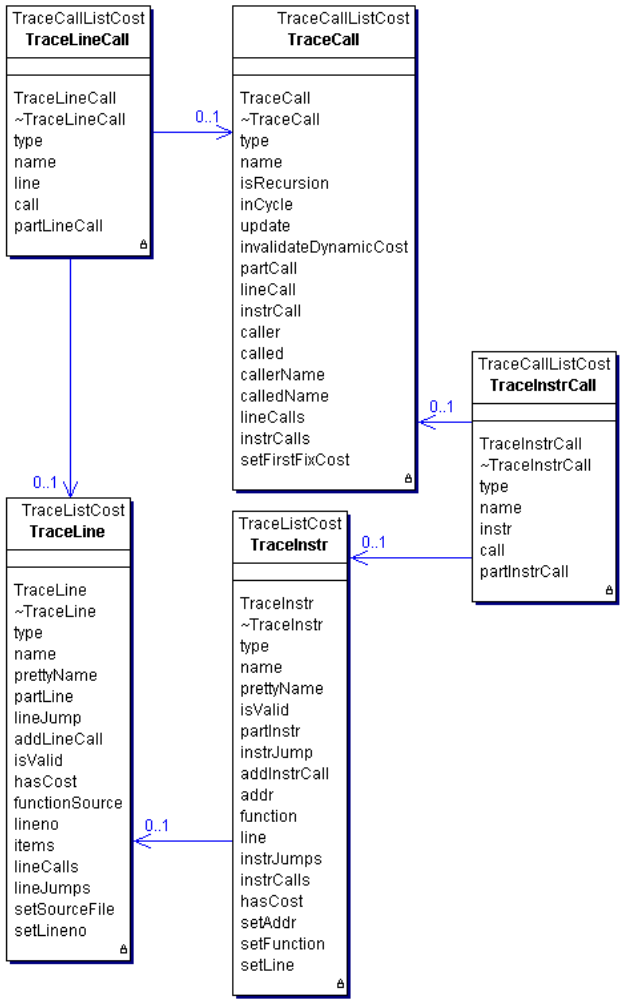

Abbildung B.5: Die Klassen TraceCall, TraceLineCall, TraceInstrCall, TraceLine und TraceInstr

Literaturverzeichnis

[1] Gordon E. Moore.: *Cramming more components onto integrated circuits*. Electronics, 38(8):114 – 117, April 1965

[2] D.C. Burger, J. R. Goodman, A. Kägi: *The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors*, Technical Report TR-95-1261, University of Wisconsin, Dept. of Computer Science, Madison, 1995

[3] Josef Weidendorfer, Markus Kowarschik, Carsten Trinitis: *A Toolsuite for Simulation Based Analysis of Memory Access Behavior*, International Conference on Computational Science (ICCS 2004), Krakov, Poland, June 2004

[4] Josef Weidendorfer: *Performance Analysis of GUI applications on Linux*

[5] Josef Weidendorfer: *KCachegrind - Profiling Visualization*, Homepage: http://kcachegrind.sourceforge.net/

[6] Peter Rechenberg, Gustav Pomberger: *Informatik-Handbuch*, Carl Hanser Verlag, 2. Auflage, 1999

[7] James Rumbaugh, Ivar Jacobson, Grady Booch: *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999

[8] Grady Booch, James Rumbaugh, Ivar Jacobson: *The Unified Modeling Language User Guide*, Addison-Wesley, 1999

[9] Sinan Si Alhir: *UML in a Nutshell*, O'Reilly, 1998

[10] Sinan Si Alhir: *Understanding Use Case Modeling*, 2000

[11] Bruce Eckel, Chuck Allison: *Thinking in C++, Vol. 2: Practical Programming*, 2. Auflage, 2002

[12] Jesse Liberty: *C++ Unleashed*, SAMS, 1998

[13] J. Seward, N. Nethercote: *Valgrind, an open-source memory debugger for x86-GNU/Linux*, Homepage: http://valgrind.kde.org/

[14] *Cygwin, a linuxlike environment for windows*, Homepage: http://www.cygwin.com/

[15] J. M. Anderson et al.: *Continous profiling: where have all the cicles gone?* Sixteenth ACM Symposium on Operating System Principles, ACM Operating Systems Review 31/5 (1997) 1-14

[16] S. Graham, P. Kessler, M. McKusick: *GProf: A Call Graph Execution Profiler*, SIGPLAN Notices Volume 39, Issue 4 (April 2004), SPECIAL ISSUE: 1982

[17] O. Traub, S. Schechter, M.D. Smith, *Ephemeral instrumentation for lightweight program profiling*, In Proceedings of PLDI ´00, 2000

[18] *Performance Management Guide*,
Homepage: http://publib16.boulder.ibm.com/pseries/en_US/aixbman/prftungd/prftungd02.htm

[19] *Oprofile*, Homepage: http://oprofile.sourceforge.net/

[20] Prasanna S. Panchamukhi, *Smashing performance with OProfile*, Homepage: http://www-106.ibm.com/developerworks/linux/library/l-oprof.html, 16 October 2003

[21] *pfmon*, Homepage: http://www.hpl.hp.com/research/linux/perfmon/pfmon.php4

[22] *Data Local Iterative Methods For The Efficient Solution of Partial Differential Equations*, http://www10.informatik.uni-erlangen.de/Research/Projects/DiME/

[23] J. Mayer, I. Melzer, F. Schweiggert, *Lightweight Plug-In-Based Application Development*, In: Proceedings of the Net.ObjectDays 2002, tranSIT GmbH, Ilmenau, Germany, October 2002, pp. 97-111.

[24] *Performance Application Programming Interface*, http://icl.cs.utk.edu/papi/

[25] *DynInst*, Homepage: http://www.dyninst.org/

[26] *Dynamic Probe Class Library*,
Homepage: http://oss.software.ibm.com/developerworks/opensource/dpcl/

[27] *DynaProf*, Homepage: http://www.cs.utk.edu/ mucci/dynaprof/

[28] *VTune*, Homepage: http://www.intel.com/software/products/vtune/

[29] *Cachegrind*, http://developer.kde.org/ sewardj/docs-2.0.0/cg_main.html

[30] *HPCView*, Homepage: http://www.cs.rice.edu/ dsystem/hpcview/

[31] J. M. Crummey, R. Fowler, G. Marin, *HPCView: A Tool for Top-down Analysis of Node Performance*, Journal of Supercomputing, 23:81-104, 2002.

[32] *Visual Profiler*, Homepage: ttp://www.ncsa.uiuc.edu/UserInfo/Resources/Software/Tools/VProf/

[33] A. Harrer, *Muster in der Software-Technik*, Vorlesungs-Skript Sommersemester 2002

Abbildungsverzeichnis

3.1 Ablauf eines Optimierungsvorgangs 20
3.2 Use Case Diagramm 22
3.3 Use Case: Interactive Profiling 22
3.4 Use Case: Analyze Program 23
3.5 Use Case: Compare Tracefiles 24
3.6 Datenimport 25
3.7 TracePart 26
3.8 TraceCost-Klassen 26
3.9 Datenmodell 27
3.10 Fenster der grafischen Oberfläche 28
3.11 Visualisierungs-Klassen 29
3.12 TreeMap-Klassen 29
3.13 Callgraph-Klassen 30

4.1 PlugInSystem 32
4.2 Datenmodell 33

5.1 Plugin-Konzept 38
5.2 PlugInLoader 39
5.3 Interfaces für Import-Filter 39
5.4 Interfaces für interaktives Profiling 40
5.5 Interfaces für Visualisierungs-Plugins 41

A.1 Übersicht der TraceCost-Klassen 44
A.2 Die Klassen TraceItem, TraceCost und TraceListCost 45
A.3 Die Klassen TraceCumulativeCost, TraceCumulativeListCost und TraceCostItem 45
A.4 Die Klassen TraceCallCost und TraceCallListCost 45

B.1 Datenmodell 46
B.2 Die Klassen TraceCostMapping und TraceData 47
B.3 Die Klassen TraceFile, TraceClass, TraceObject 48

B.4 Die Klassen TraceFunction und TraceFunctionSource 49
B.5 Die Klassen TraceCall, TraceLineCall, TraceInstrCall, TraceLine und TraceInstr 50