Cachegrind - Valgrind

Seminar: Leistungsanalyse unter Linux

Valgrind

Marlo Häring

Betreuer: Timo Minartz
Abgabedatum: 27.03.2012

Inhaltsverzeichnis

1 EINLEITUNG 3

2 SPEICHERAUFBAU 3

2.1 HEAP (DYNAMISCHER SPEICHER) 3
2.2 STACK 4

3 VALGRIND – ÜBERBLICK 5

4 MEMCHECK 6

4.1 WAS KANN MEMCHECK ERKENNEN? 6
4.2 WIE WIRD MEMCHECK BENUTZT? 7
4.3 MEMCHECK – ANWENDUNGSBEISPIEL 8
4.4 PROFILERSTELLUNG MIT MEMCHECK 9

5 CACHEGRIND 10

5.1 WAS KANN CACHEGRIND? 10
5.2 HARVARD-ARCHITEKTUR 11
5.3 WIE WIRD CACHEGRIND BENUTZT? 12

6 CALLGRIND 14

6.1 WAS KANN CALLGRIND? 14
6.2 WIE WIRD CALLGRIND BENUTZT? 14
6.3 CALLGRIND ANWENDUNGSBEISPIEL 15

7 KCACHEGRIND 16

8 MASSIF 18

8.1 WAS KANN MASSIF? 18
8.2 WIE WIRD MASSIF BENUTZT? 19

9 WAS KANN HELGRIND? 19

10 WAS MACHT VALGRIND MIT DEINEM PROGRAMM? 20

11 ZUSAMMENFASSUNG 21

ABBILDUNGS- UND QUELLENVERZEICHNIS 23

1 Einleitung

Ein sehr wichtiger Aspekt bei der Entwicklung eines Programms ist die Geschwindigkeit des Programmablaufs. Es geht dabei primär um die Optimierung von Ausführzeiten der Funktionen, die häufig durchlaufen werden. Der Entwickler muss einen Eindruck bekommen, in welchem Programmteil die meiste Zeit investiert wird. Dies erfährt man, indem ein Profil vom Programm erstellt wird, aus dem eine Aufteilung der Laufzeit auf die einzelnen Funktionen des Programms ersichtlich ist. Mithilfe dieser Übersicht kann der Programmcode gezielt im Hinblick auf die Laufzeit optimiert werden. Mit einer wiederholten Profilerstellung lassen sich anschließend Optimierungen auch konkret nachweisen. Das Tool zur Profilerstellung ist eines der Programme, die in dem Framework von *Valgrind* enthalten ist. Das wohl bekannteste Werkzeug der Serie ist *Memcheck*, das eine Vielzahl von Problemen in *C/C*++-Code aufzeigt. Besonders hilfreich ist es für Anfänger, da gerade bei *C/C*++ viele Fehler im direkten Umgang mit Speicherbereichen gemacht werden. Das Framework ist heutzutage schon so ausgereift, dass es Einsatz in vielen großen Softwareprojekten wie zum Beispiel bei dem Fenstermanager *KDE* findet.

Unterstützt werden mittlerweile die Architekturen *x86*, *amd64*, *ppc32* und *ppc64*. Um die genannten Aufgaben zu erfüllen, stellt *Valgrind* eine künstliche CPU zur Verfügung. Die Tools wurden modular gebaut, sodass sich Erweiterungen leicht realisieren lassen und die anderen bereits entwickelten Tools nicht behindern.

2 Speicheraufbau

2.1 Heap (dynamischer Speicher)

Der *Heap* stellt einen Speicherbereich zur Verfügung, der zur Laufzeit benutzbar ist. In der Programmiersprache *C* wird mit der *malloc*-Funktion ein Zeiger auf das erste Datenfeld des angeforderten Speichers zurückgegeben. Ist dies fehlgeschlagen, gibt die Funktion den Nullzeiger zurück. Auf diese Weise ist es möglich zur Laufzeit Speicher anzufordern. Dies ist dann nötig, wenn die benötigte Speichergröße beispielsweise von Angaben des Anwenders abhängt. Die *malloc*-Funktion braucht dabei die Größe in Bytes, die reserviert werden soll. Dabei wird häufig die Funktion *sizeof()* verwendet, die als Parameter beispielsweise einen primitiven Datentypen nimmt und dessen Größe in Bytes zurückgibt. Auf diese Weise können Datenfelder im *Heap* komfortabel reserviert werden.

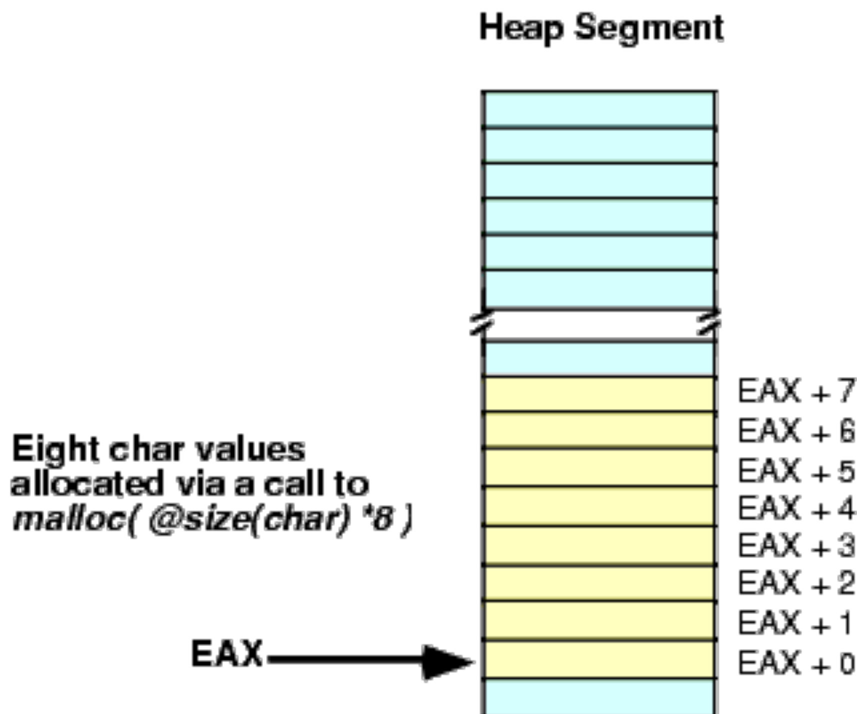

Abb. 1: Reservierung von 8 Bytes mit *malloc* auf den *Heap*
http://webster.cs.ucr.edu/AoA/Windows/HTML/images/MemoryAccessandOrg14.gif

Diese Adresse gilt es nicht zu verlieren, damit der Speicherbereich nach Gebrauch wieder ordnungsgemäß freigegeben werden kann, um ihn für folgende Reservierung zur Verfügung zu stellen. Dabei spielt es keine Rolle, in welcher Reihenfolge die Reservierungen und Freigaben von verschiedenen Speicherbereichen vorgenommen werden.

Programmiersprachen wie *Java* haben eine automatische Speicherverwaltung. In regelmäßigen Abständen wird geprüft, ob auf Speicherbereiche im *Heap* noch gültige Referenzen benutzt werden. Ist dies nicht der Fall, wird der Speicher wieder freigegeben. Das häufige Reservieren und Freigeben stellt allerdings einen hohen Verwaltungsaufwand dar, weil bei jeder Reservierung nach passenden Bereichen gesucht und gegebenenfalls umsortiert werden muss, wenn kein passender Bereich vorhanden ist.

2.2 Stack

Die Aufgabe des *Stacks* ist es, den Zustand des Programms abzuspeichern. Das beinhaltet insbesondere die Rücksprungadressen und die lokalen Variablen, die eine Funktion benötigt. Die Benutzung des *Stack*speichers ist meist schneller als das dynamische Reservieren auf dem *Heap*, da der organisatorische Aufwand entfällt. Allerdings kann es bei großen Mengen zu einem *Stackoverflow* kommen. Die Reihenfolge der Freigabe beim *Stack* ist festgelegt: der zuletzt reservierte Speicher wird zuerst wieder freigegeben (*last in - first out*). Bei moderner Technologie hat jeder *Thread* seinen eigenen *Stack*, da *Threads* heutzutage auch parallel berechnet werden können. *Heap* und *Stack* wachsen im Speicher gegeneinander.

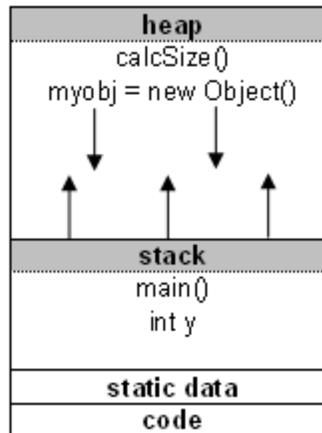

Abb. 2: *Stack* wächst von unten; *Heap* wächst von oben
http://www.maxi-pedia.com/web_files/images/HeapAndStack.png

3 Valgrind – Überblick

Valgrind ist eine Zusammenstellung aus verschiedenen Werkzeugen. Jedes einzelne Werkzeug hat seine eigenen Analyseschwerpunkte. Die Syntax der Tools ist allerdings einheitlich:

valgrind --tool=<toolname> <toolparameter> <programm> <programm parameter>

Auf diese Weise kann jedes Werkzeug komfortabel über die Befehlszeile aufgerufen werden. Die wichtigsten Tools sind folgende:

- *Memcheck* (Speicheranalyse)
- *Cachegrind* (Cacheanalyse)
- *Callgrind* (Laufzeitanalyse)
- *Massif* (Heapanalyse)

Dabei arbeitet *Valgrind* mit dem *dynamic binary instrumentation* (*DBI*) Ansatz. Hierbei werden zusätzliche Anweisungen an den Originalcode zur Laufzeit angefügt. Dies unterscheidet sich stark von dem *source level instrumentation* (*SLI*) Ansatz, bei dem der Analysecode schon vor dem Kompilieren dem Originalcode angefügt wird.

4 Memcheck

4.1 Was kann Memcheck erkennen?

Dieses Tool von *Valgrind* beschäftigt sich hauptsächlich mit der Speicherverwaltung. Dazu gehört die Allozierung von Speicherplatz, der Zugriff auf reservierten Speicherplatz und das Freigeben von Speicherplatz. Dies ist auch der gewünschte Weg, wenn es darum geht, dynamisch Speicher zur Laufzeit benutzen zu wollen. Sollte reservierter Speicher nicht freigegeben werden, kommt es zu Speicherlecks, die das Nutzen oder Freigeben von bereits reserviertem Speicherplatz nicht mehr ermöglichen. So würde Speicherplatz entstehen, der für den Rest der Laufzeit unbrauchbar ist.

Zur Problemlösung analysiert *Memcheck* das Lesen und Schreiben von Speicherbereichen. Außerdem werden alle Reservierungen und Freigaben abgefangen, die durch die Funktionen *malloc*, *new* und *free* durchgeführt werden. Dadurch wird es *Memcheck* möglich, Zugriffe auf nicht initialisierten Speicherbereich zu erkennen. Darüber hinaus wird das Lesen oder Schreiben auf bereits freigegebenen Speicherplatz erkannt. Weitere häufige Fehler sind das Lesen oder Schreiben außerhalb des reservierten Speichers. *Memcheck* kann diese Zugriffe ebenfalls erkennen. Reserviert man Speicher mit *malloc* muss dieser auch mit *free* freigegeben werden. *Memcheck* erkennt auch, ob für das Freigeben die passende Funktion zur Reservierung aufgerufen wurde, sodass mit *malloc* reservierter Speicher nicht mit *delete* freigegeben wird. Achten Entwickler nicht darauf, kann es zu nicht definierten Zuständen führen, wodurch meist extrem merkwürdige Folgefehler entstehen.

```
#include <iostream>
#include <stdlib.h>

int main()
{
	char *p1 = (char*)malloc(1);
	char *p2 = new char;

	delete(p1);
	free(p2);

	return 0;
}
```

In diesem Codebeispiel wird die Zeigervariable *p1* mit *malloc* initialisiert und *p2* mit *new*. Anschließend werden fälschlicherweise die *p1* Variable mit *delete* und *p2* mit *free* freigegeben.

```
==2980== Memcheck, a memory error detector
==2980== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==2980== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==2980== Command: ./main
==2980==
==2980== Mismatched free() / delete / delete []
==2980==    at 0x4028B6F: operator delete(void*) (vg_replace_malloc.c:457)
==2980==    by 0x80485D8: main (main.cpp:9)
==2980==  Address 0x42f2028 is 0 bytes inside a block of size 1 alloc'd
==2980==    at 0x4029ACC: malloc (vg_replace_malloc.c:263)
==2980==    by 0x80485B8: main (main.cpp:6)
==2980==
==2980== Mismatched free() / delete / delete []
==2980==    at 0x4028E58: free (vg_replace_malloc.c:427)
==2980==    by 0x80485E4: main (main.cpp:10)
==2980==  Address 0x42f2060 is 0 bytes inside a block of size 1 alloc'd
==2980==    at 0x4029685: operator new(unsigned int) (vg_replace_malloc.c:282)
==2980==    by 0x80485C8: main (main.cpp:7)
==2980==
==2980==
==2980== HEAP SUMMARY:
==2980==     in use at exit: 0 bytes in 0 blocks
==2980==   total heap usage: 2 allocs, 2 frees, 2 bytes allocated
==2980==
==2980== All heap blocks were freed -- no leaks are possible
==2980==
==2980== For counts of detected and suppressed errors, rerun with: -v
==2980== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 17 from 6)
```

Fehler dieser Art bleiben lange unentdeckt, weil sie schwer zu finden sind. Häufig laufen Programme erst einmal fehlerfrei mit Speicherleck. Sichtbar werden diese indirekt durch das Auftreten von Folgefehlern. Eine auf ein Speicherleck zurückzuführende Fehlerbehebung kann sehr zeitraubend sein. Um solche Fehler von vornherein zu vermeiden, sollte man von Zeit zu Zeit *Memcheck* über sein Softwareprojekt laufen lassen um sicherzustellen, dass es dabei nicht zum Verlust von Speicher kommt.

4.2 Wie wird Memcheck benutzt?

Wenn das Programm, von dem ein Profil erstellt werden soll, als Quelltext vorliegt, sollte es mit *-g* kompiliert werden, damit *Memcheck* in den Meldungen genaue Zeileninformationen angeben kann. *Memcheck* erstellt Profile auch über Programme, die nicht mit Debugginformationen kompiliert wurden, kann da aber kaum direkte Verweise auf den Quelltext geben.

Darüber hinaus ist es ratsam sein Programm mit *-O0* zu kompilieren, sodass der geschriebene Code nicht optimiert wird und die Fehler direkt auf den Quelltext bezogen werden können. Sollte dem Compiler erlaubt sein zu optimieren, ist es ihm auch möglich, den Quellcode zu ändern. Dies hat zur Folge, dass *Memcheck* keine

genauen Zeilenangaben ausgeben kann. Optimierungseinstellungen ab *-O2* sind also nicht mehr ratsam. *Memcheck* folgert dann manchmal irrtümlicherweise, nicht initialisierte Variablen gefunden zu haben, die eigentlich gar nicht existieren.

Die Profilerstellung wird ähnlich wie bei dem gewöhnlichen Programmstart gestartet. Das normale Starten eines Programms aus einer Kommandozeile heraus sieht wie folgt aus:

prog arg1 arg2

Um *Valgrind* zu benutzen, wird es mit den entsprechenden Parametern vor den gewöhnlichen Programmstart gesetzt.

valgrind --leak-check=yes prog arg1 arg2

Da *Memcheck* das Standardtool ist kann *--tool=memcheck* weggelassen werden. Mit dem Parameter *--leak-check* wird die detaillierte Speicherleckerkennung benutzt. Bei der Profilerstellung wird das Programm erheblich langsamer ablaufen und es kommt zu erhöhtem Speicherbedarf, da *Memcheck* zur Analyse auch Speicher in Anspruch nehmen muss.

4.3 Memcheck – Anwendungsbeispiel

Hier ein Beispielprogramm, welches zeigt, wie *Valgrind* ein Programm analysiert und Meldungen ausgibt.

```
1.#include <stdlib.h>
2.
3.void f(void)
4.{
5.   int *x = malloc(10 * sizeof(int));
6.     x[10] = 0;//dynmische   Reservierung   im    Heap   wird
   überlaufen
7.   //Reservierung von x wird nicht frei
8.}
9.
10.    int main(void)
11.    {
12.       f();
13.       return 0;
14.    }
```

Dieses Beispielprogramm enthält zwei Probleme.

Problem 1 (Speicherfehler):
In *Zeile 5* wird dynamisch – also zur Laufzeit – Speicherplatz im *Heap* reserviert. Dies erledigt die Funktion *malloc* und gibt anschließend eine Adresse auf das erste

Element im Speicher zurück. Nach der Ausführung von *Zeile 5* haben wir demnach die Möglichkeit, 10 *int*-Felder (0-9) im *Heap* zu benutzen. In *Zeile 6* jedoch wird versucht, das 10. Element auf „0“ zu setzen.

Problem 2 (Speicherleck):
Die Funktion *f* reserviert zur Laufzeit Speicher. Wichtig dabei ist, dass der reservierte Speicher auch wieder freigegeben wird, da er sonst für den Rest der Laufzeit reserviert bleibt und unbrauchbar wird. In diesem Beispiel wird der Speicher nicht wieder freigegeben. Wenn die Funktion *f* terminiert, geht der Wert der Variable von *x* verloren, wodurch die Adresse des dynamisch reservierten Speichers auch verloren geht. In diesem Fall würde es also zu einem Speicherleck kommen.

4.4 Profilerstellung mit Memcheck

Der oben gezeigte Quellcode wird mit der Befehlszeile „*gcc -g -O0 -o main main.c*“ kompiliert, wodurch Optimierung ausgeschlossen und Debuginformationen mit kompiliert werden. Die Profilerstellung des Programms wird mit dem Kommando „*valgrind --leak-check=yes ./main*“ eingeleitet. Das Programm nimmt nun ca. 20- bis 30-mal mehr Zeit in Anspruch und benötigt erheblich mehr Arbeitsspeicher. Nach der Profilerstellung zeigt *Memcheck* die gefundenen Fehler auf der Konsole an. Die meisten Fehlermeldungen sehen folgendermaßen aus:

```
==16008==    Invalid write of size 4
==16008==    at 0x400512: f (main.c:6)
==16008==    by 0x400522: main (main.c:12)
==16008==    Address 0x51c3068 is 0 bytes after a block of size 40 alloc'd
==16008==    at 0x4C28FAC: malloc (vg_replace_malloc.c:236)
==16008==    by 0x400505: f (main.c:5)
==16008==    by 0x400522: main (main.c:12)
```

Solche Fehlermeldungen enthalten viele Informationen, aus denen die wichtigsten herausgefiltert werden müssen, sodass der Fehler im Quelltext gefunden werden kann. Diese sieben Zeilen deuten auf den ersten Fehler hin, bei dem es im *Heap* zu einem fehlerhaften Zugriff kommt. Die Zahl, die eingeschlossen von den beiden Gleichheitszeichen steht, ist die *Prozess-ID* und häufig nicht relevant. Der darauf folgende Text „*Invalid write of size 4*“ macht deutlich, dass es beim Schreiben zu einem Fehler gekommen sein muss. Die Zeile darunter zeigt den *Stacktrace* auf, sowie die zugehörigen Adressen.

Der zweite Teil der Meldung zeigt auf, dass der Block auf den in der *Zeile 5* zugegriffen wird, hinter dem reservierten Speicherbereich liegt. Der Abstand zwischen den 40 reservierten Bytes und dem Speicher, auf den zugegriffen werden soll, beträgt 0 Bytes. Anschließend folgt auch wieder der *Stacktrace* mit den zugehörigen Adressen.

Aus dieser Meldung wird der erste Fehler schon sehr deutlich herauskristallisiert. Es wird deutlich, in welcher Zeile auf nicht reservierten Speicher zugegriffen wird und in welcher Zeile der Speicherbereich des Feldes dynamisch reserviert wird.

Fehler sollten immer in der Reihenfolge behoben werden, in der sie auftreten, da sie oftmals die Ursache für Folgefehler sind.

Eine Fehlermeldung für ein Speicherleck hat oftmals folgendes Schema:

```
==16077==    40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==16077==    at 0x4C28FAC: malloc (vg_replace_malloc.c:236)
==16077==    by 0x400505: f (programm_aus_buch.c:5)
==16077==    by 0x400522: main (programm_aus_buch.c:12)
```

Durch den *Stacktrace* wird deutlich an welcher Stelle ein Speicherleck erkannt wird. Hier ist beispielsweise zu sehen, dass es sich bei der Meldung um den reservierte Speicher in der Funktion *f* in *Zeile 5* handelt. Die am häufigsten vorkommenden Lecks sind die, bei denen dynamisch reservierter Speicher definitiv verloren ist. Derartige Lecks haben die höchste Priorität und sollten unbedingt beseitigt werden. Des Weiteren gibt es noch Meldungen, bei denen es wahrscheinlich zu Speicherlecks kommt. Dabei verliert das Programm an Speicher, es sei denn, der Zeiger auf den reservierten Speicher wird kontrolliert verschoben, wie zum Beispiel in die Mitte des Arrays.

5 Cachegrind

5.1 Was kann Cachegrind?

Dieses Tool hat die Aufgabe zu zeigen, welche Daten wann im Cache liegen. Zu diesem Zweck wird ein Profil einer sehr detaillierten Cachesimulation erstellt. Ziel ist es, eine möglichst hohe *Hitrate* zu erzielen. Die *Hitrate* ist der Quotient aus Anzahl der Datenzugriffe, die im Cache liegen und Anzahl aller Zugriffe. Die *Missrate* beschreibt das Gegenereignis, also die Wahrscheinlichkeit, dass ein Datum nicht im Cache liegt.

Bei der Profilerstellung hält *Cachegrind* für jede Programmzeile fest, welche *Hitrate* an der Stelle vorliegt, sowie die Anzahl der Speicherreferenzen die durch die Zeile dazugekommen sind. Diese Informationen lassen sich auch auf Funktionen oder Module abstrahieren. Bei den *x86*- und *amd64*-Architekturen ermöglicht die *CPUID*-Option *Cachegrind*, die Konfiguration für die Cachesimulation selber vorzunehmen.

Der gesamte Cache ist in verschiedene Levels aufgeteilt, da es nicht möglich ist, gleichzeitig große Caches mit schnellen Zugriffszeiten zu konstruieren. Aus diesem Grund ist heutzutage eine Cachehierarchie die am weitesten verbreitete Technik. Dabei werden Caches aufsteigend nach Größe und absteigend nach Zugriffszeit hintereinander geschaltet. Die Caches werden dann mit *Level 1* (*L1*), *Level 2* (*L2*) bis *Level x* bezeichnet. Dies ist die gleiche Reihenfolge, in der benötigte Daten in den Caches gesucht werden. Dabei wird das Auffinden einer benötigten Datei in einem der Caches als *Hit* bezeichnet. Kommt es zu einem *Hit* wird das Datum an die CPU

gegeben und in den *L1*-Cache übernommen. Sollte ein Datum nicht im Cache gefunden werden, wird es *Miss* genannt. Anschließend muss das benötigte Datum aus langsameren Hintergrundspeichern nachgeladen werden. Ein *Miss* kann je nachdem, in welchem Level ein Datum vermisst wird, 10 (*L1 miss*) bis 200 (*L2 Miss*) Zyklen kosten. Mit diesem Werkzeug ist es möglich, die Laufzeit seines Programms stark zu optimieren. Darüber hinaus kann geprüft werden, ob das Programm mit den heutigen Caches harmoniert, sprich zu normalen *Hit*raten kommt. Und schließlich berechnet *Cachegrind* die mittleren Zugriffszeiten.

Nehmen wir an, es sind folgende Werte gegeben:

- die Zugriffszeit auf den Cache *c*,
- die Hitrate *h*,
- die Zugriffszeit auf den Speicher *m*,

so gilt folgende Formel:

$$\text{Mittlere Zugriffszeit} = c + (1 - h)\, m$$

Haben wir eine Hitrate, die sich 100% nähert, läuft die Zugriffszeit auf *c* hinaus. Geht die Trefferrate allerdings gegen „0“, bildet sich die mittlere Zugriffszeit aus der Summe der Zeit *c*, die es dauert, die Daten im Cache vergeblich zu suchen, und der Zeit *m*, die es dauert die fehlenden Daten in den Cache zu laden. Mittlerweile gibt es Technologien, bei der schon während der Suche im Cache eine Speicheranfrage durchgeführt wird.

5.2 Harvard-Architektur

Sollen hohe Rechengeschwindigkeiten erreicht werden, kommt es meist auf zwei Faktoren an. Wichtig ist einerseits die Geschwindigkeit, mit der eine CPU die anstehenden Anweisungen abarbeiten kann, andererseits müssen die Anweisungen, die ausgeführt werden, rechtzeitig und schnell zur Verfügung stehen. Ist dies nicht der Fall, muss die CPU viele kostbare Zyklen warten bis die Anweisungen aus langsameren Speichern nachgeladen werden. In der Geschichte der Prozessortechnologie wurden CPUs immer schneller und konnten immer mehr Rechnungen in kürzerer Zeit durchführen, da es möglich wurde, immer mehr Transistoren auf gleichem Raum zu organisieren. Dieser Effekt ist auch bei Speichern eingetreten. Jedoch wird dadurch nur der Speicher größer, was wiederum zu erhöhten Zugriffszeiten führt.

So entwickelte sich die *Harvard-Architektur*, die zwei Caches benutzt: einen für die Befehle, ein weiterer für die Daten. Diese Architektur wird auch als *Split-Cache* bezeichnet und hat sich heutzutage bei modernen *CPUs* durchgesetzt. Der Vorteil ist, dass der Cache, der die Befehle hält, diese nicht wieder in den Speicher schreiben muss, da sich die im Laufe des Programmablaufs nicht ändern. Außerdem können Befehle und Daten parallel und unabhängig voneinander in die Rechenwerke geladen werden, was auch ein Vorteil gegenüber der *Von-Neumann-Architektur* ist, bei der zwei aufeinanderfolgende Zyklen benötigt werden.

5.3 Wie wird Cachegrind benutzt?

Wenn das Werkzeug *Cachegrind* auf das Zielprogramm angewendet werden soll, muss „*valgrind --tool=chachegrind*" vor die aufzurufende Kommandozeile geschrieben werden. Bei dem *ls*-Befehl, der mit dem Parameter *-l* ausgeführt werden soll, würde die entsprechende Befehlszeile dementsprechend folgendermaßen aussehen:

valgrind --tool=cachegrind ls -l

Die Ausgabe dieses Befehls könnte so aussehen (Ausschnitt):

```
==3476==
==3476== I   refs:      1,707,442
==3476== I1  misses:        4,573
==3476== LLi misses:        2,541
==3476== I1  miss rate:      0.26%
==3476== LLi miss rate:      0.14%
==3476==
==3476== D   refs:        702,550  (525,784 rd   + 176,766 wr)
==3476== D1  misses:        6,222  (  5,141 rd   +   1,081 wr)
==3476== LLd misses:        3,682  (  2,747 rd   +     935 wr)
==3476== D1  miss rate:       0.8% (    0.9%     +     0.6%  )
==3476== LLd miss rate:       0.5% (    0.5%     +     0.5%  )
==3476==
==3476== LL refs:          10,795  (  9,714 rd   +   1,081 wr)
==3476== LL misses:         6,223  (  5,288 rd   +     935 wr)
==3476== LL miss rate:        0.2% (    0.2%     +     0.5%  )
```

Hier ist ersichtlich, dass es sich bei der Auswertung um eine *Harvard-Architektur* handelt, da der erste Absatz den Cache der Befehle auswertet (*Instruction-Cache*) und der mittlere Teil den Cache der Daten (*Data-Cache*). Im dritten Teil ist der Cache mit dem höchsten Level noch einmal separat aufgeführt. Er setzt sich zusammen aus dem *LLi* (*last-level-instruction*) und dem *LLd* (*last-level-data*). In den ersten beiden Gruppen sind die absoluten Zahlen der Zugriffe aufgeführt, gefolgt von den absoluten *Misses* aus dem ersten und dem letzten Level. Bei dem Daten-Cache wird zwischen Lese- und Schreibzugriffen differenziert. Darauf folgt die relative *Miss*rate des ersten und letzten Levels.

Nach der Ausführung dieses Befehls kommt es allerdings nicht nur zu der Ausgabe dieses Textes. Es wird außerdem im aktuellen Verzeichnis eine Datei mit dem Namen „*cachegrind.out.<pid>*" erzeugt. Diese Datei ist zwar lesbar, allerdings sollte der „*cg_annotate*"-Befehl ausgeführt werden, da dann Werte für jede einzelne Funktion sichtbar werden.

```
I1 cache:         32768 B, 64 B, 4-way associative
D1 cache:         32768 B, 64 B, 8-way associative
LL cache:         8388608 B, 64 B, 16-way associative
Command:          ls -l
Data file:        cachegrind.out.3118
Events recorded:  Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Events shown:     Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Event sort order: Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Thresholds:       0.1 100 100 100 100 100 100 100 100
Include dirs:
User annotated:
Auto-annotation:  off
```

Im ersten Teil wird eine Zusammenfassung der Hardwarekonfiguration aufgeführt, damit feststeht, wozu diese Datei gehört und welcher Befehl damit analysiert wurde. Außerdem ist aufgeführt, welche Events aufgezeichnet wurden und in welcher Reihenfolge sie sortiert sind. Zudem ist es möglich, eine Grenze (*Threshhold*) anzugeben, sodass nicht zu viele Informationen von unwichtigen Funktionen ausgegeben werden.

Darauf folgt die Auswertung des gesamten Programms:

```
--------------------------------------------------------------------------------
       Ir  I1mr  ILmr      Dr  D1mr  DLmr      Dw D1mw DLmw
--------------------------------------------------------------------------------
1,332,561 2,127 1,761 409,759 3,766 2,390 116,921  768  673  PROGRAM TOTALS
```

Diese Zeile enthält die gleichen Informationen, wie die Ausgabe direkt nach dem Ausführen von *Cachegrind*.

Abschließend folgen die Informationen für jede einzelne Funktion.

```
--------------------------------------------------------------------------------
Ir I1mr ILmr      Dr D1mr DLmr     Dw D1mw DLmw  file:function
--------------------------------------------------------------------------------
464,816    1    1 139,460   21   20      0    0    0  /build/buildd/eglibc-
       2.13/string/../sysdeps/i386/i686/multiarch/../strcmp.S:__GI_strcmp
86,050   23   12  35,662  507  190 12,870    9    0  /build/buildd/eglibc-2.13/elf/dl-
       lookup.c:do_lookup_x
67,259   11   11  26,438  777  589  2,354    1    0  /build/buildd/eglibc-2.13/elf/dl-
       addr.c:_dl_addr
64,454   26   18  13,208    1    0 18,924    1    1  /build/buildd/eglibc-
       2.13/intl/l10nflist.c:_nl_make_l10nflist
50,860    2    2  18,549  178   83      0    0    0  /build/buildd/eglibc-
       2.13/string/../sysdeps/i386/i686/multiarch/../strcmp.S:strcmp
46,123    4    4   7,159   16   13      0    0    0  /build/buildd/eglibc-
       2.13/string/../sysdeps/i386/i686/multiarch/../../i586/strlen.S:__GI_strlen
40,257   16   10   8,967  170  129  5,082    9    0  /build/buildd/eglibc-2.13/elf/dl-
       lookup.c:_dl_lookup_symbol_x
```

Hier sind nun alle Informationen, die auch nach der Ausführung des *Cachegrind*-Kommandos angezeigt werden, für jede einzelne Funktion sichtbar, was nun einen

Anhaltspunkt zum Optimieren gibt. In der letzten Spalte sind die Dateinamen und die Funktionsnamen – durch einen Doppelpunkt getrennt – aufgeführt (Dateiname: Funktionsname).

Darüber hinaus gibt es sogar die Möglichkeit Informationen über jede Codezeile zu bekommen, indem „*cg_annotate*“ mit der *--auto* Option ausgeführt wird. Hierzu ein kleines Beispiel:

```
    Ir I1mr ILmr      Dr D1mr DLmr      Dw D1mw DLmw

     .    .    .       .    .    .       .    .    .  #include <stdio.h>
     .    .    .       .    .    .       .    .    .
     .    .    .       .    .    .       .    .    .  int main()
     4    0    0       0    0    0       1    0    0  {
50,006    0    0  20,001    0    0       1    0    0    for (int i=0;i<10000;i++)
20,000    0    0       0    0    0  20,000    0    0      printf("Hello world!");
     .    .    .       .    .    .       .    .    .
     1    0    0       0    0    0       0    0    0    return 0;
     2    0    0       2    0    0       0    0    0  }
```

6 Callgrind

6.1 Was kann Callgrind?

Callgrind lehnt sich stark an *Cachegrind* an, bietet aber noch zusätzliche Analysemöglichkeiten. Hier ist es auch möglich, die Beziehungen zwischen aufrufenden und aufgerufenen Funktionen zu betrachten. *Cachegrind* verteilt die Laufzeitanteile auf die Funktionen und unterscheidet dabei zwischen *inklusiver* und *exklusiver* Laufzeit. Es addieren sich bei *inklusiven* Kosten alle Kosten der Funktionen, die aus einer Funktion aufgerufen werden dazu. So sind die *inklusiven* Kosten der *main*-Funktion fast 100%. Der einzige Grund, warum es im Ergebnis nicht exakt 100% ergibt, erklärt sich durch vernachlässigbare kleine Kosten, die für die Initialisierung zustande kommen. Die *exklusiven* Kosten einer Funktion berücksichtigen nur die Kosten der eigenen Anweisungen.

6.2 Wie wird Callgrind benutzt?

Um *Callgrind* für die Profilerstellung eines Programms nutzen zu können, sieht die Syntax folgende Schreibweise vor:

valgrind --tool=callgrind <callgrind Parameter> Programmaufruf <Parameter>

Auf diese Weise wird die Ausführung des Programms erheblich langsamer, da der auszuführende Code wieder mit hinzugefügten Steueranweisungen, die für die Profilerstellung benötigt werden, an eine virtuelle CPU umgeleitet wird. Nachdem das

Programm terminiert, erstellt *Valgrind* in dem aktuellen Verzeichnis eine Datei mit dem Namen „*callgrind.out.<process id>*“. In dieser Datei sind nun alle Informationen über die Profilerstellung festgehalten und lassen sich mit diesem Befehl auslesen:

callgrind_annotate <parameter> callgrind.out.<process id>

Nun werden die aufwendigsten Funktionen absteigend ausgegeben. Zu jeder Funktion wird die absolute Anzahl an Anweisungen angegeben. So sorgt der Parameter *--auto=yes* dafür, dass alle relevanten Quellcodedateien, die gefunden werden, mit der Anzahl der Aufrufe pro Zeile ergänzt werden. Auf diese Weise wird ersichtlich welche Funktionen oder Programmzeilen optimiert werden sollten, um die größtmögliche Laufzeitverbesserung zu erzielen.

6.3 Callgrind Anwendungsbeispiel

Vom folgenden Programm soll nun ein Profil mithilfe des Werkzeugs *Callgrind* erstellt werden. Dazu wird der Quelltext mit folgendem Befehl kompiliert:

g++ -g –O2 main.cpp

Debuginformationen werden erstellt; die Optimierungsstufe beträgt 2. Das ist der Zustand, in dem das Programm freigegeben werden würde. Um die Profilerstellung zu starten wird folgender Befehl ausgeführt:

valgrind --tool=callgrind -v --dump-every-bb=1000000 ./a.out

Hier starten wir *Valgrind* mit dem Tool *Callgrind* und lassen uns zusätzliche Ausgaben mit dem Parameter *-v* anzeigen. Nach der Terminierung der verlangsamten Ausführung erstellt *Callgrind* in dem aktuellen Verzeichnis die Datei, die die Ergebnisse enthält. Mit diesem Befehl lassen sich die Informationen ausgeben:

callgrind_annotate --auto=yes callgrind.out.<process id>

Am Anfang werden die Funktionen nach Anweisungen sortiert aufgelistet. In jeder Zeile wird einmal die absolute Zahl der Anweisungen ausgegeben; anschließend folgen der Dateipfad und der Funktionsname. Auch hier ist es möglich, einen Schwellenwert zu definieren, um nicht von Informationen überflutet zu werden. Meist sind nur die Funktionen mit hohem Aufwand wichtig, um Anhaltspunkte für Optimierungen zu haben. Darauf folgt dann eine Übersicht der Quelldateien, die *Callgrind* als wichtig empfindet, und schreibt die *exklusiven* Kosten zu jeder Zeile dazu.

```
--------------------------------------------------------------------------------
-- User-annotated source: main.c
--------------------------------------------------------------------------------
    Ir

     .  #include <stdio.h>
     .
     .  int main()
     7  {
20,000    for (int i=0;i<10000;i++)
     .      printf("Hello world!");
     .
     .    return 0;
     4  }
```

Im Beispiel oben kann man erkennen, dass die *for*-Schleife 20.000 Anweisungen durchführt. Diese Zahl ist leicht nachvollziehbar, da pro Schleifendurchlauf ein Vergleich und eine Addition benötigt werden. Da die *for*-Schleife 10.000-mal durchlaufen wird, kommen wir auf 2 * 10.000 Anweisungen.

Callgrind bietet weitere Einstellungsmöglichkeiten. Beispielsweise ist es zur gezielten Analyse einer Funktion möglich, die Profilerstellung schon mit dem Aufruf der Funktion beginnen zu lassen (*--dump-before=function*). Alternativ kann mit ihr auch erst im Anschluss der Funktion begonnen werden (*--dump-after=function*). Schließlich kann auch eingestellt werden, von wo an der Anweisungszähler zu zählen beginnen soll (*--zero-before=function / --zero-after=function*). Auf diese Weise kann man selektive Informationen bei der Profilerstellung bekommen.

7 KCachegrind

Alle Informationen, die die Tools *Cachegrind* und *Callgrind* erzeugen, befinden sich in Dateien, die erstellt werden, nachdem die Werkzeuge terminieren. Diese Dateien lassen sich jetzt mit verschiedenen Parametern zu den dazugehörigen *annotate*-Befehlen auslesen. Da dies recht aufwändig ist, und damit sich der Anwender nicht alle notwendigen Parameter-Einstellungen merken muss, wurde das Programm *KCachegrind* entwickelt. Dies stellt ein GUI bereit, das ausschließlich mit *Qt4* entwickelt wurde und die Massen an Informationen übersichtlich darstellt.

Wird das Programm direkt mit dem Befehl „*kcachegrind*“ aus der Kommandozeile heraus gestartet, sucht es automatisch nach bestehenden Profildateien, die in dem aktuellen Verzeichnis bereits vorliegen, und liest diese ein. Bei der Profildatei vom *Helloworld*-Programm kommt es dabei zu folgender Ausgabe:

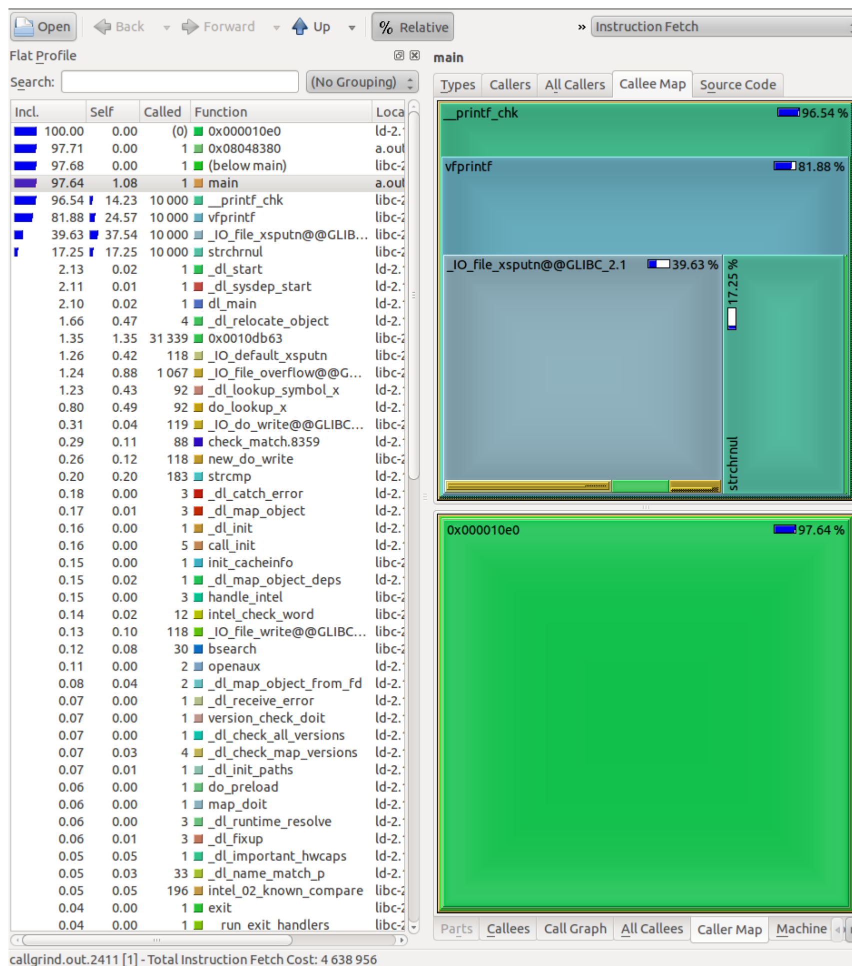

Abb. 3: *Kcachegrind* Übersicht

Das Programm zeigt auf der linken Seite eine Liste aller Funktionen, die aufgerufen wurden. Zu jeder Funktion stehen die inklusiven und exklusiven Kosten, die Anzahl der Aufrufe, sowie der Ort, von dem die Funktion aufgerufen wurde. Diese Liste lässt sich natürlich – wie gewohnt – nach jeder Spalte sortieren, damit je nach Optimierungswunsch die relevanten Funktionen oben stehen. Man kann sich wahlweise die relativen oder die absoluten Werte anzeigen lassen, indem man auf die *% Relative*-Schaltfläche klickt.

Auf der rechten Seite hat der Benutzer nun verschiedene Möglichkeiten, sich zu der links ausgewählten Funktion verschiedene andere Informationen anzeigen zu lassen. Auf dem Ausschnitt sehen wir zu der *main*-Funktion rechts oben die *Callee Map*. Diese zeigt, welche Funktionen die aktuell ausgewählte Funktion aufruft. Dabei passen sich die Flächengrößen proportional der Anzahl der Anweisungen an, sodass schnell erkennbar ist, welche Funktionen die meisten Anweisungen in Anspruch nehmen. Außerdem wird durch die dreidimensionale Anzeige die dazwischen liegende Anzahl der Funktionsaufrufe (*Distanz*) ersichtlich.

Im unteren Bereich des rechten Teils ist die *Caller Map*. Diese zeigt an, welche Funktion die aktuell ausgewählte aufruft. Die Darstellungsart ist hierbei identisch.

Ein weiteres Register der Maske zeigt den *Call Graph*:

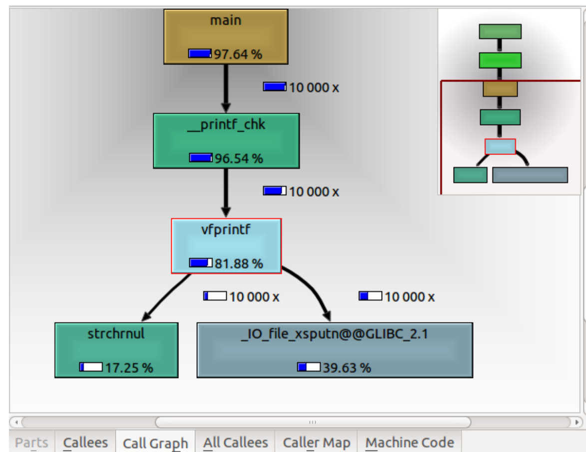

Abb. 4: *Kcachegrind* Aufrufsgraph

Hier sieht man jede Funktion als Knoten eines gerichteten Graphen. Auf diese Weise ist der Programmfluss leicht nachvollziehbar. Außerdem werden ausschließlich die teuersten Funktionen angezeigt.

Darüber hinaus ist es wie mit *callgrind_annotate* möglich, den Aufwand direkt auf die einzelne Codezeile abzubilden. Die Anweisungen lassen sich sogar als einzelne Assemblerzeile betrachten.

Mit *--dump-every-bb=count* kann zusätzlich festgelegt werden, dass automatisch nach einer bestimmten Anzahl an Anweisungen neue Profildateien angelegt werden.

Es gibt auch die Option, das Programm *kcachegrind* zuerst und anschließend die Profilerstellung zu starten. In *kcachegrind* kann manuell festgelegt werden, wann Profildateien erstellt werden sollen. Diese brauchen – je nach Umfang – wenige Augenblicke für die Erstellung und werden dann direkt in *kcachegrind* angezeigt. Der Dateiname setzt sich wie folgt zusammen: *callgrind.out.<process id>.<aufsteigende Nummer>*. Entdeckt das Programm mehrere Profildateien im aktuellen Pfad, öffnet *kcachegrind* alle Dateien in einer Instanz und im GUI kann komfortabel zwischen den einzelnen Dateien gewechselt werden. Dies ist bei einer automatischen Profilerstellung mit der *--dump-every-bb=count* Option sehr hilfreich, da dort häufig recht viele Dateien erstellt werden.

8 Massif

8.1 Was kann Massif?

Mit *Massif* kann ein Profil für den *Heap* erstellt werden. Dabei wird ermittelt, wie viel vom *Heap* für das Programm benutzt wird. Dies kann hilfreich sein, um den Speicherverbrauch zu reduzieren und in der Folge die Laufzeit des Programms zu

optimieren. Als Entwickler sollte man anstreben, möglichst wenige Seiten im Speicher zu gebrauchen, damit das Risiko eines Seitenfehlers reduziert wird. Dabei muss der Inhalt, der momentan nicht im Speicher vorliegt, vom langsameren Speicher nachgeladen werden. Außerdem wird deutlich, welche Bereiche im Programm wie viel Speicher im *Heap* reservieren.

8.2 Wie wird Massif benutzt?

Um *Massif* zu benutzen, ist es ratsam, sein Programm mit Debuginformationen zu kompilieren. Der Optimierungslevel beim Kompilieren wirkt sich nicht auf das Ergebnis der Profilerstellung aus. Anschließend wird das Programm mit *Massif* aufgerufen:

valgrind --tool=massif <programm> <paramater>

Auch bei dieser Profilerstellung kommt es zu einer verlangsamten Ausführung. Anschließend wird im aktuellen Verzeichnis eine Datei mit dem Namen *massif.out.<process id>* erstellt.

Mit dem Kommando *ms_print <filename>* ist es möglich, die Datei auszulesen und anzeigen zu lassen. Bei der Ausgabe werden im Kopf erst einmal der zugehörige Befehl und die Parameter, mit denen die Profilerstellung gestartet wurde, ausgegeben. Im Anschluss daran wird ein Graph gezeichnet, der die Größe des dynamisch reservierten Speichers auf dem *Heap* skizziert. Standardmäßig wird auf der x-Achse die Zeit und auf der y-Achse die Speichergröße dargestellt. Bei kleineren Programmen ist nicht viel zu erkennen, da die meisten Anweisungen Initialisierungsanweisungen sind. Deshalb bilden die Speicherreservierungen nur den letzten Ausschnitt des Graphen. So ist es möglich, die Zeiteinheit nicht von den Anweisungen abhängig zu machen, sondern von den Reservierungen und Freigaben von Speicher, wodurch ein aussagekräftigerer Graph entsteht.

9 Was kann Helgrind?

Helgrind ist ein Tool, mit dem man Synchronisationserrors in *C*-, *C++*- und *Fortran*-Programmen finden kann. Das Programmieren von parallelen Arbeitsabläufen wird meistens mit *Threads* realisiert, die aufgrund modernster Technologien auch parallel ausgeführt werden können. Dabei müssen das Zugreifen und Lesen von Daten, die von mehreren Threads genutzt werden, organisiert werden. Dies wird häufig durch das Setzen und Freigeben von Bits realisiert, sodass der Zugriff auf Daten in *kritischen Pfaden* organisiert abläuft. Dabei wartet ein *Thread* solange, bis ein gesetztes Bit aufgehoben wird, sodass die weitere Ausführung unbedenklich ist. Probleme ergeben sich, wenn Variablen von verschiedenen *Threads* in falscher Reihenfolge gesperrt werden, sodass diese sich gegenseitig blockieren und das Programm nicht weiter ausgeführt werden kann. Dies wird *deadlock* genannt. Fehler dieser Art versucht *Helgrind* zu detektieren und davor zu warnen.

10 Was macht Valgrind mit deinem Programm?

Bei der Entwicklung von *Valgrind* wurde darauf geachtet, dass die Profilerstellung so wenig Einfluss wie möglich auf das Zielprogramm vornimmt. Daher kann *Valgrind* auch auf bereits kompilierte Programme ausgeführt werden. Spezielle Einstellungen für die zusätzliche Erstellung von Debuginformationen sind nicht notwendig. So lässt sich *Valgrind* mit dem folgenden Befehl auch auf den *ls* Befehl aufrufen. Zusätzlich ist es natürlich auch möglich, verschiedene Parameter anzufügen:

valgrind --tool=memcheck ls –l

Valgrind übernimmt beim Starten des Programms die Kontrolle; unabhängig vom angewandten Werkzeug. Abhängig vom Werkzeug werden Instruktionen eingefügt und an eine virtuelle CPU weitergegeben. Dabei werden die zuvor eingefügten Instruktionen von dem Werkzeug benutzt, um den weiteren Verlauf der Ausführung zu steuern. Die Ausführung weicht dabei von der normalen Ausführung ab, da jeder Speicherzugriff oder und jede Variablenzuweisung durch entsprechenden Code simuliert und auf diese Weise mitgeloggt wird. Der Code wird zur Laufzeit disassembliert, wodurch eine *intermediate Representation* (*IR*) entsteht. Dabei handelt es sich um eine Zwischenversion, die der Compiler aus dem Quellcode erzeugt, bevor dieser in den Maschinencode übersetzt wird. Die *IR* des Originalcodes ist dabei eine maschinen- und programmiersprachenunabhängige Darstellung, auf die viele Compileroptimierungen aufbauen. Auf dieser Ebene fügt das entsprechende Tool zusätzliche Anweisungen ein. Der entstehende Code wird dann wieder in Maschinencode kompiliert und ausgeführt.

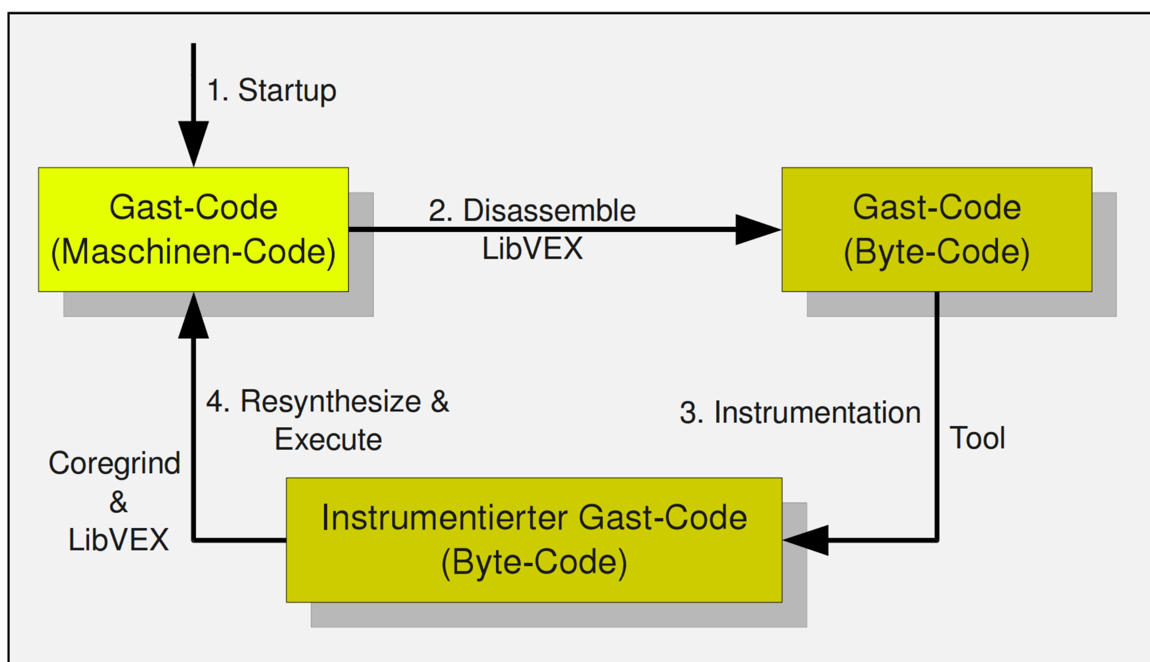

Abb. 5: *Valgrind* Vorgehensweise
http://os.inf.tu-dresden.de/papers_ps/pohle-diplom.pdf

1. Der Originalcode wird an *Valgrind* gegeben
2. Der Originalcode wird in den *IR* umgewandelt
3. Der Code für die Instrumentalisierung wird angehängt
4. Code wird wieder kompiliert und ausgeführt.

So muss beispielsweise beim Aufruf eines *Systemcalls* sichergestellt werden, dass *Valgrind* nicht die Kontrolle über das Programm verliert. Daher wird vorher der *Stackpointer* gesichert, und die virtuellen Registerinhalte werden in die echten Register kopiert; mit Ausnahme des Befehlszählers. Jetzt wird der *Systemcall* ausgeführt, anschließend die Registerinhalte wieder in ihre virtuellen Repräsentationen verschoben und der *Stackpointer* wiederhergestellt, sodass *Valgrind* den weiteren Verlauf bestimmt. Dies ist die Vorgehensweise bei einem *Systemcall*, da die Kontrolle des Programmflusses dabei kurz an das Betriebssystem abgegeben werden muss. Hier drin liegt der Grund, warum es bei der Anwendung eines Werkzeugs zu einer um Faktor 10 bis 50 längeren Laufzeit kommt.

Es müssen aber nicht unbedingt zusätzliche Werkzeuge auf das Zielprogramm angewendet werden. Baut man keine zusätzlichen Anweisungen ein, wird der auszuführende Code direkt an die virtuelle CPU weitergegeben. Bei dieser Einstellung dauert die Ausführung des Programms nur ungefähr vier Mal so lange.

Die Werkzeuge von *Valgrind* analysieren die Programmanweisungen in allen Bibliotheken, die benutzt werden. So wird beispielsweise auf Speicherlecks in den benutzten *GNU-C*-Bibliotheken hingewiesen, die aus vielen Zeilen von Code bestehen, die jede für sich analysiert werden. Besteht keine Möglichkeit, den Code in vorkompilierten dynamischen Bibliotheken zu verändern, können Dateien, die nicht betrachtet werden sollen, in einer Datei aufgelistet werden. Diese Liste wird vor dem Start von *Valgrind* ausgelesen, sodass *Valgrind* diese Dateien nicht mehr berücksichtigt.

11 Zusammenfassung

Valgrind ist in der Lage, viele Speicher- und Logikfehler aufzuspüren. Der Entwickler wird bei seinen Bemühungen unterstützt, die Anzahl von Fehlern (und Folgefehlern) zu reduzieren. Außerdem werden „Flaschenhälse“ erkannt, und Programme können gezielt optimiert werden. Der Quellcode von *Valgrind* ist frei verfügbar und kostenlos, steht unter der *GNU General Public License*. Deshalb können diese Tools auch noch spezialisiert und an den jeweiligen Anwendungszweck angepasst werden. Es spielt keine Rolle, um was für eine Zielanwendung es sich dabei handelt. *Valgrind* eignet sich für Datenbanksysteme, Spiele, Browser, Netzwerk-Serverprogramme und mehr. Auch ist die Programmiersprache nicht durch *Valgrind* festgelegt. Es lässt sich außer *C/C++* auch auf *Java*, *Perl*, *Python*, *Assembler*, *Fortan*, *Ada* und weitere Programmiersprachen anwenden. Auf den folgenden gängigen Betriebssystemen und Architekturen ist *Valgrind* anwendbar:

- **x86/Linux:** support is mature and almost complete.
- **AMD64/Linux:** support is mature and almost complete.
- **PPC32/Linux:** support is new but fairly complete.
- **PPC64/Linux:** support is new but fairly complete.
- **x86/Darwin (Mac OS X):** support is new.
- **AMD64/Darwin (Mac OS X):** support is new.
- **S390X/Linux:** support is new in 3.7.0.
- **ARM/Linux:** support for ARMv7 is fairly complete.
- **ARM/Android:** support is new in 3.7.0.

Durch die Abstraktion der Cachesimulation mit mehreren Levels sind keine aufwändigen Hardwarezugriffe nötig, sodass sich diese Tools auf alle Architekturen anwenden lassen, die auf diese Weise aufgebaut sind. *Valgrind* ist für Linux-Entwickler zu einem fast unverzichtbaren, mächtigen Werkzeug geworden, mit dem viel Zeit gespart werden kann. Durch den Einsatz dieses Tools gelangen schließlich stabilere und schnellere Programme an den Endanwender, und dies sind heutzutage die beiden wohl wichtigsten Faktoren für Softwareprodukte.

Abschließend eine Liste von Projekten, bei denen *Valgrind* zum Einsatz kam bzw. immer noch kommt:

Firefox, OpenOffice, StarOffice, AbiWord,
Opera, KDE, GNOME, Qt,
libstdc++, MySQL, PostgreSQL, Perl,
Python, PHP, Samba, RenderMan,
Nasa Mars Lander software, SAS, The GIMP,
Ogg Vorbis, Unreal Tournament, Medal of Honour, RenderMan
...

Abbildungsverzeichnis

Abb. 1: Reservierung von 8 Bytes mit *malloc* auf den *Heap* 4
Abb. 2: *Stack* wächst von unten; *Heap* wächst von oben 5
Abb. 3: *Kcachegrind* Übersicht 17
Abb. 4: *Kcachegrind* Aufrufsgraph 18
Abb. 5: *Valgrind* Vorgehensweise 20

Quellenverzeichnis

Bücher

- J. Seward, N. Nethercote, J. Weidendorfer: Valgrind 3.3. Network Theory Ltd, 2008
- Andrew S. Tanenbaum: Computerarchitektur, Pearson Studium, 2005

Links

- http://dot.kde.org/2006/02/20/interview-valgrind-author-julian-sewad
- http://haifux.org/lectures/239/ValgrindLecture.pdf
- http://valgrind.org/docs/manual/manual.html
- http://www.elektronik-kompendium.de/sites/com/0309291.htm
- http://de.wikipedia.org/wiki/Harvard-Architektur
- http://www.mixed-mode.de/fileadmin/templates/Dokumente/Downloads/Emb_SW_Report_Sep09_Valgrind.pdf
- http://lttng.org/tracingwiki/index.php/Dynamic_Binary_Instrumentation
- http://os.inf.tu-dresden.de/papers_ps/pohle-diplom.pdf
- http://valgrind.org/docs/valgrind2007.pdf
- http://valgrind.org/docs/callgrind2004.pdf
- http://www.maxi-pedia.com/what+is+heap+and+stack
- http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/16-Intermediate-Rep.pdf