Valgrind

Proseminar Effiziente Programmierung in C

Maximilian Hartz

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

13.06.2021

Gliederung (Agenda)

Was ist Valgrind [1, 2, 3]

- Analyse Programm
- 2002
- Julian Seward
- Open Source (GNU GPL v2)
- Kommandozeile:
  `valgrind [valgrind parameter] --tool=<Toolname> programm [programm parameter]`
- Valgrind-Core + Tool

Was macht Valgrind [1, 2, 3]

- kein neu-kompilieren notwendig
- Just-in-time-compiler
  - Maschinencode $\rightarrow$ Zwischensprache $\rightarrow$ Maschinencode
  - Zwischensprache (Vex) plattformunabhängig
- übersetzt auch shared- und static-libraries

Valgrind Technik [1, 2]

- blockweise Übersetzung
- virtuelle Register im Hauptspeicher
- Übersetzung:
  1. Maschinencode → Vex
  2. Tool bearbeitet Vex
  3. Vex → Maschinencode
- sehr viel Optimierung

Zu Beachten [1, 2]

- debug-info (`-g` Flagge) aktivieren
- kein Multithreading
  alles wird seriell ausgeführt

```
Thread 1                |Thread 2
*teste(ptr != 0)        |
                        | ptr = 0
func(ptr)               |
```

- schlechte Performance

Tools [1]

- Memcheck: Speichermanagement-Analyse
- Cachegrind: Profiler für Cache und Branch-prediction
- Callgrind: Cache Profiler und Call-graph-generator
- Helgrind: findet Fehler in Multithreading
- Massif: Profiler für den Heap
- Nulgrind: ein Tool das nichts macht
- und viele mehr

Memcheck

- Speichermanagement
- malloc
- free
- Pointer

Problem 1

```
5  void func(){
6      int* ptr = malloc(sizeof(int));
7      free(ptr);
8      int val = *ptr;
9  }
```

Listing 1: nicht allokierter Speicher

- undefiniertes Verhalten
- kann ggf. einen Speicherfehler hervorrufen

Fehlermeldung 1

```
5  void func(){
6      int* ptr = malloc(sizeof(int));
7      free(ptr);
8      int val = *ptr;
9  }
```

Listing 2: nicht allokierter Speicher

```
==9310== Invalid read of size 4
==9310==    at 0x109167: func (main.c:8)
==9310==    by 0x109181: main (main.c:13)
...
```

Listing 3: Memcheck Ausgabe

Fehlermeldung 1

```
5  void func(){
6      int* ptr = malloc(sizeof(int));
7      free(ptr);
8      int val = *ptr;
9  }
```

Listing 4: nicht allokierter Speicher

```
...
==9166==  Address 0x4a56040 is 0 bytes inside a block of
    ↪ size 4 free'd
==9166==    at 0x483CA3F: free (in ...)
==9166==    by 0x10918E: func (main.c:7)
==9166==    by 0x1091AC: main (main.c:13)
==9166==  Block was alloc'd at
==9166==    at 0x483B7F3: malloc (in ...)
==9166==    by 0x10917E: func (main.c:6)
==9166==    by 0x1091AC: main (main.c:13)
```

Listing 5: Memcheck Ausgabe

Problem 2

```
5  int* pointer = malloc(sizeof(int));
6  if(*pointer == 0)
7  {
8      //code hier
9      //dies muss nicht immer passieren
10 }
```

Listing 6: uninitialisierter Wert

- `*pointer` ist uninitialisiert und kann somit alles sein
- richtiger wäre hier `calloc`

Fehlermeldung 2

```
5  int* pointer = malloc(sizeof(int));
6  if(*pointer == 0)
7  {
8      //code hier
9      //dies muss nicht immer passieren
10 }
```

Listing 7: uninitialisierter wert

```
==15468== Conditional jump or move depends on
    ↪ uninitialised value(s)
==15468==    at 0x1091AB: main (main.c:5)
```

Listing 8: Memcheck Ausgabe

Problem 3

```
4  void func()
5  {
6      int* pointer = malloc(sizeof(int));
7      //tu etwas mit dem pointer
8  }
```

Listing 9: memory leak

- belegt Speicher, gibt ihn aber nie frei
- irgendwann ist Speicher voll

Fehlermeldung 3

```
4 void func()
5 {
6     int* pointer = malloc(sizeof(int));
7     //tu etwas mit dem pointer
8 }
```

Listing 10: memory leak

```
==15712== 4 bytes in 1 blocks are definitely lost
    ↪ in loss record 1 of 1
==15712==    at 0x483B7F3: malloc (in ...)
==15712==    by 0x10915E: func (main.c:6)
==15712==    by 0x109181: main (main.c:12)
```

Listing 11: Memcheck Ausgabe

Technik

- V-Bits
- A-Bits

V-Bits [1]

- gibt an ob Bit initialisiert ist
- bei jeder Operation berechnet
- Überprüfung nur bei:
  - Berechnung von Speicheradresse
  - bedingten Sprunganweisungen
  - Systemcalls
- malloc/new $\rightarrow$ invalide
- calloc $\rightarrow$ valide
- realloc $\rightarrow$ zusätzliche Bits invalide

A-Bits [1]

- gibt an ob Byte allokiert ist
- bei Speicherzugriff überprüft
- malloc/new $\rightarrow$ valide
- free $\rightarrow$ invalide
- globale Daten $\rightarrow$ valide

Tricks [1]

- merkt sich Speicherblöcke
- freigeben nur mit richtige Funktion (C++)
- findet nicht freigegebene Blöcke
  - Still reachable (nicht angezeigt)
  - Definitely lost
  - Indirectly lost (nicht angezeigt)
  - Possibly lost

Memcheck Zusammenfassung

- findet viele Speichermanagementfehler
- diese Fehler sind schwer zu finden
- Fehler bei korrekten Programmen
- nicht sehr schnell

Cachegrind und Callgrind [1]

- Cachegrind: Cache- und Branch-Profiler
- Callgrind: basiert auf Cachegrind
- KCachegrind: GUI zur Ergebnissanzeige
  - nicht Teil von Valgrind
- cg_annotate bzw. callgrind_annotate

Cache [4]

- zwischen RAM und Registern
- sehr schnell
- wenig Speicher
- Level: RAM $\rightarrow$ L3 $\rightarrow$ L2 $\rightarrow$ L1 $\rightarrow$ Register
- Daten nicht im Cache: cache miss

Probleme

```
1 int i, j, a[1024][1024];
2 for(i = 0; i < 1024; i++){
3     for(j = 0; j < 1024; j++){
4         a[j][i] = 0;
5     }
6 }
```

Listing 12: 2D-Array Zugriff (langsam)[1]

```
1 int i, j, a[1024][1024];
2 for(i = 0; i < 1024; i++){
3     for(j = 0; j < 1024; j++){
4         a[i][j] = 0;
5     }
6 }
```

Listing 13: 2D-Array Zugriff (schnell)[2]

[1] angelehnt an [5]
[2] angelehnt an [5]

Cache in Cachegrind [1]

- 3 Arten
  - I1: Level 1 Befehlscache
  - D1: Level 1 Datencache
  - LL: "Last Level" Befehl + Daten Cache
- cache misses werden gezählt

```
==70303== D   refs:       9,049,437  (8,037,222 rd   + 1,012,215 wr)
==70303== D1  misses:     1,002,897  (    2,329 rd   + 1,000,568 wr)
==70303== LLd misses:        64,086  (    2,110 rd   +    61,976 wr)
==70303== D1  miss rate:       11.1% (      0.0%     +      98.8%  )
==70303== LLd miss rate:        0.7% (      0.0%     +       6.1%  )
```

Listing 14: Cachegrind Ausgabe (Daten)

Cache Simulation [1, 5]

- Cache wird simuliert
- 5 Befehlsarten:
  1. kein Speicher
  2. nur lesen
  3. nur schreiben
  4. lesen/schreiben (gleiche Adresse)
  5. lesen/schreiben (unterschiedliche Adressen)
- nach schreiben sind Daten in L1

cg_annotate

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

         .    .    .          .      .      .          .          .  void slow()
     4,104    0    0      1,025  1,024  1,023          2          1  {
         .    .    .          .      .      .          .          .      int i, j, a[1024][1024];
     3,076    1    1      2,049      0      0          1          0      for(i = 0; i < 1024; i++){
3,149,824    0    0  2,098,176      0      0      1,024          0          for(j = 0; j < 1024; j++){
7,340,032    0    0  2,097,152      0      0  1,048,576  1,048,562 64,420              a[j][i] = 0;
         .    .    .          .      .      .          .          .              }
         .    .    .          .      .      .          .          .      }
         6    0    0          4      1      0          0          0  }
```

Abbildung: cg_annotate langsamer Code

Callgrind Graph [1]

- speichern des Aufrufverlaufs (call stack)
- inklusive Cache Performance
  - Befehlsanzahl von Subfunktionen werden addiert

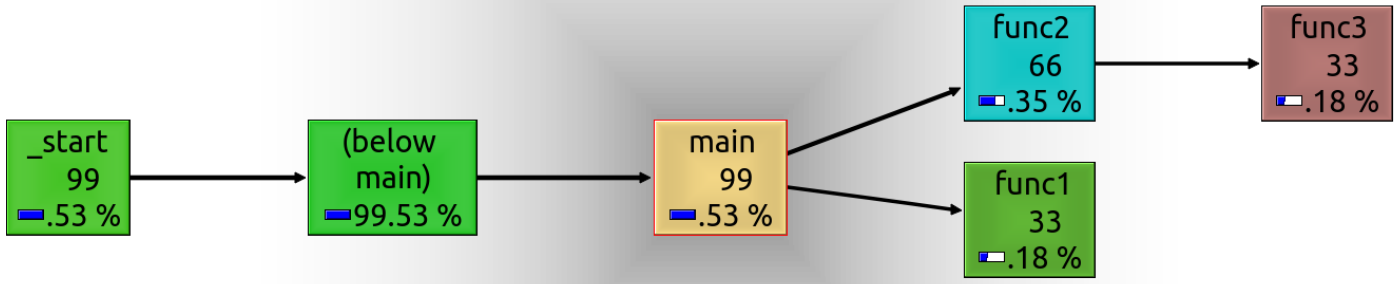

Abbildung: KCachegrind Call graph

Nutzer und Auszeichnungen [1, 6, 7, 8, 9, 10]

- In vielen Projekten genutzt
  - Firefox
  - OpenOffice
  - MySQL und SQLite
  - Gimp
- Auszeichnungen:
  - merit Open Source Award
  - Google-O'Reilly Open Source Award (Best Toolmaker)
  - TrollTech's Open Source Development Award

Zusammenfassung

- starkes und vielseitiges Analysewerkzeug
- kein rekompilieren nötig
- anwendbar auf fast jedes Programm
- Memcheck:
  - findet Speicherfehler
  - oft schwer findbar
- Cachegrind
  - findet langsamen Code
  - hilfreich für effiziente Programme
- Tools zeigen Fehler
  beheben sie aber nicht

Literatur I

[1] Valgrind Developers. Valgrind user manuel. https://valgrind.org/docs/manual/manual.html, 2021. Zugriff 6.6.2021.

[2] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[3] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):44–66, 2003. RV '2003, Run-time Verification (Satellite Workshop of CAV '03).

[4] Jim Handy. *The Cache Memory Book*. Seiten 89/90, Zugriff 12.6.2021.

Literatur II

[5] Nicholas Nethercote. Dynamic binary analysis and instrumentation. Technical Report UCAM-CL-TR-606, University of Cambridge, Computer Laboratory, November 2004.

[6] Debugging firefox with valgrind. `https://firefox-source-docs.mozilla.org/contributing/debugging/debugging_firefox_with_valgrind.html`. Zugriff 6.6.2021.

[7] Hacking:debug/test. `https://wiki.gimp.org/wiki/Hacking:Debug/Test`. Zugriff 6.6.2021.

Literatur III

[8] Jens-Heiner Rechtien. Validating and dedbugging openoffice.org with valgrind. http://www.kegel.com/openoffice/valgrindingOOo.html. Zugriff 6.6.2021.

[9] How sqlite is tested. https://www.sqlite.org/testing.html. Zugriff 6.6.2021.

[10] Tools that were used to create mysql. https://dev.mysql.com/doc/refman/8.0/en/tools-used-to-create-mysql.html. Zugriff 6.6.2021.