Optimierung von Programmen

Profiling

Christian Pape

Hochschule Karlsruhe

Wintersemester 2017/18

Inhalt

Begriffe

Profiler

to profile: Ermitteln von Informationen oder Werte.
Software-Profiler: charakteristische Werte eines Programms bestimmen.
Charakteristische Werte in der Informatik:

- Speicher- oder Zeitverbrauchs eines Programms
- Aufrufhäufigkeit von Anweisungen oder Funktionen
- Lebenszyklus von Objekten.
- *Aufrufgraph* (call graph) von Funktionen.

Begriffe

Aufrufgraph

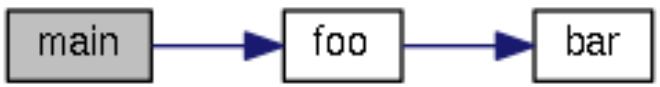
- Gerichteter Graph mit
  - Funktionen als Knoten und den
  - Funktionsaufrufe (aus Rumpf) als Kanten.
- Spezialfall **Aufrufbaum**: dynamisch, Aufrufe während des Programmablaufs. Keine Zyklen. Berechenbar.
- Aufrufgraph: statisch, Aufrufbeziehungen im Programm. Zyklen bei Rekursion.
- Berechnung statischer Aufrufbaum ist *unentscheidbar*, nur Approximationen möglich.

Begriffe

Aufrufgraph Beispiel

Schleife (main), Verzweigung (foo), direkte Rekursion (bar -> bar).

```
#include <iostream>
int bar(int a) {
  return a > 0 ? (bar(a - 1) + 1) : 1;
}
int foo(int a, int b) {
  return bar(a) + bar(b);
}
int main(void) {
  for (int i = 0; i < 5; i++) {
    std::cout << foo(i, 5) << std::endl;
  }
}
```



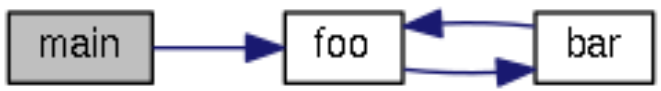
Aufrufbaum mit doxygen und graphiz erstellt.

Begriffe

Aufrufgraph Beispiel 2

Mit indirekte Rekursion (foo <-> bar).

```
#include <iostream>
int foo(int a, int b);
int bar(int a) {
  return a > 0 ? (foo(a - 1, a - 1) + 1) : 1;
}
int foo(int a, int b) {
  return bar(a) + bar(b);
}
int main(void) {
  for (int i = 0; i < 5; i++) {
    std::cout << foo(i, 5) << std::endl;
  }
}
```



Begriffe

Grundblock(graph)

Analog zu Aufrufgraph:

- Grundblock: Zusammengehörige Anweisungen ohne Verzweigungen.
- Grundblockgraph: Knoten Grundblöcke, Kanten Programmverzweigungen.

```
for (int i = 0; i < 5; i++) {
  std::cout << foo(i, 5) << std::endl;
}
```

Grundblöcke (vermutlich, letzter eventuell mehrere Grundblöcke):

1. **int** i = 0
2. i < 5
3. std::cout << foo(i, 5) << std::endl; i++

Anwendungszweck und Auswahl

- Einsatz meist zur Programmoptimierung
- Aber auch zur Fehlersuche, z.B. Speicherlecks
- Vor Einsatz:
  - Was für Werte sollen gemessen werden?
  - Genauigkeit der Messung?
- Bei Optimierung: kritische Programmabschnitte identifizieren, die die meisten Resourcen verbrauchen: den Brennpunkten (hot spots).

Anwendungszweck und Auswahl

Grobe Kategorisierung

- Funktionsweise:
  - Statisch: Quelltext des Programms wird analysiert.
  - Dynamisch: Ein Programmablauf wird analysiert mit konkreten Daten.
- Art des Programmcodes:
  - Quelltext: Der Quelltext muss vorhanden sein.
  - Binär: Kein Quelltext nötig.
- Art der Ausführung:
  - Maschinennah: Programm wird direkt auf CPU ausgeführt.
  - Interpretiert: CPU wird simuliert.

Im folgenden nur dynamische Analyse.

Anwendungszweck und Auswahl

Funktionsweise

Es gibt verschiedenen Methoden, wie ein Profiler zur Laufzeit Daten sammelt: [Fog, 2017, Abschnitt 3.2]

- *Instrumentierung*: Zusätzliche Anweisungen werden in das Programm eingefügt, um zum Beispiel zu zählen wie oft eine Funktion aufgerufen wurde.
- *Debugging*. Haltepunkte für den Debugger werden in das Programm eingefügt.
- *Zeitliche Abtastung*: Das Programm wird in zeitlichen Abständen unterbrochen.
- *Ereignis-basierte Abtastung*: Mit CPU-spezifische Befehlen wird das Programm bei bestimmten Ereignissen, z.B. Cache-misses, unterbrochen.

Werkzeuge

Übersicht

Für die Veranstaltung ausprobierte Profiler:

Tool	
gcov	Instrumentierung, Quelltext
gprof	Instrumentierung, Quelltext
Valgrind	Instrumentierung, Binärcode, CPU-Simulation
Linux Perf	Ereignis-basiert, Binärcode, ...
OProfile	Ereignis-basiert, Binärcode, ...
Pin	Instrumentierung, Binärcode, Ereignis-basiert, ...

Hauptsächlich für Unix oder GNU-Umgebung vorhanden.

gcov

- Teil der GNU Compiler Collection (GCC) [Stallman and Community, 2017].
- Zeilenabdeckung und Häufigkeit aufgerufener Programmzeilen.
- Compiler instrumentiert erzeugten Code.
- Übersetzte Programm erzeugt während Ausführung eine Logdatei mit Endung `.gcov`.
- `gcov` wird für einen Quelltext aufgerufen und erzeugt eine annotierte Quelltextdatei.

gcov

Beispiel sort.cc

Sortieren durch direktes Auswählen.

```
#include <algorithm>
#include <array>
#include <iostream>

template <class T, size_t SIZE>
void sort(std::array<T,SIZE> & a) {
  for (auto it = a.begin(); it != a.end() ; it++) {
    std::swap( *it, *std::min_element(it, a.end()) );
  }
}

int main(void) {
  std::array<int, 10000> a = {6, 5, 4, 3, 2, 1, 0, };
  sort(a);
  for (int x : a) {
    std::cout << x << " ";
  }
  std::cout << std::endl;
}
```

gcov

Beispiel

- `-ftest-coverage`: Compiler erzeugt Datei `sort.gcno` mit dem Grundblockgraphen.
- `-fprofile-arcs`: Ausgeführtes Programm erzeugt `sort.gcda`, (GNU Compiler arc data). Enthält Verzweigungen von einem Block zum anderen (arc transition counts), die Zählwerte für Messungen (value profile counts) und mehr.

```
> g++ -Wall -pedantic -ftest-coverage -fprofile-arcs sort.cc
> ./a.out >/dev/null
```

gcov

Beispiel

- Annotierte Quelltexte erstellen mit `gcov sort.cc`
- Dateiendung `.gocv`.
- Auch für Header-Dateien, wenn sie Definitionen enthalten (Templates)
- Sie enthalten die Ausführungshäufigkeit jeder Quelltextzeile.

```
> gcov sort.cc
File 'sort.cc'
Lines executed:100.00% of 11
Creating 'sort.cc.gcov'
(weitere Ausgaben)
```

gcov

sort.cc.gcov

```
        -:    0:Source:sort.cc
        -:    0:Graph:sort.gcno
        -:    0:Data:sort.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:// g++ -Wall -pedantic sort.cc -o sort
        -:    2:// für gcov
        -:    3://
        -:    4:#include <algorithm>
        -:    5:#include <array>
        -:    6:#include <iostream>
        -:    7:
        -:    8:template <class T, size_t SIZE>
        1:    9:void sort(std::array<T,SIZE> & a) {
    10001:   10:  for (auto it = a.begin(); it != a.end(); it++) {
    10000:   11:    std::swap( *it, *std::min_element(it, a.end()) );
        -:   12:  }
        1:   13:}
        -:   14:
        1:   15:int main(void) {
        1:   16:  std::array<int, 10000> a = {6, 5, 4, 3, 2, 1, 0, };
        1:   17:  sort(a);
    10001:   18:  for (int x : a) {
    10000:   19:    std::cout << x << " ";
        -:   20:  }
        1:   21:  std::cout << std::endl;
        4:   22:}
```

gcov

- Direktes Auswählenhat quadratischen Aufwand (Anzahl Vergleiche)
- Vergleiche innerhalb Minmumsuche mit Iteratoren (`stl_algo.h`)
- grep Ausgabe filtern: nur Zeilen die :ënthalten.

```
> grep -v "\-:" stl_algo.h.gcov
    10000: 5453:     __min_element(_ForwardIterator __first, _ForwardIterator __last,
    10000: 5456:       if (__first == __last)
    #####: 5457: return __first;
    10000: 5458:       _ForwardIterator __result = __first;
100000000: 5459:       while (++__first != __last)
 49995000: 5460: if (__comp(__first, __result))
    34988: 5461:   __result = __first;
    10000: 5462:       return __result;
    10000: 5475:     inline min_element(_ForwardIterator __first, _ForwardIterator __last)
    10000: 5485: __gnu_cxx::__ops::__iter_less_iter());
```

gprof

- Teil der GNU Binutils. [Graham et al., 1982]
- Instrumentierung des Codes: Zusätzlich Anweisungen, um die Aufrufe der Funktion selbst und die von hier aufgerufenen Funktionen zu protokollieren.
- Stichprobenartige Auswertung des Programmzählers, um die Ausführungszeit von Funktionen zu bestimmen.
- Dies Verfahren liefert keine genau Zeitmessung!

gprof

Analyse in drei verschiedenen Formen:

- *Flaches Profil* (flat profile): Gibt an, wie viel Zeit das Programm innerhalb einer Funktion verbraucht hat und wie oft eine Funktion aufgerufen wurde.
- *Aufrufgraph* (call graph): Gibt für jede Funktion an, wie oft, von wo sie aufgerufen wurde und welche Funktionen sie selbst aufruft.
- *Annotierter Quelltext* (annonated source): Der Quelltext mit angehefteter Aufrufhäufigkeit zu jeder Zeile.

gprof

- Mit Compiler-Option `-pg` übersetzen.
- Workaround eines Bugs bei manchen GCC Verisonen: `-no-pie`
- Programm wird normal ausgeführen.
- Datei `gmon.out` mit den gesammelten Daten wird erzeugt.
- Ein Aufruf `gprof` liest diese Datei ein und gibt ein Analyse als Textausgabe aus (nur ersten Zeilen unten).

```
> g++ -pg -Wall -pedantic sort.cc
> ./a.out >/dev/null
> gprof
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 45.45      0.05     0.05 49995000     0.00     0.00  bool __gnu_cxx::__ops::_It
er_less_iter::operator()<int*, int*>(int*, int*) const
 36.36      0.09     0.04    10000     0.00     0.01  int* std::__min_element<in
t*, __gnu_cxx::__ops::_Iter_less_iter>(int*, int*, __gnu_cxx::__ops::_Iter_less_
iter)
```

Valgrind

- `Valgrind` [Nethercote and Seward, 2007] ist eine Toolbox, um dynamische Programmanalyse-Werkzeuge zu erstellen.
- Lauffähige Version, um Speicherzugriffsfehler zu finden oder Profile für Cache-Zugriffe und Sprungvorhersagen zu erstellen existiert.
- Primär für x86- und AMD64-Rechnerarchitekturen auf Unix-Betriebssystemen wie Linux und Mac OS.
- Teilweise unvollständige Portierungen für andere Plattformen wie FreeBSD oder Solaris.
- In Programmiersprache C geschrieben.
- Instrumentierung ausführbarer *binärer* Programme dynamisch zur Laufzeit (dynamic analysis instrumentation)

Valgrind

- Binärcode wird nicht direkt ausgeführt.
- Valgrind lädt das Binärprogramm in seinen Prozessraum und übersetzt es blockweise, just-in-time in ein Zwischenformat (intermediate representation, IR).
- Zur Analyse wird Code für Schattenwerte für die originalen Resourcen, wie Register oder Cachespeicher, geführt und Anweisungen zur Analyse eingefügt.
- Funktioniert nicht bei selbst-modifizierenden Code.
- Er verlangsamt auch die Ausführung erheblich.

Valgrind

Basisversion

- *Memcheck*. Sucht nach Speicherzugriffsfehler.
- *Cachegrind*. Misst Speicher- und Cache-Zugriffe sowie Sprungvorhersagen.
- *Callgrind*. Misst wie Callgrind Speicherzugriff und ordnet sie mit Hilfe des Aufrufgraphen einzelnen Funktionen zu.
- *Hellgrind*. Sucht nach Fehlern in Threads bei nebenläufigen Programmen.
- *DRD*. Sucht wie Hellgrind nach Fehlern in Threads, verwendet dabei allerdings alternative Methoden.
- *Massif*. Misst den Verbrauch des dynamischen Speichers (heap).

Valgrind

Auruf

`Valgrind` wird mit folgendem Kommando gestartet:

```
valgrind [options] --tool=<name> prog-and-args
```

- `<name>` ist der Name des ausgewählten Werkzeug.
- Falls nichts angegeben ist, wird `memcheck` verwendet.
- `options` sind zusätzliche Optionen für `Valgrind`.
- `prog-and-args` ist der Programmaufruf mit allen Optionen des zu prüfenden Programms.
- Die zu prüfenden Programme sollen mit Debug-Informationen übersetzt sein.

Valgrind

Beispiel memcheck

```
#include <iostream>

class EvilMemory {
  int y;
public:
  int & func(int *a) {
    int x = *a;

    *(a+1) = y; // (1) y nicht initialisiert
                // (2) a+1 zeigt auf ungueltigen Speicher
    return x;   // (3) Referenz auf lokales Objekt
  }
};

int main(void) {
  EvilMemory *evil = new EvilMemory(); // (4) memory leak
  int z = 1;

  z = evil->func(&z);
  std::cout << z << std::endl;

  return 1;
}
```

Valgrind

Valgrind memcheck

```
> g++ -g -Wall -pedantic memory.cc
memory.cc: In member function ‘int& EvilMemory::func(int*)’:
memory.cc:7:9: warning: reference to local variable ‘x’ returned [-Wreturn-local-addr]
     int x = *a;
         ^
> valgrind ./a.out >/dev/null
Memcheck, a memory error detector
==6046== Copyright (C) 2002-2015, and GNU GPL’d, by Julian Seward et al.
==6046== Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for copyright info
==6046== Command: ./a.out
==6046==
==6046== Invalid read of size 4
==6046==    at 0x1087D5: main (memory.cc:20)
==6046==  Address 0x0 is not stack’d, malloc’d or (recently) free’d
==6046==
==6046== Process terminating with default action of signal 11 (SIGSEGV)
==6046==  Access not within mapped region at address 0x0
==6046==    at 0x1087D5: main (memory.cc:20)
```

- Anderen drei Fehler: nichts.
- Programm wird (simuliert) ausgeführt und der Speicherfehler führt zum "Absturz".
- Fehler beheben, dann weitere Fehler suchen.

Literaturverzeichnis


Fog, A. (2017).
Optimizing software in C++. An optimization guide for Windows Linux and Mac platforms.
http://www.agner.org/optimize/optimizing_cpp.pdf.


Graham, S. L., Kessler, P. B., and Mckusick, M. K. (1982).
Gprof: A call graph execution profiler.
SIGPLAN Not., 17(6):120–126.


Nethercote, N. and Seward, J. (2007).
Valgrind: A framework for heavyweight dynamic binary instrumentation.
SIGPLAN Not., 42(6):89–100.


Stallman, R. M. and Community, G. D. (2017).
Using The Gnu Compiler Collection. For GCC version 7.2.0.
GNU Press, Boston, MA, USA.