von Arnout Engelen
<arnouten(Q)bzzt.net>

Über den Autor:

Arnout Engelen studiert Informatik an der Universität von Nijmegen (Niederlande) und ist bei TUNIX beschäftigt, einer Internet−Sicherheitsfirma. In seiner Freizeit läuft er lange Strecken und spielt Tenorsaxophon.

C/C++−Programme optimieren mit dem Profiler gprof

Zusammenfassung:

Eine der wichtigsten Sachen, die man im Kopf behalten muß, wenn man eine Applikation optimiert, ist: optimiere dort, wo es zählt. Es nützt nichts, stundenlang ein Stück Code zu optimieren, das sowieso nur während 0,04 Sekunden läuft.

gprof bietet einen überraschend einfachen Weg, deine C/C++−Applikation zu profilieren und die interessanten Stellen gleich zu finden. Eine kleine Fallstudie zeigt, wie mittels gprof die Laufzeit einer realen Applikation von über 3 Minuten auf unter 5 Sekunden reduziert wurde, indem zwei Datenstrukturen als wichtig erkannt und optimiert wurden.

Historisch gesehen reicht das Programm bis 1982 zurück, als es auf dem SIGPLAN−Symposion über Compilerbau vorgestellt wurde. Es ist inzwischen ein Standardwerkzeug, das es auf praktisch allen UNIX−Arten gibt.

Übersetzt ins Deutsche von:
Viktor Horvath
<ViktorHorvath(at)gmx.net>

Profiling kurz und bündig

Das Konzept des Profiling ist sehr einfach: Indem man festhält, zu welchen Zeiten ein Programm Funktionen betritt und verläßt, ist es möglich zu berechnen, in welchen Teilen des Programms es sich die meiste Zeit aufhält. Die Durchführung dieser Messung klingt jetzt nach etwas, das viel Mühe kostet – glücklicherweise ist dem nicht so! Man muß lediglich mit einer zusätzlichen gcc−Option kompilieren (`-pg`), das Programm laufen lassen (um die Daten für das Profiling zu erzeugen) und `gprof` mit der erzeugten Statistikdatei aufrufen, um sie in einer komfortableren Art darzustellen.

Fallstudie: Pathalizer

Ich benutze hier eine reale Applikation als Beispiel, einen Teil von pathalizer: das Programm `event2dot`, das eine „events“–Datei von Pathalizer in eine „dot“–Datei von graphviz übersetzt.

Kurz gesagt, liest es die Ereignisse aus einer Datei und speichert sie als Graphen (mit den Seiten als Knoten und mit den Übergängen zwischen Seiten als Kanten). Diese Graphensammlung wird dann in einen großen Graph im graphviz „dot“–Format „zusammengefaßt“.

Zeitmessung der Applikation

Zuerst starten wir das Programm, das wir optimieren wollen, ohne Profiling und messen, wie lange es braucht. Die Programmquellen für das Beispiel sind ebenso wie Beispieldaten von beträchtlicher Größe (55.000 Zeilen) verfügbar.

Auf meinem Rechner dauerte ein Lauf von `event2dot` mehr als drei Minuten auf diesen Daten:

```
real    3m36.316s
user    0m55.590s
sys     0m1.070s
```

Das Profiling

Profiling mit gprof wird eingeschaltet, indem man die Option `-pg` zum Kompiliervorgang hinzufügt. Wir kompilieren die Applikation also:

```
g++ -pg dotgen.cpp readfile.cpp main.cpp graph.cpp config.cpp -o event2dot
```

Wir können jetzt `event2dot` abermals auf unserer Testdatei rechnen lassen. Währenddessen wird Profiling–Information zu `event2dot` gesammelt und eine Datei „gmon.out“ generiert. Wir sehen uns das Ergebnis mittels `gprof event2dot | less` an.

gprof zeigt uns jetzt die wichtigsten Funktionen:

```
 % cumulative  self              self     total
 time seconds  seconds  calls s/call s/call name
43.32   46.03  46.03 339952989  0.00  0.00 CompareNodes(Node *,Node *)
25.06   72.66  26.63    55000   0.00  0.00 getNode(char *,NodeListNode *&)
16.80   90.51  17.85 339433374  0.00  0.00 CompareEdges(Edge *,AnnotatedEdge *)
12.70  104.01  13.50    51987   0.00  0.00 addAnnotatedEdge(AnnotatedGraph *,Edge *)
 1.98  106.11   2.10    51987   0.00  0.00 addEdge(Graph *,Node *,Node *)
 0.07  106.18   0.07        1   0.07  0.07 FindTreshold(AnnotatedEdge *,int)
 0.06  106.24   0.06        1   0.06 28.79 getGraphFromFile(char *,NodeListNode *&,Config *)
 0.02  106.26   0.02        1   0.02 77.40 summarize(GraphListNode *,Config *)
 0.00  106.26   0.00    55000   0.00  0.00 FixName(char *)
```

Die interessanteste Spalte ist die erste: Das ist der prozentuale Anteil dieser Funktion an der gesamten Programmlaufzeit.

Die Optimierung

Das Programm verbringt demnach fast die Hälfte seiner Zeit in `CompareNodes`. Ein schnelles `grep` zeigt, daß `CompareNodes` nur von `CompareEdges` aufgerufen wird, welches wiederum nur von `addAnnotatedEge` benutzt wird – diese beiden befinden sich auch in der Liste. Das sieht nach einer interessanten Stelle zur Optimierung aus.

Wir stellen fest, daß `addAnnotatedEdge` eine verlinkte Liste durchläuft. Obwohl eine verlinkte Liste einfach zu implementieren ist, ist sie nicht die beste aller Datenstrukturen. Wir entscheiden, `g->edges` durch einen Binärbaum zu ersetzen: Das sollte die Suche in der Struktur stark beschleunigen und trotzdem einen Durchgang ermöglichen.

Ergebnisse

Wir sehen, daß die Ausführungszeit tatsächlich reduziert wird:

```
real    2m19.314s
user    0m36.370s
sys     0m0.940s
```

Ein zweiter Lauf

Der nochmalige Lauf von gprof offenbart:

```
%   cumulative self              self    total
 time   seconds seconds calls  s/call  s/call name
87.01     25.25  25.25  55000    0.00    0.00 getNode(char *,NodeListNode *&)
10.65     28.34   3.09  51987    0.00    0.00 addEdge(Graph *,Node *,Node *)
```

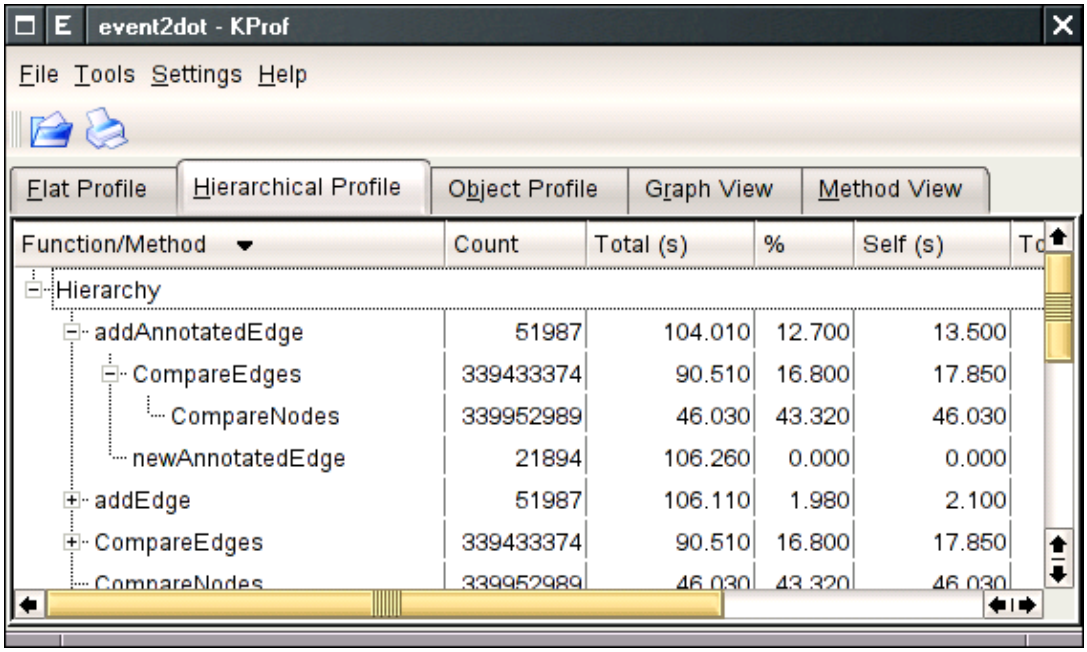
Eine Funktion, die bislang über die Hälfte der Zeit verbraucht hat, wurde bis in die Irrelevanz reduziert! Das wollen wir noch einmal versuchen: Wir ersetzen die `NodeList` durch eine `NodeHashTable`.

Auch das ist eindeutig eine große Verbesserung:

```
real    0m3.269s
user    0m0.830s
sys     0m0.090s
```

Andere Profiler für C/C++

Es gibt einige Profiler, die die Daten von gprof benutzen, z.B. KProf (Screenshot) und cgprof. Obwohl die graphischen Ansichten eine nette Sache sind, finde ich persönlich gprof auf der Kommandozeile praktischer.



Profiling von anderen Sprachen

Wir haben nun das Profiling von C/C++–Applikationen mit gprof besprochen, aber Ähnliches kann auch mit anderen Sprachen gemacht werden. Für Perl kannst du das Modul Devel::DProf benutzen. Starte deine Applikation mit `perl -d:DProf mycode.pl` und sieh die Ergebnisse mit `dprofpp` an. Wenn du deine Java–Programme mit gcj kompilieren kannst, kannst du einfach gprof benutzen, es wird zur Zeit jedoch nur Java–Code mit einem einzigen Thread unterstützt.

Fazit

Wir haben gesehen, daß wir durch Profiling schnell die Teile einer Applikation finden können, die von einer Optimierung profitieren würden. Indem wir dort optimierten, wo es drauf ankam, haben wir die Laufzeit des Beispielprogramms von 3:36 Minuten auf weniger als fünf Sekunden gesenkt.

Links

- Pathalizer: http://pathalizer.sf.net
- KProf: http://kprof.sf.net
- cgprof: http://mvertes.free.fr
- Devel::DProf http://www.perldoc.com/perl5.8.0/lib/Devel/DProf.html

- gcj: http://gcc.gnu.org/java
- Pathalizer−Beispieldaten: Download für article371

Der LinuxFocus Redaktion schreiben © Arnout Engelen "some rights reserved" see linuxfocus.org/license/ http://www.LinuxFocus.org	Autoren und Übersetzer: en −−> −− : Arnout Engelen <arnouten(Q)bzzt.net> en −−> de: Viktor Horvath <ViktorHorvath(at)gmx.net>

2005−07−30, generated by lfparser_pdf version 2.51