Optimiser les programmes C/C+ + en utilisant GProf profiler

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

L´auteur:

Arnout Engelen est étudiant en informatique à l'université de Nijmegen, Pays−Bas et est employé chez TUNIX, une entreprise spécialisée dans la sécurité Internet. Durant son temps libre, il aime courir et jouer du saxophone.

Résumé:

Une des choses les plus importantes qu'il est bon de garder en tête lorsque l'on souhaite optimiser une application est : optimiser là où c'est utile. Cela ne sert à rien de passer des heures à optimiser un morceau de code qui met 0.04 secondes à s'exécuter.

GProf fournit une façon étonnament simple de profiler (étudier le temps d'exécution) vos applications C/C++ et met en lumière les parties intéressantes très rapidement. Une courte étude de cas montre comment GProf fut utilisé pour réduire la durée d'exécution d'une application. En identifiant 2 structures de données importantes et en les optimisant, nous sommes passés de 3 minutes à moins de 5 secondes.

Historiquement, le programme remonte à 1982, quand il fut introduit dans le Symposium SIGPLAN sur la construction de compilateurs. C'est maintenant un outil standard disponible sur pratiquement toutes les formes d'UNIX.

Traduit en Français par:
Florent Morel <fleuh−(at)−free.fr>

Le Profiling en quelques mots

Le concept du profiling est vraiment très simple : en enregistrant l'heure d'entrée et de sortie de chaque fonction, il est possible de calculer quelles sont les parties du code qui prennent le plus de temps. Effectuer ces mesures semble être une tâche nécessitant beaucoup d'efforts − heureusement, ce n'est pas le cas. C'est si simple qu'il suffit de compiler en rajoutant une option gcc ('−pg'), lancer le programme (pour collecter les informations de profiling) et de lancer 'gprof' sur le fichier contenant les statistiques d'exécution afin de les présenter d'une manière plus agréable.

Etude de cas : Pathalizer

Je vais prendre le cas d'une réelle application, qui fait partie de pathalizer : l'exécutable `event2dot` qui traduit un fichier 'évènements' de pathalizer en un fichier contenant un graphique sous forme de points (format graphviz).

Pour résumer, il lit les évènements depuis un fichier, les stocke sous forme de graphes (les pages sont transposées en noeuds et les transitions entre pages en bords). Cette collection de graphes est ensuite 'condensée' en un gros graphe qui est imprimé sous forme de points au format graphviz.

Mesurer le temps d'exécution de l'application

En premier lieu, nous lançons le programme que nous voulons optimiser sans profiling et mesurons le temps d'exécution. Les sources utilisées sont fournies ainsi qu'un fichier exemple d'entrée d'une taille considérable (55 000 lignes).

Sur ma machine, l'exécution de `event2dot` met plus de 3 minutes avec ce fichier :

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

Le profiling

Pour permettre le profiling, il suffit d'ajouter l'option '–pg' lors de la compilation. Recompilons donc l'application avec cette option :

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

Nous pouvons maintenant lancer `event2dot` à nouveau sur notre fichier « test–events ». Durant cette exécution, les informations de profiling concernant `event2dot` vont être récupérées et un fichier 'gmon.out' sera généré. Visualisons les résultats en lançant 'gprof `event2dot` | less'.

gprof montre que les fonctions suivantes sont importantes :

```
 % 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 *)
```

La colonne la plus intéressante est la première : il s'agit du pourcentage de temps d'exécution du programme pris par cette fonction.

L'optimisation

Cela montre que le programme passe presque la moitié du temps dans la fonction `CompareNodes`. Un rapide grep montre que CompareNodes n'est appelée que par `CompareEdges`, qui elle–même n'est appelée que par `addAnnotatedEdge` – ces deux fonctions étant aussi dans la liste. Cela semble donc être un bon point de départ pour réaliser l'optimisation.

Nous avons remarqué que `addAnnotatedEdge` traverse une liste chaînée. Bien que facile à implémenter, une liste chaînée n'est pas le meilleur type de données. Nous décidons de remplacer g–>edges par un arbre binaire : cela devait permettre d'accélérer sensiblement les recherches, en laissant la possibilité de le parcourir.

Résultats

Nous pouvons remarquer la réduction du temps d'exécution :

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

Second passage

Lancer gprof à nouveau révèle :

```
%   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 *)
```

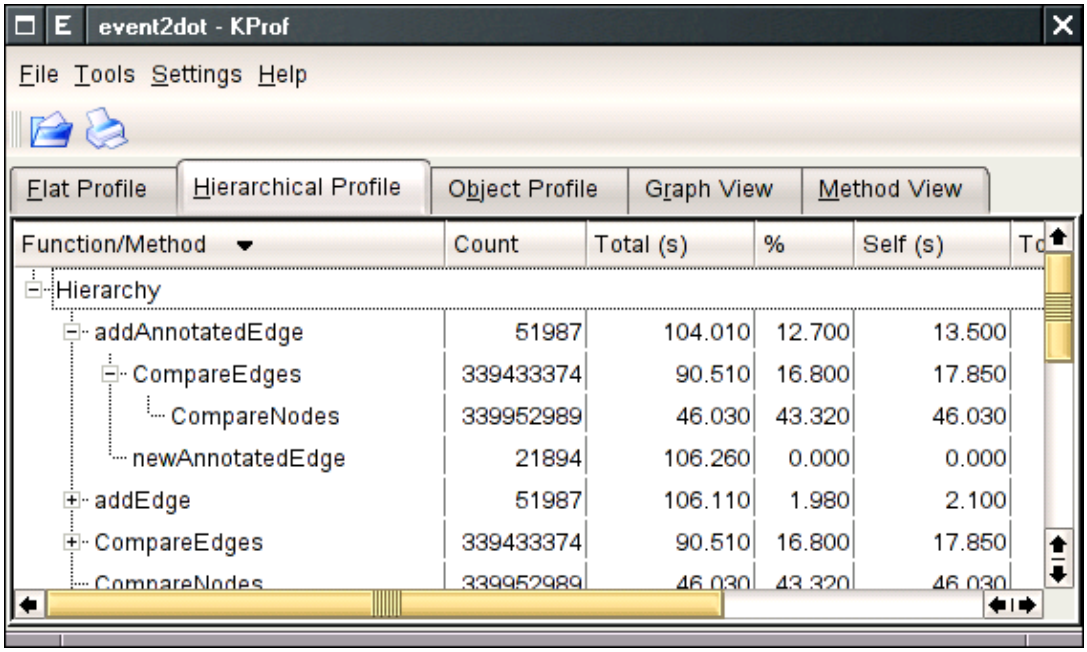
Il semble que les fonctions qui prenaient plus de la moitié du temps ont maintenant été réduites à une durée négligeable ! Essayons à nouveau : remplaçons NodeList par une NodeHashTable.

C'est aussi une réelle amélioration :

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

Autres profilers C/C++

Plusieurs profilers disponibles utilisent les données de gprof, c'est le cas de KProf (capture d'écran) et de cgprof. Bien que l'interface graphique soit agréable, je pense que la ligne de commande de gprof est plus efficace.



Profiling avec d'autres langages

Ce tutoriel présente le profiling d'applications C/C++ avec gprof, mais il est possible de le faire pour d'autres langages : pour Perl, utilisez le module Devel::DProf. Lancez l'application avec `perl -d:DProf mycode.pl` et visualsez les résultats avec `dprofpp`. Si vous pouvez compiler vos programmes Java avec gcj, vous pouvez utiliser gprof, bien que l'utilisation soit limitée à une seule thread.

Conclusion

Nous avons vu que, en utilisant le profiling, il est possible de trouver des portions d'une application pouvant bénéficier d'une optimisation. En optimisant là où c'est utile, nous avons réduit le temps d'exécution de l'application testée de 3min36s à moins de 5 secondes.

Références

- 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
- Fichiers d'exemples de pathalizer : download for article371

Translation information:
en −−> −− : Arnout Engelen <arnouten(Q)bzzt.net>
en −−> fr: Florent Morel <fleuh−(at)−free.fr>

2005−09−15, generated by lfparser_pdf version 2.51