Boîte à outils du programmeur : profiler des programmes avec gprof

Linux Gazette n°100 — Mars 2004

Vinayak Hegde

Table des matières

1. Introduction à la série

Linux (et les autres Unix) fournit beaucoup de petits utilitaires pratiques que l’on peut combiner pour faire des choses intéressantes. Créer ces logiciels ou les utiliser pour peaufiner vos programmes procure une certaine joie. Dans cette série, nous allons examiner certains de ces outils utiles pour les programmeurs. Ces outils vous aideront à mieux coder et vous faciliteront la vie.

2. Qu’est-ce que le profilage ? Pourquoi en avez-vous besoin ?

Après avoir conçu et codé un programme, vient le stade de son optimisation. Avant de parler de profilage et d’optimisation en général, j’aimerais attirer votre attention sur deux citations à propos de l’optimisation :

- « Il y a plus de péchés informatiques commis au nom de l’efficacité (sans pour autant l’atteindre) qu’au nom de toute autre raison — y compris la stupidité aveugle. » (William A. Wulf)

- « Il faut oublier l'efficacité pour disons 97% du temps : l'optimisation prématurée est à la source de tous les maux. » (Donald E. Knuth)

La plupart des programmes suivent en gros la règle connue sous le nom de 80:20 : vous exécutez 20% du code pendant 80% du temps. Comme il est supposé par les citations ci-dessus, le temps du programmeur a plus de valeur que le temps machine. Nous avons donc vu l'avènement de langages comme Java et C#, qui réduisent le temps nécessaire à la programmation et donnent aux programmeurs plus de temps à consacrer à la logique qu'aux subtilités de l'architecture de la machine sous-jacente. Ceci a augmenté le temps d'exécution des programmes mais économisé du temps de programmation. Cependant, nous devons optimiser pour accélérer l'exécution d'un programme. Souvent, les compilateurs le font automatiquement. Par exemple, le compilateur GCC utilise les drapeaux `-O` (remarquez la majuscule) pour spécifier le niveau d'optimisation. Le profilage est une méthode qui peut nous aider à trouver les sections du code/des fonctions que nous devons optimiser pour améliorer la performance d'un programme. Vous conviendrez qu'il est beaucoup plus intelligent d'optimiser une fonction qui est appelée des milliers de fois quand le programme s'exécute qu'une fonction appelée une dizaine de fois dans un programme. Quand nous profilons un programme, nous arrivons à savoir quelles sont les parties du code fréquemment utilisées et quelles fonctions prennent le plus de temps processeur. Ces deux types de codes sont de bons candidats pour l'optimisation. Comme les données sont collectées en utilisant une trace d'exécution réelle, cette méthode est également efficace pour trouver des bogues cachés. Du fait que vous ne pouvez pas prévoir qu'une fonction donnée sera appelée 1 000 fois pendant l'exécution, il peut s'agir d'un défaut de conception et d'un bogue potentiel. C'est presque aussi utile que les revues de code dans les projets étendus et complexes.

On peut obtenir principalement deux types d'informations de profilage :

- Le profil plat (Flat Profile)

  Le profil plat détaille combien de temps processeur dure chaque fonction et combien de fois elle a été appelée. C'est un bref résumé des informations de profilage rassemblées. Ceci vous donnera une idée des fonctions qui peuvent être réécrites ou affinées pour gagner en performance.

- Le graphe d'appels (Call Graph)

  Le graphe d'appels montre pour chaque fonction du code le nombre de fois où elle a été appelée par différentes fonctions, y compris elle-même. Ceci peut suggérer quels sont les appels de fonctions à éliminer ou à remplacer par d'autres fonctions plus efficaces. Ces informations montrent les relations existant entre les différentes fonctions et peuvent être utilisées pour découvrir des bogues dans le code. De plus, vous pouvez optimiser certains chemins du code après avoir examiné les graphes d'appels.

3. Comment obtenir des informations de profilage ?

Le code source doit être compilé avec l'option `-pg` (ainsi qu'avec `-g` si vous souhaitez un profilage ligne par ligne). Si le nombre de lignes dans le fichier `Make` est réduit, vous pouvez ajouter ces options à la fin de chaque commande de compilation. Mais si le nombre de commandes de compilation est élevé, vous pouvez définir/redéfinir le paramètre *`CFLAGS`*/*`CXXFLAGS`* dans le `Makefile` et l'ajouter à toutes les commandes de compilation du `Makefile`. Voici un exemple d'utilisation de gprof avec l'utilitaire GNU make :

Nous pouvons utiliser ce make pour compiler d'autres logiciels comme Apache, lynx et cvs. À titre d'exemple, construisons apache à l'aide de ce make. Lorsque nous décompressons, configurons et lançons make sur le source d'Apache, un fichier nommé `gmon.out` contenant les informations de profilage est généré. Vous pouvez constater que make s'exécute plus lentement que prévu car il enregistre les données de profilage. Il est important de se rappeler que, lors de la collecte des données de profilage, il faut exécuter le programme en lui donnant les entrées que nous lui donnons normalement et en le terminant lorsque tout est terminé. De cette manière, vous aurez simulé un scénario réel pour collecter les données.

4. Analyser la sortie du profilage

Lors de la dernière étape, nous avons obtenu une sortie exécutable appelée `gmon.out`. Malheureusement, il n'existe pour le moment aucun moyen de spécifier le nom du fichier des données de profilage. Ce fichier `gmon.out` peut être interprété par gprof pour générer une sortie humainement lisible. La syntaxe correspondante est :

Vous trouverez le fichier complet ici (http://linuxgazette.net/100/misc/vinayak/profile-make-with-Apache.txt). Voici un extrait du profil plat :

Nous pouvons tirer les conclusions suivantes des données ci-dessus :

1. Trois fonctions (`file_hash_2`, `new_pattern_rule` et `pattern_search`) représentent presque tout le temps d'exécution.
2. Il y a six appels de fonction à `patter_search`, mais il faut compter une moyenne de 2.81 millisecondes pour chaque appel.

Les informations collectées sont néanmoins insuffisantes. Ce make compilé spécialement a donc été utilisé pour compiler lynx, cvs, make et patch. Tous les fichiers `gmon.out` renommés ont été rassemblés et les données de profilage ont été compilées à l'aide des commandes suivantes :

Vous trouverez ce fichier ici (http://linuxgazette.net/100/misc/vinayak/overall-profile.txt).

Voici un extrait du profil plat :

Comme nous pouvons le voir, la représentation a quelque peu changé depuis le profil de make que nous avons obtenu en compilant Apache.

1. Il y a 23 480 appels à la fonction `find_char_unquote` et son exécution prend plus de 1/6e du temps total d'exécution du programme.
2. La fonction `eval` n'est invoquée que 37 fois, mais elle prend tout de même 1/11e du temps d'exécution du programme. Il est possible que cette fonction fasse beaucoup de choses et soit un bon candidat à un découpage en plusieurs fonctions. Il est également à noter que chaque appel à `eval` prend 4.76 millisecondes, ce qui est assez énorme comparativement aux autres fonctions.
3. Les fonctions `pattern_search` et `update_file_1` prennent presque 1/4 du temps d'exécution mais ne totalisent que 268 appels. Ces fonctions peuvent peut-être aussi être divisées en de plus petites fonctions.

Jetons maintenant un coup d'½il au graphe d'appels de la compilation d'Apache.

Nous pouvons faire les observations suivantes de l'extrait ci-dessus :

1. La première colonne est une référence à l'index donné à la fin de la sortie de gprof.
2. La deuxième colonne donne le temps total passé dans la fonction `eval` y compris les appels à d'autres fonctions.
3. La troisième et la quatrième colonne donnent le temps total passé dans la fonction elle-même et dans ses appels aux autres fonctions.
4. Le premier nombre de la cinquième colonne donne le nombre d'appels à la fonction depuis `eval`, et le second nombre donne le nombre total d'appels non récursifs à cette fonction par tous ses appelants.
5. S'il y a des appels récursifs à la fonction elle-même ou à une fonction mutuellement récursive, alors le nom de la fonction est ajouté à l'ensemble (comme pour `eval_makefile` et `eval` ci-dessus).
6. Certaines fonctions sont toujours appelées par `eval`. On peut y trouver un avantage dans certains cas si le temps dédié à l'appel de la fonction lui-même peut être éliminé.

5. Autres fonctionnalités de gprof

L'utilisation de gprof vous permet aussi d'obtenir un listing annoté et un profilage ligne par ligne. Ceci peut être utile une fois que vous avez identifié les sections de code qui doivent être optimisées. Ces options vous aideront à creuser le code source pour y trouver les insuffisances. Le profilage ligne par ligne conjugué au profil plat peut être utilisé pour vérifier quels sont les chemins du code fréquemment exécutés. Le listing annoté peut être utilisé pour creuser les appels de fonctions eux-mêmes jusqu'au bloc de base (boucles et instructions de branchement) pour découvrir quelles boucles sont le plus exécutées et quelles branches sont prises le plus fréquemment. Ceci est utile pour affiner le code pour une performance optimale. Il existe d'autres options qui sortent du cadre de cet article. Reportez-vous à la documentation de gprof pour plus de détails. Il existe une interface KDE à gprof appelée kprof. Reportez-vous à la section Ressources pour connaître l'URL.

6. Conclusion

Les outils de profilage comme gprof peuvent être d'une grande aide dans l'optimisation des programmes. Le profilage est une des premières étapes de l'optimisation manuelle pour trouver les goulots d'étranglement et les supprimer.

7. Ressources

- La page GNU Info de gprof
- L'interface KDE à gprof KProf (http://kprof.sourceforge.net)

- Function Check — un autre outil de profilage. Il permet de contourner certains défauts de gprof.

Vinayak suit en ce moment un cursus d'APGDST au NCST. Il s'intéresse aux réseaux, aux systèmes de calcul parallèles et aux langages de programmation. Il pense que Linux aura le même effet sur l'industrie du logiciel que celui que l'imprimerie a eu sur le monde de la science et de la littérature. Pendant ses loisirs inexistants, il écoute de la musique et lit des livres. Il travaille actuellement au projet LIberatioN-UX pour lequel il rend accessible des solutions Linux peu onéreuses pour les milieux scolaires ou d'entreprise en configurant des stations amorçables à distance (clients légers).

Paru dans le n°100 de la Linux Gazette de mars 2004.

Traduction française par Isabelle Hurbain. `<isabelle POINT hurbain CHEZ pasithee POINT net>`.

Relecture de la traduction française par Joëlle Cornavin `<jcornavi CHEZ club TIRET internet POINT fr>`.