Environnement de Programmation et Programmation Système

Gilles Menez

Université de Nice – Sophia-Antipolis
Département d'Informatique
email : menez@unice.fr
www : www.i3s.unice.fr/~menez

23 septembre 2009: V 1.1

Table des Matières (1)

1 Références

2 Qualité du logiciel
- De quel point de vue ?
- Qualités pour le développeur
- Qualités pour l'utilisateur
- Objectifs de qualité

3 Analyse de performances
- Analyse de performances
- Préambule algorithmique
- Motivations
- Cycle d'optimisation
- Que mesure t'on ?
- Méthodes de mesure
- Méthode manuelle
- Commande `time`
- How Unix Keeps Time
- UTime : Micro Second resolution timers for Linux
- Résultats de la commande `time`
- Analyse de la commande `time`
- Mesure statistique : l'exemple des E/S
- Instrumentation du code
- Instrumentation manuelle
- Instrumentation manuelle : utilisation des registres matériels
- Profiling
- Principes du profiling
- Outils de profiling
- Etapes du profiling
- Profiler : Gprof
- Utilisation de Gprof
- "Flat view"
- Statistical Sampling Error
- "Call graph view"
- "Entrées" d'une "Call graph view"
- Profiler des bibliothèques
- "Frontend" graphique
- Kprof
- Conclusions

4 Index

Table des Matières (2)

Références

J.P. Braquelaire
Méthodologie de la programmation en Langage C - Principes et Applications.
Masson, 1995, 2e édition.

Kernighan B.W., Ritchie D.M
Le Langage C - C ANSI.
Masson, Prentice Hall, 1994, 2e édition.

P.Collet
Bibliothèques
2007-2008 Université de Nice Sophia Antipolis

P.Collet
Mesures de Performances
2007-2008 Université de Nice Sophia Antipolis

Qualité du logiciel

et

Analyse de Performances

De quel point de vue ?

Lorsque l'on aborde la thématique de la **qualité du logiciel** , il faut bien distinguer des points de vue qui ne sont pas forcément concordants.

Utilisateur/Développeur

① Les qualités **utiles à l'utilisateur** (et souhaitées par le client) :
- ➢ Phases d'exploitation

② Les qualités **utiles au développeur** :
- ➢ Phases de construction et de maintenance

Dilemnes typiques :

- ➢ fiabilité vs cout du développement,
- ➢ choix techniques vs adéquation fonctionnelle (cas des GUI),
- ➢ ...

Qualités pour le développeur

① Documentation

- ➢ Tout ce qu'il faut, rien que ce qu'il faut, là où il faut, quand il faut, correcte et adaptée au lecteur : **crucial** !

② Modularité = Fonctionnalité + Interchangeabilité + Évolutivité + Réutilisabilité

- ➢ Fonctionnalité
  - ➟ Localiser un phénomène unique, facile à comprendre et à spécifier
- ➢ Interchangeabilité
  - ➟ Pouvoir substituer une variante d'implémentation sans conséquence fonctionnelle (et souvent non-fonctionnelle) sur les autres parties
- ➢ Évolutivité
  - ➟ Facilité avec laquelle un logiciel peut être adapté à un changement ou une extension de sa spécification
- ➢ Réutilisabilité
  - ➟ Aptitude à être réutilisé, en tout ou en partie, tel que ou par adaptation, dans un autre contexte : autre application, autre machine, autre système . . .

Qualités pour l'utilisateur

① Fiabilité = Validité + Robustesse

- ➢ Validité (Efficacité) = correction, exactitude
  - ➟ Efficacité : qualité d'une chose ou d'une personne qui donne le résultat escompté
  - ➟ Assurer exactement les fonctions attendues, définies dans le cahier des charges et la spécification, en supposant son environnement fiable
  - ➟ Adéquation aux besoins
- ➢ Robustesse : faire tout ce qu'il est utile et possible de faire en cas de défaillance : pannes matérielles, erreurs humaines ou logicielles, malveillances. . .

② Performance (parfois appelée efficacité)

- ➢ Utiliser de manière optimale les ressources matérielles : temps d'utilisation des processeurs, place en mémoire, précision . . .

③ Convivialité

- ➢ Réaliser tout ce qui est utile à l'utilisateur, de manière simple, ergonomique, agréable (documentation, aide contextuelle . . .

Objectifs de qualité :

① Réduire le nombre d'erreurs résiduelles
② Maîtriser coût et durée du développement
③ Sans nuire à la créativité et à l'innovation

Sous des contraintes :

- Adéquation aux besoins,
- Efficacité temps/espace,
- Fiabilité,
- Testabilité,
- Traçabilité,
- Adaptabilité,
- Maintenabilité,
- Convivialité (interface et documentation).

`Un impératif industriel : rejoindre les objectifs de productivité ! !`

Analyse de performances

Les performances sont un critère évident de qualité.

- ➢ Dans un cours de compilation on travaillerait sur une conception/implémentation performante.
- ➢ Dans le cadre de ce cours, l'application existe et on l'analyse :
  - ➟ Il peut s'agir de mieux comprendre les limitations en temps/espace (occupation de l'espace mémoire) d'une application,
  - ➟ L'analyse peut aussi constituer une réflexion préalable à des améliorations :
    1. Changer la structure de code,
    2. Augmenter les capacités de la plateforme matérielle (# processeurs),
    3. ...

Pas d'optimisation sans analyse de performances !

- ➟ Sans comprendre, l'optimisation s'avère souvent peu rentable,
- ➟ Pire, engendre une complexité inutile et/ou des pertes de qualités !

  (par exemple la portabilité si vous décidez une évolution JNI ...)

Préambule algorithmique

La complexité . . . ça compte !

”La meilleure implémentation du monde ne peut sauver un mauvais algorithme”.

Ceci étant, l’optimisation de performance conserve nécessairement un intérêt :

- Même un facteur 2, c’est intéressant,
- et encore plus si le temps calcul initial est de 6 mois !

D’autant que les modèles de complexité sont tellement simplistes au regard des architectures matérielles et logicielles actuelles :

- ”Au fait, est-ce que ça vaut le coup de passer en Quadcore avec 12Mo de cache si l’application est écrite en `Java` sans `threading` ?”

Motivations

1)Trouver les **bottlenecks** :

Les **bottleneck** sont les goulots (souvent d'étranglements) de l'application.

Il existe une heuristique qui établit que :

"La plupart des programmes suivent la règle des 80/20"

➠ Ils exécutent 20 % du code pendant 80 % du temps.

2) Réduire les **bottlenecks** :

Evidemment, c'est là qu'il faut agir :

➠ Optimiser !

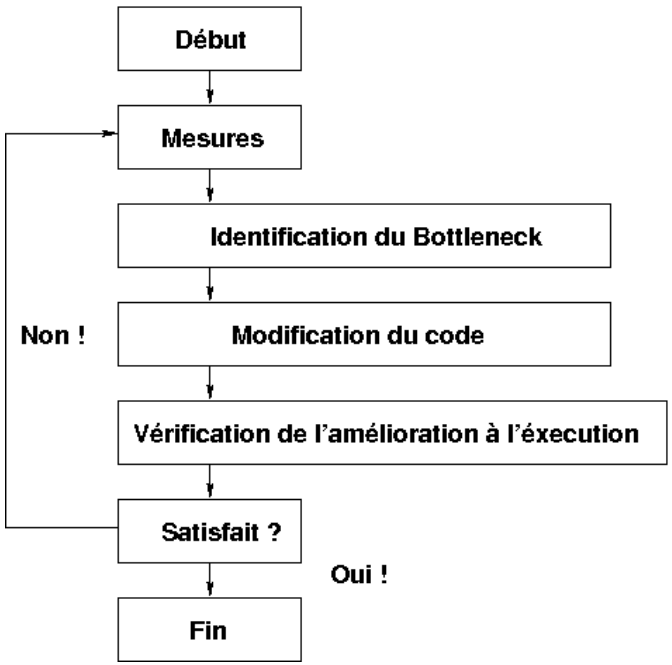
Mais attention aux conséquences . . .

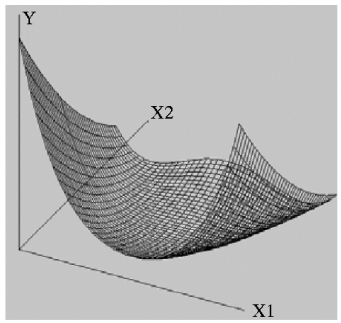
Cycle d'optimisation

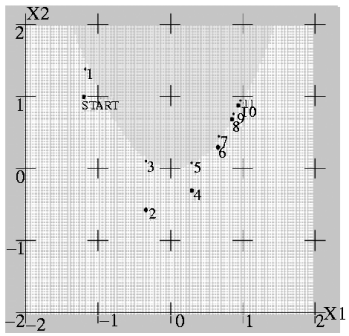
Prérequis au niveau du code mesuré :

- Complètement implémenté et testé . . . presque définitif !
- Sur une version `release` : on a mis les flags "Optimization" !
- Avec des données représentatives . . . "planification d'expériences" ?

Cycle :







Que mesure t'on ?

Difficile de théoriser :

Le système que l'on mesure est extrêmement complexe !

- OS : scheduling, gestion mémoire, accés disques, réseau ...
- Compilateurs : utilisation des flags, techniques de génération de code, ...
- Architecture matérielle : CPUs, Chipset, BUS, ...
- Code Tiers : Bibliothèques, ...

Compte tenu de ce qu'est un bottleneck, il n'est pas surpenant que l'analyse consiste à définir combien de temps est passé et dans quelles parties du code.

L'analyse cherche donc à répondre aux questions :

- Quelles fonctions sont souvent utilisées ?
- Quelles fonctions prennent le plus de temps CPU ?
- Quel est le coût des E/S ?

... Et ainsi localise les `bottlenecks`.

Méthodes de mesure

Plusieurs méthodes pour obtenir ces informations :

① Méthode manuelle
② Instrumentation du source : manuelle ou automatique
③ Mesure statistique (par échantillonnage)
④ Simulation
⑤ Compteurs Matériels

Pas nécessairement exclusive !

Méthode manuelle

Ca fait longtemps qu'on peut faire autrement qu'utiliser le chronographe de sa montre !
Il existe une commande UNIX de mesure globale du temps : `time` .

➢ `time(1)` can exist as a standalone program (such as GNU time) or as a shell builtin (e.g. in tcsh or in zsh).

`time <programme> <arguments du programme>` fournit :

① le **temps écoulé** (i.e. "elapsed time" ou "wall time" ou "real CPU time") :
temps réel de l'exécution de la commande = temps écoulé entre le lancement est la terminaison du programme. Ce temps dépend de la charge de la machine.

② le **temps utilisateur** (i.e "user time" ou "user CPU time") :
temps CPU utilisé par le programme utilisateur.

③ le **temps système** (i.e. "sys time") :
temps utilisé par le système (en mode noyau) pour gérer l'exécution du job.

Théoriquement sur un système mono* :

temps écoulé = temps utilisateur + temps système

Commande `time`

Exemple d'utilisation :

```
> time a.out
        real 0m4.43s
        user 0m0.08s
        sys 0m0.16s
```

Deux remarques :

① La résolution temporelle de la commande est de l'ordre de 10ms.

② La somme n'est pas juste !

$$real \neq user + sys$$

Ou est le temps manquant ?

How Unix Keeps Time

Epoch :

Comme tous les OS, Unix a le concept de temps.

- Lorsqu'il le système est off, c'est le matériel (pile quelconque) qui assure le maintien de l'heure,
- Lorsqu'il le système est booté, c'est l'OS qui gère une variable contenant le nombre de microsecondes depuis minuit 01/01/1970 GMT :
  Cette référence est appelée `epoch` .

Cette variable est mise à jour environ 100 fois par seconde.

- Le taux de rafraîchissement exact dépend d'une constante du noyau :

Dans /usr/include/sys/param.h
ou dans /usr/include/asm/param.h,

#define HZ 100

Résolution temporelle

Ce taux constitue la `résolution` temporelle :

- On y fait souvent référence comme le `tick` horloge.
- Ca n'a rien à voir avec la fréquence de l'horloge CPU.

Les mesures temporelles ne peuvent pas être plus précises que cela . . . sauf utilisation de registres matériels (cf plus loin.)

Les différents temps :

Le système comptabilise dans le bloc de contrôle de chaque processus le nombre de ticks en mode `user` et en mode `kernel`.

- Valeurs accessibles via l'API `times` .

UTime : Micro Second resolution timers for Linux

Les limites de *HZ* :

One of the ways to increase the temporal granularity of Linux would be to program the timer chip of the PC to interrupt the kernel at higher frequencies :

- ➢ L'interruption permettant au kernel de mettre à jour la variable de gestion du temps.

This is not an acceptable solution as **the overhead increase** due to this is **tremendous**.

- ➢ La machine passe son temps à mettre à jour une variable !
- ➢ For example, if we program the timer chip to interrupt the CPU at 40μs , the interrupt processing cost is so high there is no time left for any other computation.

So, basically, we need to program the timer chip to generate interrupts only when there is some scheduled work that needs to be accomplished.

This is what we have achieved :

`http://www.ittc.ku.edu/utime/`

Résultats de la commande `time`

`Ou est passé le temps manquant ?`

① Les opérations d'E/S consomment du temps qui n'apparaît pas dans les résultats de la commande.

- ➢ La gestion des E/S demande des calculs qui sont traités comme du temps système.
- ➢ Mais le temps consommé par les accès aux disques, les interfaces vers le réseau ou tout autre type de péripériques (gestion de la mémoire virtuelle) n'est pas pris en compte.

② L'essentiel du temps "manquant" a cependant été consommé à exécuter des tâches appartenant à d'autres utilisateurs.

Analyse de la commande `time`

Un `system time disproportionné` peut indiquer un dysfonctionnement

- Nombre important de `floating-point exceptions`,
- de "page faults",
- ...

Un `user CPU time < elapsed time` peut indiquer :

- Partage de la machine,
- Grand nombre d'E/S,
- Largeur de bande réseau disponible $<$ besoin de la machine,
- Pagination et/ou swap important,
- ...

Mesure statistique : l'exemple des E/S

Le système d'E/S est une source classique de conflits de ressources.

Petit rappel :

Une E/S (entrée/sortie) est une opération de transfert d'information entre la mémoire centrale et un périphérique.

Le problème c'est que tous les programmes en cours d'exécution (y compris le noyau Unix) doivent se partager une bande passante finie et délimitée par :

① les faibles capacités (au regard de celles du CPU) des périphériques,

② et par le bus de la machine.

Mesure statistique : l'exemple des E/S

Les outils disponibles pour analyser les problèmes induits par les E/S sont souvent basés sur une approche statistique qui permet d'établir des décomptes ou des moyennes.

A étudier : `iostat` , `vmstat` , `netstat`

Sur la base :

- `http://www.linuxjournal.com/article/8178`
- `http://www.drzz.net/articles.php?id=80`

La mesure statistique peut aussi concerner le choix "aléatoire" d'instances de test.

- Chaque instance « couvre » une partie des chemins possibles de l'application.

Instrumentation du code

Principe :

Il s'agit d'ajouter au code source initial, des instruments

➢ (généralement des appels de fonctions),

permettant de collecter puis de récupérer les informations souhaitées.

L'instrumentation peut être

➢ manuelle :
le testeur ajoute des lignes de codes "selon son instinct".

➢ ou automatique :
utilisation d'outils de **profiling** .

Instrumentation manuelle : exemple du ”temps cpu”

`http://www.gnu.org/s/libc/manual/html_node/CPU-Time.html`

To get a process' **CPU time**, you can use the `clock` function.

- This facility is declared in the header file time.h.
- . . . and then divide by CLOCKS_PER_SEC (the number of clock ticks per second) to get processor time, like this :

```
1  #include <time.h>
2  clock_t start, end;
3  double cpu_time_used;
4  start = clock();
5  ... /* Do the work. */
6  end = clock();
7  cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

Do not use a single CPU time as an amount of time : it doesn't work that way.

- Either do a subtraction as shown above or query processor time directly. (See Processor Time.)

Utilisation des registres matériels(1)

`http://www.mostang.com/~davidm/papers/expo97/paper/doc004.html`

The Unix way of measuring time is by calling gettimeofday(). This returns the current real time at a resolution of typically one timer tick (about 1ms on the Alpha).

- The advantage of this function is that it's completely portable across all Linux platforms.
- The disadvantage is its relatively poor resolution (1ms corresponds to 500,000 CPU cycles on a 500MHz CPU !)
- and, more severely, it involves a system call.

  A system call is relatively slow and has the tendency to mess up your memory system. E.g., the cache gets loaded with kernel code so when your program resumes execution, it sees many cache misses that it wouldn't see without the call to gettimeofday().

This is all right for measuring times on the order of seconds or minutes, but for finer-grained measurements, something better is needed.

Utilisation des registres matériels (2)

Fortunately, most modern CPUs provide a register that is incremented either at the clock frequency of the CPU or an integer fraction thereof.

- The Alpha architecture provides the `rpcc` (**read processor cycle count**) instruction.
- It gives access to a 64 bit register that contains a 32 bit counter in the lower half of the register.
- This counter is incremented once every N clock cycles.
- **All current chips use $N = 1$, so the register gets incremented at the full clock frequency** (but there may be future Alpha processors where $N > 1$).
- The top half of the value returned by rpcc is operating system dependent.

  Linux and Digital Unix return a correction value that makes it easy to implement a cycle counter that runs only when the calling process is executing (i.e., this allows to measure the process's virtual cycle count).

Utilisation des registres matériels (3)

With gcc, it's very easy to write inlined functions that provide access to the cycle counters :

```
1   static inline u_int realcc (void) {
2     u_long cc;
3     /* read the 64 bit process cycle counter into variable cc: */
4     asm volatile("rpcc %0" : "=r"(cc) : : "memory");
5     return cc;                    /* return the lower 32 bits */
6   }
7
8
9   static inline unsigned int virtcc (void) {
10    u_long cc;
11    asm volatile("rpcc %0" : "=r"(cc) : : "memory");
12    return (cc + (cc<<32)) >> 32; /* add process offset and count */
13  }
```

Utilisation des registres matériels (4)

With this code in place,

- function `realcc()` returns the 32 bit real-time cycle count
- whereas function `virtcc()` returns the 32 bit virtual cycle count (which is like the real-time count except that it doesn't count when the process isn't running).

Calling these functions involves **very small overheads** :

- the slowdown is on the order of 1-2 cycles per call and adds only one or two instructions (which is less than the overhead for a function call).

Utilisation des registres matériels (5)

A good way of using these functions is to create an execution time histogram.

- ➢ For example, the function below measures individual execution times of calls to sqrt(2.0) and prints the results to standard output (as usual, care must be taken to ensure that the compiler doesn't optimize away the actual computation).

Printing the individual execution times makes it easy to create a histogram with a little post-processing :

```
1  void measure_sqrt (void) {
2    u_int start, stop, time[10]; int i; double x = 2.0;
3    for (i = 0; i < 10; ++i) {
4      start = realcc(); sqrt(x); stop = realcc();
5      time[i] = stop - start;
6    }
7    for (i = 0; i < 10; ++i) printf(" %u", time[i]); printf("\n");
8  }
```

Note that the results are printed in a separate loop—this is important since printf is a rather big and complicated function that may even result in a system call or two.

- ➢ If printf were part of the main loop, the results would be much less reliable.

Utilisation des registres matériels (6)

A sample run of the above code might produce output like this :

120 101 101 101 101 101 101 101 101 101

Since this output was obtained on a 333MHz Alpha,

- 120 cycles corresponds to 36ns
- and 101 cycles corresponds to 30ns.

The output shows nicely how the first call is quite a bit slower since the memory system (instruction cache in particular) is cold at that point.

- Since the square-root function is small enough to easily fit in the first-level instruction cache, all but the first calls execute at exactly the same time.

Utilisation des registres matériels (7)

① You may wonder why the above code uses realcc() instead of virtcc().

The reason for this is simple : we want to know the results that were affected by a context switch.

➢ By using realcc(), a call that suffers a context switch will be much slower than any of the other calls.
This makes it easy to identify and discard such unwanted outliers.

② The cycle counter provides a very low-overhead method of measuring individual clock cycles.

➢ On the down side, it cannot measure very long intervals.

On an Alpha chip running at 500MHz, a 32 bit cycle counter overflows after just eight and a half seconds ! This is not normally a problem when making fine-grained measurements, but it is important to keep the limit in mind.

Utilisation des registres matériels (8)

Des bibliothèques pour Linux et i386 :

`http://www.scl.ameslab.gov/Projects/Rabbit/menu.html`

Profiling

Objectifs de la méthode :

Le **profiling** (”profilage” en francais) permet d’accèder à une vision **locale** de l’analyse de performances :

- Permet de déterminer les parties de code les plus consommatrices avant le travail d’optimisation,
- Récupère des informations sur :
  - Le temps,
  - le nombre d’appels des procédure,
  - le graphe des appels,
  - des statistiques sur des compteurs hardwares . . .

Différentes granularités sont possibles pour la notion de ”vision locale” :

1. la fonction,
2. la boucle,
3. le bloc de base,
4. et même la ligne de code.

Principes du profiling

Il existe plusieurs types de profiling :

① **PC Sampling** : interruption périodique du système ou trap des compteurs hardware.

- ➢ A chaque interruption . . . on regarde où on est ?
- ➢ On peut ainsi par exemple générer un histogramme ”nombre d’appel d’une fonction”.
- ➢ Ne nécessite en général pas de modification du code.
- ➢ La qualité de l’estimation dépend de la fréquence d’échantillonnage : C’est bien une approche statistique !

② **Basic-Bloc Counting** : l’unité d’analyse est le bloc de base.

- ➢ Produit le comptage du nombre de fois que les instructions s’exécutent.
- ➢ Ce comptage est réalisé grace à du code placé en entrée des BBs.

③ **Instrumentation** automatique par le **profiler** (i.e. outil de profiling).

- ➢ On récupère tout un tas d’informations . . .

Outils de profiling

1. Outils génériques :
   - prof, gprof sur les OS UNIX/linux
   - VPROF (http ://aros.ca.sandia.gov/ cljanss/perf/vprof/)
   - SvPablo
   - TAU (OpenMP, MPI, MPI/OpenMP)
   - VAMPIR / VampirTrace : MPI
   - HPCView
   - Dynaprof (nécessite dyninst ou DPCL)
2. Outils constructeurs :
   - perfex et ssrun (SGI IRIX),
   - tprof et trace (IBM),
   - PAT (CRAY),
   - histx, pfmon (Linux),
   - Vtune (Intel),
   - hiprof, pixie (Alpha Tru64)
   - ...

Etapes du profiling

① **Instrumentation** automatique du programme :

➢ Par le compilateur (`gcc` par exemple) pour le profiler via des options de compilation **-p**, **-pg**, ...

"Generate extra code to write profile information suitable for the analysis program gprof. You must use this option when compiling the source files you want data about, and you must also use it when linking."

➢ Par le profiler pour `hiprof` , `pixie` , `histx` ...

② **Exécution** du programme instrumenté :

➢ Crée un ou des fichiers de données pour le profiler (mon.out, gmon.out, ...)

③ **Utilisation** du profiler pour l'extraction et la lecture des résultats : prof, `gprof` , hiprof, iprep, ...

Profiler `Gprof`

- G pour Graph, pas pour GNU
- Profile les langages C, Fortran, Pascal
- Produit les informations sur le graphe d'appel
- Nécessite la recompilation et la reliure de l'application
- Calcule le temps passé dans chaque routine. Les temps sont ensuite propagés le long du graphe d'appel.
- Peut aussi fournir un source annoté, avec un profilage ligne par ligne

  -A[aymspec]

- Devenu un standard sous linux.

Utilisation de `Gprof`

① Compilation avec les options ≪ gprof ≫ :

CFLAGS = -pg

```
$(CC) -c $(CFLAGS) main.c -o a.out
```

L'option -pg est aussi utilisée en reliure pour relier les versions des bibliothèques qui ont été compilées pour le profilage.

② Exécution du programme : `./a.out`

➢ Génération d'un fichier `gmon.out` qui contient les informations mesurées en format interne.

③ Traduction des informations de mesure en format "lisible" :

```
gprof a.out gmon.out > mainc_profiling.txt
```

On peut fournir plusieurs fichiers de profilage (plusieurs jeux de données) afin que les informations de profilage s'additionnent :

```
gprof a.out gmon.out gmon1.out > mainc_profiling.txt
```

Informations obtenues : "Vue flat"

La "Vue Flat" montre :

- le temps CPU par fonction
- et combien de fois elle a été appelée.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
...
```

- Vue résumée des fonctions à réécrire ou à améliorer . . .

Informations obtenues : "Vue flat"

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
```

① **% time** : Le pourcentage du temps total d'exécution du programme utilisé par cette fonction.

② **cumulative seconds** : la somme des temps cumulés par cette fonction et celles aprés dans la liste.

③ **self seconds** : Le nombre de secondes consommées par cette fonction seule.
C'est le critère majeur d'ordonnancement de la liste.

④ **calls** : Le nombre d'appel de la fonction (si elle a été profilée, sinon rien).

⑤ **self ms/call** : Le nombre moyen de millisecondes de chaque appel (si la fonction a été profilée, sinon rien).

⑥ **total ms/call** : Le nombre moyen de millisecondes passées dans cette fonction et ses descendants (si la fonction a été profilée, sinon rien).

⑦ **name** : Le nom de la fonction.

http://sourceware.org/binutils/docs-2.19/gprof/index.html

Statistical Sampling Error

Une petite phrase débute la ”vue flat” :

- Elle indique la période d'échantillonnage utilisée par `gprof`

  Sampling rate = 100Hz soit 0.01s

Compte tenu du fait que le programme s'exécute en (0.06s), cela signifie qu'il y a eu 6 échantillons durant le programme :

- Deux fois, le programme était dans la fonction `open`
- Une fois, le programme était dans la fonction `offtime`, puis `memccpy`, `write`, `mcount`.

Echantillonnage : Danger !

Il est difficile de considérer ces informations comme significatives.

- Sur une autre exécution `mcount` pourrait parfaitement être en 0.02 . . . ou encore en 0.00

Informations obtenues : "Vue Call Graph"

La "Vue Call Graph" montre :

- Pour chaque fonction, le nombre de fois qu'elle a été appelée par différentes fonctions (et aussi par elle-même).

```
granularity: each sample hit covers 2 byte(s) for 20.00% of 0.05 seconds

index % time    self  children    called     name
                                               <spontaneous>
[1]     100.0    0.00    0.05                 start [1]
                 0.00    0.05       1/1           main [2]
                 0.00    0.00       1/2           on_exit [28]
                 0.00    0.00       1/1           exit [59]
-----------------------------------------------
                 0.00    0.05       1/1           start [1]
[2]     100.0    0.00    0.05       1         main [2]
                 0.00    0.05       1/1           report [3]
-----------------------------------------------
                 0.00    0.05       1/1           main [2]
[3]     100.0    0.00    0.05       1         report [3]
                 0.00    0.03       8/8           timelocal [6]
                 0.00    0.01       1/1           print [9]
                 0.00    0.01       9/9           fgets [12]
                 0.00    0.00      12/34          strncmp <cycle 1> [40]
                 0.00    0.00       8/8           lookup [20]
                 0.00    0.00       1/1           fopen [21]
                 0.00    0.00       8/8           chewtime [24]
                 0.00    0.00       8/16          skipspace [44]
-----------------------------------------------
[4]      59.8    0.01        0.02        8+472     <cycle 2 as a whole>   [4]
                 0.01        0.02      244+260         offtime <cycle 2> [7]
                 0.00        0.00      236+1           tzset <cycle 2> [26]
-----------------------------------------------
```

- Suggestion de quels appels peuvent être supprimés ou remplacés par des fonctions plus efficaces,
- Détermination des interrelations entre différentes fonctions (découverte de bugs),
- Possibilité d'optimisation de certains chemins dans le graphe d'appels.

"Entrées" d'une "Call graph view"

La vue est divisée (par des lignes hachurées) en **entrées** : Une pour chaque fonction.

- ➢ Chaque entrée a une ou plusieurs lignes.

```
granularity: each sample hit covers 2 byte(s) for 20.00% of 0.05 seconds

index % time    self  children    called     name
                                               <spontaneous>
[1]     100.0    0.00    0.05                  start [1]
                 0.00    0.05       1/1            main [2]
                 0.00    0.00       1/2            on_exit [28]
                 0.00    0.00       1/1            exit [59]
-----------------------------------------------
                 0.00    0.05       1/1            start [1]
[2]     100.0    0.00    0.05       1          main [2]
                 0.00    0.05       1/1            report [3]
-----------------------------------------------
                 0.00    0.05       1/1            main [2]
[3]     100.0    0.00    0.05       1          report [3]
                 0.00    0.03       8/8            timelocal [6]
                 0.00    0.01       1/1            print [9]
                 0.00    0.01       9/9            fgets [12]
                 0.00    0.00      12/34           strncmp <cycle 1> [40]
                 0.00    0.00       8/8            lookup [20]
                 0.00    0.00       1/1            fopen [21]
                 0.00    0.00       8/8            chewtime [24]
                 0.00    0.00       8/16           skipspace [44]
-----------------------------------------------
[4]      59.8    0.01        0.02        8+472      <cycle 2 as a whole>   [4]
                 0.01        0.02      244+260          offtime <cycle 2> [7]
                 0.00        0.00      236+1            tzset <cycle 2> [26]
-----------------------------------------------
```

Dans chaque entrée, il y a une **ligne primaire** avec un index entre crochets.

- ➢ La fin de cette ligne indique la fonction correspondante à l'entrée.
- ➢ Les lignes précedentes (de la ligne primaire) indique les fonctions *appelantes*.
- ➢ Les lignes suivantes indique les fonctions appelées (*subroutines*).

Analyse d'une entrée

Vue « call graph » de make Apache

Temps total dans eval

```
index % time    self  children    called           name
-----------------------------------------------
[25]     3.7    0.00     0.00       6         eval  [25]
               0.00     0.00     219/219        try_variable_definition [28]
               0.00     0.00      48/48         record_files [40]
               0.00     0.00     122/314        variable_expand_string  [59]
               0.00     0.00       5/314        allocated_variable_expand_file  [85]
               0.00     0.00     490/490        readline [76]
               0.00     0.00     403/403        collapse_continuations [79]
               0.00     0.00     355/355        remove_comments [80]
               0.00     0.00     321/963        find_char_unquote [66]
               0.00     0.00     170/170        get_next_mword [88]
               0.00     0.00     101/111        parse_file_seq [93]
               0.00     0.00     101/111        multi_glob [92]
               0.00     0.00      48/767        next_token [70]
               0.00     0.00      19/870        variable_buffer_output [68]
               0.00     0.00     13/2529        xmalloc [64]
               0.00     0.00       2/25         xrealloc [99]
                                   5            eval_makefile  [49]
```

Index des fonctions à la fin

Nb d'appel fait par eval

Nombre total d'appels non récursifs faits à la fonction

Appels (mutuellement) récursifs

23

Profiler des bibliothèques

Gprof ne profile pas les bibliothèques partagées. Il faut utiliser sur le même principe que `Gprof` :

Sprof et LD_PROFILE

Une bibliothèque à la fois !

① Placer la variable `LD_PROFILE` avec le nom de la bibliothèque partagée

export LD_PROFILE=my_obj

② Exécuter l'application

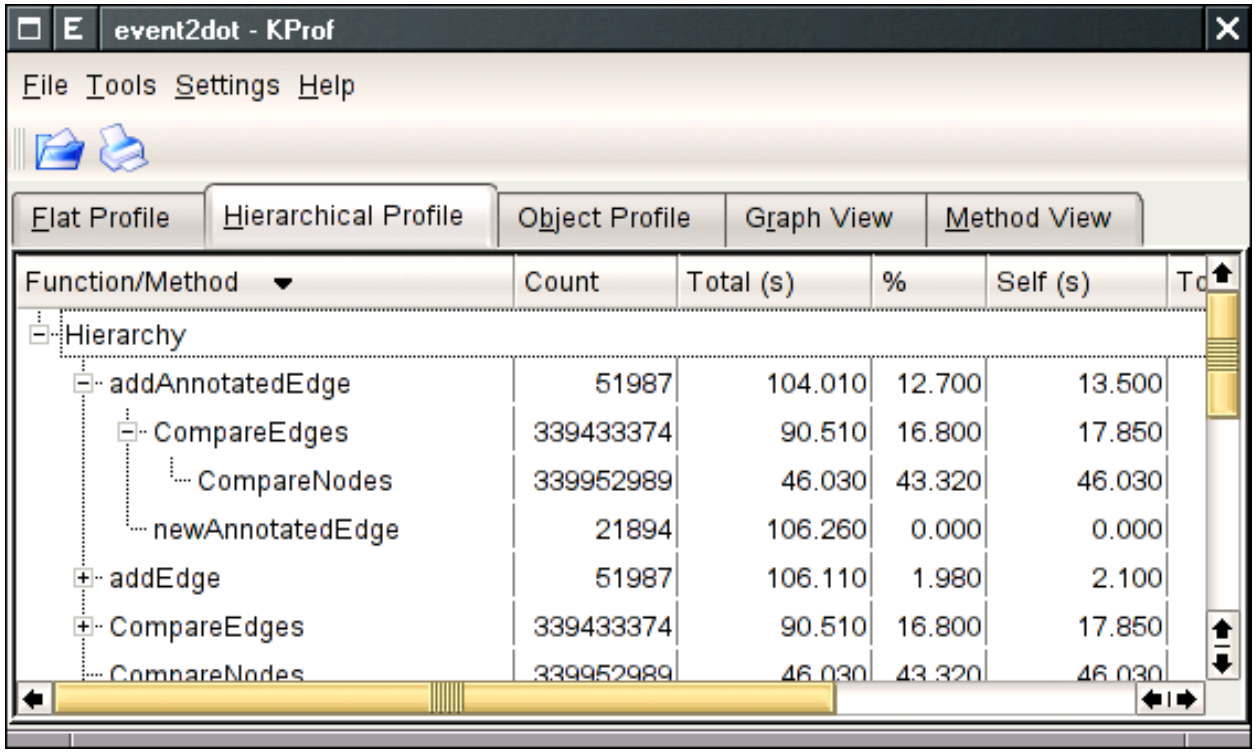
③ Cela crée un fichier `/var/tmp/my_sobj.profile`

④ Exécuter `sprof` :

```
sprof my_sobj my_sobj.profile
```

”Frontend” graphique : Kprof

Il présente les informations de profiling graphiquement : ”list-view”, ”tree-views”, . . .



http://kprof.sourceforge.net/

Supported profilers are :

- GNU gprof, the most commonly available profiler of unix platforms.
- Function Check, a recent and much better profiler for Unix using special GCC tricks.
- Palm OS Emulator (compiled with profiling turned on), which can generate execution profiling results for Palm OS applications.

KProf provides access to the following features :

- Flat profile view displays all function / methods and their profiling information.
- Hierarchical profile view displays a tree for each function / method with the other functions / methods it calls as subelements.
- Object profile view, for C++ developers, groups the methods in a tree view by object name.
- Graph view is a graphical representation of the call-tree, requires GraphViz to work.
- Method view is a more detailed look at an individual method - cross referenced.
- Recursive functions carry a special icon to clearly show that they are recursive.
- Right-clicking a function or method displays a pop-up with the list of callers and called functions. You can directly go to one of these functions by selecting it in the pop-up menu.
- The flat profile view provides an additional filter edit box to filter the display and show only the functions or methods containing the text that you enter.
- Function parameters hiding if the function name is unique (i.e. no different signatures)
- C++ template abbreviation (template parameters can be hidden)
- Automatic generation of call-graph data for GraphViz and VCG, two popular graph image generators.
- Diff mode support to compare two profile results.

Conclusions

Des méthodes et des outils souvent faciles d'accés.

- ➢ Si l'on en connait les pièges . . .

```
! Phase indispensable avant toute optimisation !
```

Index :