Analyse de performance et optimisation

David Geldreich (DREAM), 19/11/2004

Plan de l’exposé

- **Analyse de performance**
- Outils
- Optimisation
- Démonstrations

Analyse de performance

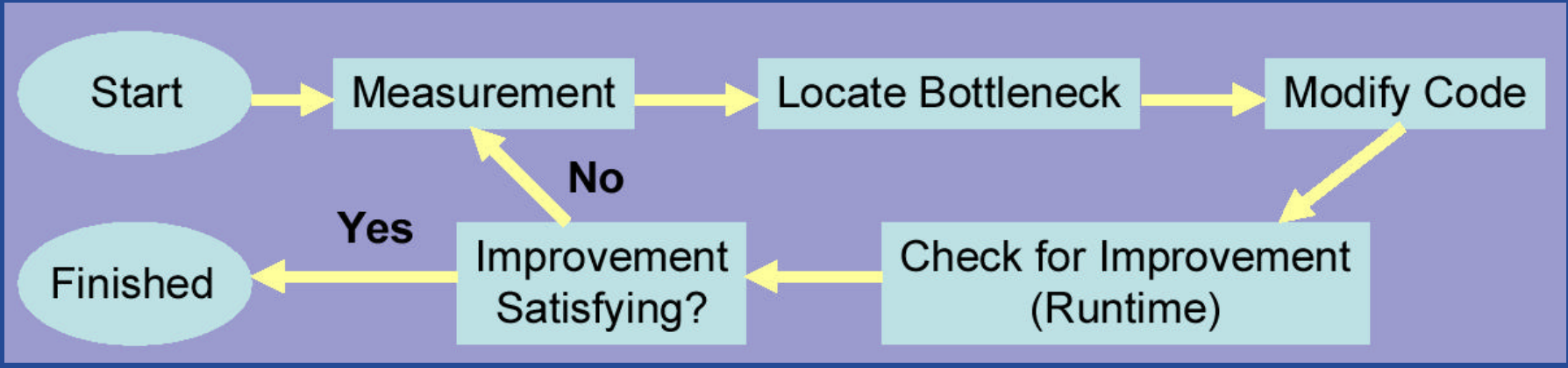
- Pas d'optimisation sans analyse de performance
- Tout le monde n'a pas un P4 @ 3Ghz
- "*We should forget about small efficiencies, say about 97% of the time : premature optimization is the root of all evil.*"
  Donald Knuth

Analyse de performance : quand ?

- Trouver les parties de code à optimiser
- Choisir parmi plusieurs algorithmes pour un problème donné
- Vérifier le comportement attendu à l'exécution
- Comprendre du code inconnu

Analyse de performance : comment ?

- Sur un code complètement implémenté (et aussi testé)
- Sur une version “release” (optimisée par le compilateur, …)
- Avec des données représentatives
- Le cycle d’optimisation :



Que mesure-t-on ?

- Bien comprendre ce qui se passe :
  - OS: scheduling, gestion de la mémoire, disque, réseau
  - Compilateur: optimisation
  - Architecture CPU, chipset, mémoire
  - Bibliothèques utilisées
- Si une application est limitée par ses I/O, inutile d'améliorer la partie calcul.
- On se limite ici aux performances CPU, mémoire.

Méthodes de mesure

- Manuel
- Instrumentation du source
- Mesure statistique (échantillonnage)
- Simulation
- Compteurs hardware

Plan de l’exposé

- Analyse de performance
- **Outils**
- Optimisation
- Démonstrations

Les outils

- Timer système (connaître la résolution)
  - Windows: QueryPerformanceCounter/QueryPerformanceFrequency
  - Linux/Unix: gettimeofday
  - Java: System.currentTimeMillis()
  - Compteur CPU Intel : RDTSC (ReaD Time Stamp Counter)
- gprof / gcc -pg
- valgrind (callgrind, massif) / kcachegrind
- IBM Rational quantify

Les outils (suite)

- Oprofile
- Intel Vtune
- PAPI (Performance API)
- JVMPI (Java Virtual Machine Profiler Interface):
  - runhprof, OptimizeIt, JMP (Java Memory Profiler)

Utilisation des outils

gprof (instrumentation générée par le compilateur) :

- compte les appels aux fonctions
- échantillonnage temporel
- compilation avec `gcc -pg`
- à l'exécution création d'un fichier `gmon.out`

Inconvénients

- que des infos sur les fonctions
- résultats pas toujours facile à analyser pour de grosses applis

Utilisation des outils

callgrind/kcachegrind : http://kcachegrind.sf.net/

- simulateur de cache au dessus de valgrind
- simulation du fonctionnement du processeur : estimation du nombre de cycle CPU pour chaque ligne de code
- kcachegrind permet d'analyser plus facilement les données

Inconvénients

- les estimations de temps peuvent être inexacte
- mesure uniquement la partie user du code

Utilisation des outils

Utilisation de callgrind/kcachegrind :

- sur le programme déjà compilé :

  `valgrind --tool=callgrind prog`
- génère un fichier `callgrind.out.xxx`
- analyse avec `callgrind_annotate` ou `kcachegrind`
- pour être utilisable malgré sa lenteur :

  ne pas simuler l'utilisation du cache : `--simulate-cache=no`

  ne lancer l'instrumentation qu'au moment utile :

  `--instr-atstart=no / callgrind_control -i on`

Utilisation des outils

massif (heap profiler) : http://valgrind.kde.org/tools.html

- tool de valgrind utilisé sur le programme compilé :

```
valgrind --tool=massif prog
```

- génère un `massif.xxx.ps` : graphe d'utilisation de la mémoire en fonction du temps
- `massif.xxx.txt` : quelle partie de code utilise quoi

Utilisation des outils

java/runhprof

- extension de la JVM Sun

```
java -Xrunhprof:cpu=samples,depth=6,thread=y prog
```

- génére un fichier `java.hprof.txt`
- analyse avec perfanal :

```
java -jar PerfAnal.jar java.hprof.txt
```

- mémoire :

```
java -Xrunhprof:heap=all prog
```

Inconvénients

- sampling grossier
- n'utilise pas toutes les possibilités de JVMPI

Plan de l’exposé

- Analyse de performance
- Outils
- **Optimisation**
- Démonstrations

Optimisation

- Ne pas optimiser trop tôt
- Garder un code maintenable
- Ne pas suroptimiser
- Ensemble de « recettes »
  - validité « limitée » dans le temps
  - spécifique à certains langages

Optimisation (suite)

- Trouver un meilleur algorithme
  - le facteur constant peut cependant rester important
- Écrire un bench/test de performance
  - permet de vérifier que la performance ne se dégrade pas
- Le goulot d'étranglement se déplace au fil des optimisations
  - exemple : les E/S peuvent devenir bloquantes

Optimisation (suite)

- Accès mémoire : première cause de lenteur
  - vitesse processeur progresse plus vite que vitesse mémoire
  - favoriser la localité des accès aux données : utilisation des caches
- Ordres de grandeur
  - Intel Pentium 4 @ 2.2Ghz, FSB 400Mhz
  - 8 KB L1 data cache : 128 lignes de 64 octets (4-way associative)
  - 12k micro-ops L1 execution cache
  - 512 KB L2 cache : 3 fois plus lent que le cache L1
  - mémoire : plus de 10 fois plus lente que le cache L1

Optimisation (suite)

- Utiliser des bibliothèques déjà optimisées
- Limiter le nombre d'appels aux fonctions coûteuses
- Utiliser des types de données adaptés (float vs double, …)
- Ordre des tests dans les `if` :
  - en C/C++, le compilateur n'a pas le droit de changer l'ordre

```
if (a == 0 && b.getFont() && c.compute())
```

Optimisation (suite)

- Laisser le compilateur faire son travail mais aidez-le :
  - sortir le code constant des boucles
  - par exemple :

```
for (i = 0; i < ima.width()*ima.height(); i++){…}
```

donne

```
int iLen = ima.width*ima.height();
for (i = 0; i < iLen; i++) {…}
```

- Écrire du code clair

Optimisation (suite)

- Aider les « features » du processeur :
  - branch prediction: mettre en premier le code de plus forte probabilité
  - out-of-bound exec : limiter les chaînes de dépendances dans le code

```
for (i = 0; i < n; i++)
   a[i] = i*i;
for (i = 0; i < n; i++)
   b[i] = 2*i;
   donne
for (i = 0; i < n; i++) {
   a[i] = i*i;
   b[i] = 2*i;
}
```

Optimisation (suite)

- Utilisation de l'assembleur à éviter
  - processeurs récents complexes : le compilateur fait mieux que nous
  - code difficile à lire et maintenir
- Cas des instructions SIMD mal gérées par les compilateurs
  - Intel: MMX, SSE/SSE2/SSE3

    http://developer.intel.com/design/pentium4/manuals/index_new.htm
  - PowerPC: Altivec

    http://developer.apple.com/hardware/ve/index.html
  - Sparc: Vis

    http://www.sun.com/processors/vis/

Optimisation : java

- Comprendre le fonctionnement de la JVM
  - Interprété vs JIT, techno HotSpot
  - `java -client` vs `java -server`
- Principale cause de lenteur : création/destruction d'objets
- Une mine d'informations sur la performance en java :
  http://java.sun.com/docs/performance/

Optimisation : multi-thread, distribué

- Quand un seul processeur ne suffit plus
- Parallélisation « automatique » : OpenMP
- Problématiques spécifiques : source de bugs

Démonstrations

- gprof
- callgrind/kcachegrind
- massif
- java/runhprof