Laborator 12 - Profiling

Materiale ajutătoare

- lab12-slides.pdf

Nice to Watch

- Google I/O 2010 - Measure in milliseconds: Meet Speed Tracer
- MIT Lecture: Performance Engineering with Profiling Tools

Prezentare teoretică

Introducere

Un **profiler** este un utilitar de analiză a **performanţei** care ajută programatorul să determine punctele critice ? **bottleneck** ? ale unui program. Acest lucru se realizează prin investigarea comportamentului programului, evaluarea consumului de memorie şi relaţia dintre modulele acestuia.

Tehnici de profiling

Tehnica de instrumentare

Profiler-ele bazate pe această tehnică necesită de obicei **modificări** în codul programului: se inserează secţiuni de cod la începutul şi sfârşitul funcţiei ce se doreşte analizată. De asemenea, se reţin şi funcţiile apelate. Astfel, se poate estima timpul total al apelului în sine cât şi al apelurilor de subfuncţii.
Dezavantajul major al acestor profilere este legat de modificarea codului: în funcţii de dimensiune scăzută şi des apelate, acest overhead poate duce la o interpretare greşită e rezultatelor.

Tehnica de eşantionare (sampling)

Profiler-ele bazate pe sampling **nu fac schimbări** în codul programului, ci verifică periodic procesorul cu scopul de a determina ce funcţie (instrucţiune) se executa la momentul respectiv. Apoi estimează frecvenţa şi timpul de execuţie al unei anumite funcţii într-o perioada de timp.

Suport pentru profiler

Suportul pentru profilere este disponibil la nivel de:

- **biblioteca C** (GNU libc), prin informaţii de timp de viaţă al alocărilor de memorie,
- **compilator**, prin modificarea codului în tehnica de **instrumentare** se poate realiza uşor în procesul de compilare, compilatorul fiind cel ce inserează secţiunile de cod necesare,
- **nucleu** al sistemului de operare, prin punerea la dispoziţie de apeluri de sistem specifice,
- **hardware**, unele procesoare sunt dotate cu contoare de temporizare (Time Stamp Counter - TSC) sau contoare de performanţă care numără evenimente precum cicluri de procesor sau TLB miss-uri.

Unelte

În continuare sunt prezentate câteva unelte folosite în profiling.

gprof

`gprof` este utilitarul de instrumentare folosit în combinaţie cu `GCC`. Vom lua ca exemplu următoarea secvenţă de cod:

main.c

```
void a_foo()
{
        <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("a_foo\n");
        usleep(2100);
}

void b_foo()
{
        <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("b_foo\n");
        usleep(4200);
}


int main(void)
{
        int i;
        for (i = 0; i < 1000; i++) {
                a_foo();
                b_foo();
                a_foo();
        }
        return 0;
}
```

Pentru folosirea `gprof` trebuie compilat programul cu suport de profiling prin folosirea opţiunii `-pg`:

1. Compilare: gcc -Wall -g -pg main.c -o main
2. Rulare: ./main
3. Analiză $ ls
   gmon.out main main.c

Avem un fişier `gmon.out` care conţine datele analizei. Acestea pot fi afişate în mai multe forme:

- flat profile: arată cât timp durează execuţia unei funcţii şi de câte ori a fost apelată fiecare funcţie.
- call graph: arată pentru fiecare funcţie - funcţiile apelante, funcţiile apelate şi de câte ori.
- annotate source listing: copie a programului iniţial împreună cu numărul de execuţii pentru fiecare linie.

Folosim acum `gprof` pentru a afişa flat profile şi call graph pentru programul `main.c`.

```
$ gprof ./main
Flat profile:
[...]
 time   seconds   seconds    calls  us/call  us/call  name
 50.33      0.01     0.01     2000     2.52     2.52  a_foo
 50.33      0.01     0.01     1000     5.03     5.03  b_foo
[...]

Call graph
index % time    self  children    called         name
                                                 <spontaneous>
[1]    100.0    0.00    0.01                     main [1]
                0.01    0.00    2000/2000        a_foo [2]
                0.01    0.00    1000/1000        b_foo [3]
-----------------------------------------------
                0.01    0.00    2000/2000        main [1]
[2]     50.0    0.01    0.00    2000             a_foo [2]
-----------------------------------------------
                0.01    0.00    1000/1000        main [1]
[3]     50.0    0.01    0.00    1000             b_foo [3]
-----------------------------------------------
```

Observaţi că:

- timpul de execuţie al fiecărei funcţii este aproximativ egal cu timpul din `usleep`.
- timpul total de execuţie al funcţiei `a_foo` este egal cu cel al funcţiei `b_foo`.

perfcounters

Majoritatea procesoarelor moderne oferă registre speciale (`performance counters`) care contorizează diferite tipuri de evenimente hardware: instrucţiuni executate, cache-miss-uri, instrucţiuni de salt anticipate greşit, fără să afecteze performanţa nucleului sau a aplicaţiilor. Aceste registre pot declanşa întreruperi atunci când se acumulează un anumit număr de evenimente şi astfel se pot folosi pentru analiza codului care rulează pe procesorul în cauză.

Subsistemul `perfcounters`:

- se găseşte în nucleul Linux începând cu versiunea 2.6.31 (`CONFIG_PERF_COUNTERS=y`)
- este înlocuitorul lui `oprofile`
- oferă suport pentru:
  - `evenimente hardware` (instrucţiuni, accese cache, ciclii de magistrală).
  - `evenimente software` (page fault, cpu-clock, cpu migrations).
  - `tracepoints` (e.g: sys_enter_open, sys_exit_open).

perf

Utilitarul `perf` este interfaţa subsistemului `perfcounters` cu utilizatorul. Oferă o linie de comandă asemănătoare cu `git` şi nu necesită existenţa unui daemon.

Un tutorial despre perf - tutorial perf.

Utilizare

```
$ perf [--version] [--help] COMMAND [ARGS]
```

Cele mai folosite `comenzi` sunt:

- `annotate` - Read perf.data and display annotated code
- `list` - List all symbolic event types
- `lock` - Analyze lock events
- `record` - Run a command and record its profile into perf.data
- `report` - Read perf.data (created by perf record) and display the profile
- `sched` - Tool to trace/measure scheduler properties (latencies)
- `stat` - Run a command and gather performance counter statistics
- `top` - System profiling tool.

perf list

- man perf-list

Afişează numele simbolice ale tuturor tipurilor de evenimente ce pot fi urmărite de `perf`.

```
$ perf list
List of pre-defined events (to be used in -e):

  cpu-cycles OR cycles                       [Hardware event]
  instructions                               [Hardware event]

  cpu-clock                                  [Software event]
  page-faults OR faults                      [Software event]

  L1-dcache-loads                            [Hardware cache event]
  L1-dcache-load-misses                      [Hardware cache event]

  rNNN                                       [Raw hardware event descriptor]

  mem:<addr>[:access]                        [Hardware breakpoint]

  syscalls:sys_enter_accept                  [Tracepoint event]
  syscalls:sys_exit_accept                   [Tracepoint event]
```

Atunci când un eveniment nu este disponibil în forma simbolică, poate fi folosit cu perf în forma procesorului din sistemul analizat.

perf stat

- perf-stat

Rulează o comandă şi afişează statisticile înregistrate de subsistemul performance counters.

```
$ perf stat ls -R /usr/src/linux
 Performance counter stats for 'ls -R /usr/src/linux':

         934.512846  task-clock-msecs         #      0.114 CPUs
              1695  context-switches         #      0.002 M/sec
               163  CPU-migrations           #      0.000 M/sec
               306  page-faults              #      0.000 M/sec
         725144010  cycles                   #    775.959 M/sec
         419392509  instructions             #      0.578 IPC
          80242637  branches                 #     85.866 M/sec
           5680112  branch-misses            #      7.079 %
         174667968  cache-references         #    186.908 M/sec
           4178882  cache-misses             #      4.472 M/sec

        8.199187316  seconds time elapsed
```

`perf stat` oferă posibilitatea colectării datelor în urma rulării de mai multe ori a unui program specificând opţiunea `-r`.

```
$ perf stat -r 6 sleep 1
 Performance counter stats for 'sleep 1' (6 runs):

           1.757147  task-clock-msecs #      0.002 CPUs    ( +-   3.000% )
                  1  context-switches #      0.001 M/sec   ( +-  14.286% )
                  0  CPU-migrations   #      0.000 M/sec   ( +- 100.000% )
                144  page-faults      #      0.082 M/sec   ( +-   0.147% )
            1373254  cycles           #    781.525 M/sec   ( +-   2.856% )
             588831  instructions     #      0.429 IPC     ( +-   0.667% )
             106846  branches         #     60.806 M/sec   ( +-   0.324% )
              11312  branch-misses    #     10.587 %       ( +-   0.851% )
        1.002619407  seconds time elapsed   ( +-   0.012% )
```

Observaţi mai sus evenimentele cele mai importante contorizate.

perf top

- man perf-top

Generează şi afişează informaţii în timp real despre încărcarea unui sistem.

```
$ ls -R /home
$ perf top -p $(pidof ls)
------------------------------------------------------------
   PerfTop:     181 irqs/sec  kernel:72.4% (target_pid: 10421)
------------------------------------------------------------
             samples  pcnt function             DSO
             _______ _____ _____________________ ____________________

              270.00 15.8% __d_lookup           [kernel.kallsyms]
              145.00  8.5% __GI___strcoll_l     /lib/libc-2.12.1.so
               99.00  5.8% link_path_walk       [kernel.kallsyms]
               97.00  5.7% find_inode_fast      [kernel.kallsyms]
               91.00  5.3% __GI_strncmp         /lib/libc-2.12.1.so
               55.00  3.2% move_freepages_block [kernel.kallsyms]
               44.00  2.6% ext3_dx_find_entry   [kernel.kallsyms]
               41.00  2.4% ext3_find_entry      [kernel.kallsyms]
               40.00  2.3% dput                 [kernel.kallsyms]
               39.00  2.3% ext3_check_dir_entry [kernel.kallsyms]
```

Observăm că funcţiile de lucru cu fişiere (parcurgere, căutare) sunt cele care apar cel mai des în outputul lui perf-top corespunzător rulării comenzii de listare recursivă a directorului `home`.

perf record

- man perf-record

Rulează o comandă şi salvează informaţiile de profiling în fişierul `perf.data`.

```
$ perf record wget http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-12

[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.008 MB perf.data (~334 samples) ]

$ ls
```

```
laborator-12  perf.data
```

perf report

- man perf-report

Interpretează datele salvate în `perf.data` în urma analizei folosind `perf-record`. Astfel pentru exemplul `wget` de mai sus avem:

```
$ perf report
# Events: 13  cycles
#
# Overhead  Command      Shared Object  Symbol
# ........  .......  .................  ......
#
    86.43%     wget             e8ee21  [.] 0x00000000e8ee21
    11.03%     wget  [kernel.kallsyms]  [k] prep_new_page
     2.37%     wget  [kernel.kallsyms]  [k] sock_aio_read
     0.11%     wget  [kernel.kallsyms]  [k] perf_event_comm
     0.05%     wget  [kernel.kallsyms]  [k] native_write_msr_safe
```

Alte utilitare

- Oprofile
- Kernrate este un echivalent al oprofile pentru Windows.
- KCachegrind
- perf-tools
- XPerf

Exerciţii

- Folosiţi arhiva lab12-tasks.zip aferentă laboratorului.

Instalati utilitarul perf pe statiile din laborator:

- sudo apt-get update
- sudo apt-get install linux-tools-3.2

1. (**2 puncte**) Intraţi în directorul `01-bubble-sort`.
   - Analizaţi conţinutul fişierului `bubble-sort.c`.
   - Implementaţi funcţia `bubble_sort`. Aceasta va realiza:
     - sortarea crescătoare a unui şir de întregi folosind algoritmul Bubble sort.
     - va folosi funcţia `swap` pentru a interschimba 2 elemente.
   - Comparaţi, folosind `gprof`, rezultatele obţinute în momentul utilizării funcţiei `bubble_sort` implementate în C şi funcţiei `bubble_sort` implementate în assembly:
     - definiţi `TYPE` cu valoarea `ASSEMBLY__` şi salvaţi rezultatele în `gmon-as.out`.
       - `$gprof ./bubble_sort > gmon-as.out`.
     - definiţi `TYPE` cu valoarea `C__` şi salvaţi rezultatele în `gmon-c.out`.
   - Folosiţi **apoi** flag-ul `-O2` pentru a compila sursa folosind suportul de optimizări ale compilatorului şi comparaţi rezultatele de profiling obţinute cu cele anterioare.
     - Observaţi `CFLAGS` din `Makefile`.
     - Salvaţi rezultatele în fişierele `gmon-as-O2.out` respectiv `gmon-c-02.out`.
   - Care sunt părţile din cod unde programul petrece cel mai mult timp?
   - **Hints**:
     - Revedeţi secţiunea gprof din laborator.
     - Puteţi folosi opţiunea `-b` cu `gprof` pentru a evita încărcarea output-ului cu explicaţii.
2. (**1 punct**) Intraţi în directorul `02-major`.
   - Folosind utilitarul `perf_3.2` determinaţi dacă limbajul C este column-major sau row-major (row-major-order).
     - Completaţi programul `row.c` astfel încât să incrementeze elementele unei matrice pe linii.
     - Completaţi programul `columns.c` astfel încât să incrementeze elementele unei matrice pe coloane.
     - Determinaţi numărul de cache-miss-uri comparativ cu numărul de accese la cache.
     - Folosiţi `perf stat` pentru a urmări evenimentele `L1-dcache-loads` şi `L1-dcache-load-misses`.
   - **Hints**:
     - Folosiţi opţiunea `-e` a utilitarului `perf` pentru a specifica un anumit eveniment de urmărit.

    - Revedeţi secţiunea perfcounters din laborator.
3. (**0.5 puncte**) Intraţi în directorul `03-busy`
    - Inspectaţi fişierul `busy.c`.
    - Rulaţi programul `busy` şi analizaţi încărcarea sistemului folosind comanda `perf top`.
    - Ce funcţie pare să încarce sistemul?
4. (**2.5 puncte**) Intraţi în directorul `04-hash/`.
    - În directorul `04-hash/` se găseşte o implementare a unui algoritm pentru tabele de dispersie.
    - Deşi dimensiunea tabelului este dublă faţă de numărul de cuvinte, programul are o comportare ineficientă.
    - Folosiţi `gprof` pentru a îmbunătăţi comportarea programului.
    - **Hints**:
        - Determinaţi funcţiile cu cele mai multe apeluri, funcţiile care per total rulează cel mai mult.
5. (**2 puncte**) Intraţi în directorul `05-find-char/`.
    - Analizaţi conţinutul fişierului `find-char.c`.
    - Compilaţi fişierul `find-char.c` şi rulaţi executabilul obţinut.
    - Identificaţi, folosind `gprof`, care este funcţia care ocupă cel mai mult timp de procesor şi încercaţi să îmbunătăţiţi performanţele programului.
6. (**2 puncte**) Intraţi în directorul `06-tlb/`.
    - Analizaţi conţinutul fişierelor `tlb-processes.c`, respectiv `tlb-threads.c`. Compilaţi cele două programe.
        - `tlb-processes.c` - foloseşte 2 procese ce accesează o zonă de memorie partajată.
        - `tlb-threads.c` - foloseşte 2 fire de execuţie ce accesează o zonă de memorie partajată.
    - Contorizaţi timpul de execuţie folosind `time`.
    - Care program se rulează mai repede? De ce? Folosiţi `perf stat` pentru a determina acest lucru.
    - **Hints**:
        - Care dintre cele 2 programe obţine cele mai multe `tlb-misses`?
        - Folosiţi `perf list` pentru a determina evenimentul corespunzător cu `data tlb misses`.

Soluţii

Resurse utile

- GNU grof manual
- linux/tools/perf
- [Announce] Performance Counters for Linux, v8
- Profiling tools and techniques
- Is Parallel Programming Hard, And, If So, What Can You Do About It?