

Laborator 12 - Profiling

Materiale ajutătoare

- [lab12-slides.pdf](#)

Nice to Watch

- [Google I/O 2010 - Measure in milliseconds: Meet Speed Tracer](#)

Prezentare teoretică

Introducere

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

Tehnici de profiling

Tehnica de instrumentare

Profiler-ele bazate pe această tehnică necesită de obicei **modificări** în codul programului: se inserează secțiuni de cod la începutul și sfârșitul funcției ce se dorește analizată. De asemenea, se rețin și funcțiile apelate. Astfel, se poate estima timpul total al apelului în sine cât și al apelurilor de subfuncții.

Dezavantajul major al acestor profilere este legat de modificarea codului: în funcții de dimensiune scăzută și des apelate, acest overhead poate duce la o interpretare greșită e rezultatelor.

Tehnica de eșantionare (sampling)

Profiler-ele bazate pe sampling **nu fac schimbări** în codul programului, ci verifică periodic procesorul cu scopul de a determina ce funcție (instrucțiune) se execută la momentul respectiv. Apoi estimează frecvența și timpul de execuție al unei anumite funcții într-o perioadă de timp.

Suporturi pentru profiler

Suportul pentru profilere este disponibil la nivel de:

- **biblioteca C** (GNU libc), prin informații de timp de viață al alocărilor de memorie,
- **compiler**, prin modificarea codului în tehnica de **instrumentare** se poate realiza ușor în procesul de compilare, compilerul fiind cel ce inserează secțiunile de cod necesare,
- **nucleu** al sistemului de operare, prin punerea la dispoziție de apeluri de sistem specifice,
- **hardware**, unele procesoare sunt dotate cu contoare de temporizare ([Time Stamp Counter - TSC](#)) sau contoare de performanță 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 combinație cu GCC. Vom lua ca exemplu următoarea secvență 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 opțiunii `-pg`:

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

Avem un fișier `gmon.out` care conține datele analizei. Acestea pot fi afișate în mai multe forme:

- **flat profile**, arată cât timp durează execuția unei funcții și de câte ori a fost apelată fiecare funcție.
- **call graph**, arată pentru fiecare funcție - funcțiile apelante, funcțiile apelate și de câte ori.
- **annotated source listing**, copie a programului inițial împreună cu numărul de execuții pentru fiecare linie.

Folosim acum `gprof` pentru a afișa flat profile și 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]
-----

```

Observați că:

- timpul de execuție al fiecărei funcții este aproximativ egal cu timpul din `usleep`.
- timpul total de execuție al funcției `a_foo` este egal cu cel al funcției `b_foo`.

perfcounters

Majoritatea procesoarelor moderne oferă registre speciale (performance counters) care contorizează diferite tipuri

de evenimente hardware: instrucțiuni executate, cache-misses, instrucțiuni de salt anticipate greșit fără să afecteze performanța nucleului sau a aplicațiilor. Aceste registre pot declanșa întreruperi atunci când se acumulează un anumit număr de evenimente și astfel se pot folosi pentru analiza codului care rulează pe procesorul în cauză.

Subsistemul `perfcounters`:

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

perf

Utilitarul `perf` este interfața subsistemului `perfcounters` cu utilizatorul. Oferă o linie de comandă asemănătoare cu `git` și nu necesită existența unui daemon.

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](#)

Afișează 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`):

<code>cpu-cycles OR cycles</code>	[Hardware event]
<code>instructions</code>	[Hardware event]
<code>cpu-clock</code>	[Software event]
<code>page-faults OR faults</code>	[Software event]
<code>L1-dcache-loads</code>	[Hardware cache event]
<code>L1-dcache-load-misses</code>	[Hardware cache event]
<code>rNNN</code>	[Raw hardware event descriptor]
<code>mem:<addr>[:access]</code>	[Hardware breakpoint]
<code>syscalls:sys_enter_accept</code>	[Tracepoint event]
<code>syscalls:sys_exit_accept</code>	[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 și afișează 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 posibilitate colectării datelor în urma rulării de mai multe ori a unui program specificând opțiunea -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% )

```

Observați mai sus evenimentele cele mai importante contorizate.

perf top

- [man perf-top](#)

Generează și afișează informații 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 funcțiile de lucru cu fișiere (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 și salvează informațiile de profiling în fișierul 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](#)

Exerciții

- Folosiți arhiva [lab12-tasks.zip](#) aferentă laboratorului.

Instalati utilitarul `perf` pe statiile din laborator:

- `sudo apt-get update`
- `sudo apt-get install linux-tools-common`
- `sudo apt-get install linux-tools-2.6.35-27`

1. (2 puncte) intrați în directorul `01-bubble-sort`.

- Analizați conținutul fișierului `bubble-sort.c`.
- Implementați funcția `bubble_sort`. Aceasta va realiza:
 - sortarea crescătoare a unui șir de întregi folosind algoritmul [Bubble sort](#).
 - va folosi funcția `swap` pentru a interschimba 2 elemente.
- Comparați, folosind `gprof`, rezultatele obținute în momentul utilizării funcției `bubble_sort` implementate în C și funcției `bubble_sort` implementate în assembly.
 - definiți TYPE cu valoarea `ASSEMBLY` și salvați rezultatele în `gmon-as.out`.
 - `$gprof ./bubble_sort > gmon-as.out`
 - definiți TYPE cu valoarea `C` și salvați rezultatele în `gmon-c.out`.
- Folosiți **apoi** flag-ul `-O2` pentru a compila sursa folosind suportul de optimizări ale compilatorului și comparați rezultatele de profiling obținute cu cele anterioare.
 - Observați `CFLAGS` din `Makefile`.
 - Salvați rezultatele în fișierele `gmon-as-O2.out` respectiv `gmon-c-O2.out`.
- Care sunt părțile din cod unde programul petrece cel mai mult timp?
- **Hints:**
 - Revedeți secțiunea [gprof](#) din laborator.
 - Puteti folosi optiunea `-b` cu `gprof` pentru a evita incarcarea outputului cu explicatii.

2. (1 punct) Intrați în directorul `02-major`.

- Folosind utilitarul `perf` determinați dacă limbajul C este column-major sau row-major ([row-major-order](#)).
 - Completați programul `row.c` astfel încât să incrementeze elementele unei matrice pe linii.
 - Completați programul `columns.c` astfel încât să incrementeze elementele unei matrice pe coloane.
 - Determinați numărul de cache-misses comparativ cu numărul de accese la cache.
 - Folosiți `perf stat` pentru a urmări evenimentele `L1-dcache-loads` și `L1-dcache-load-misses`.
- **Hints:**
 - Folosiți opțiunea `-e` a utilitarului `perf` pentru a specifica un anumit eveniment de urmărit.
 - Revedeți secțiunea [perfcounters](#) din laborator.

3. (0.5 puncte) Intrați în directorul `03-busy`

- Inspectați fișierul `busy.c`.

- Rulați programul busy și analizați încărcarea sistemului folosind comanda `perf top`.
 - Ce funcție pare să încarce sistemul?
4. (2.5 puncte) intrați în directorul `04-hash/`.
- În directorul `04-hash/` se găsește o implementare a unui algoritm pentru tabele de dispersie.
 - Deși dimensiunea tabelului este dublă față de numărul de cuvinte, programul are o comportare ineficientă.
 - Folosiți `gprof` pentru a îmbunătăți comportarea programului.
 - **Hints:**
 - Determinați funcțiile cu cele mai multe apeluri, funcțiile care per total rulează cel mai mult.
5. (2 puncte) Intrați în directorul `05-find-char/`.
- Analizați conținutul fișierului `find-char.c`.
 - Compilați fișierul `find-char.c` și rulați executabilul obținut.
 - Identificați, folosind `gprof`, care este funcția care ocupă cel mai mult timp de procesor și încercați să îmbunătățiți performanțele programului.
6. (2 puncte) Intrați în directorul `06-tlb/`.
- Analizați conținutul fișierelor `tlb-processes.c`, respectiv `tlb-threads.c`. Compilați cele două programe.
 - `tlb-processes.c` - folosește 2 procese ce accesează o zonă de memorie partajată.
 - `tlb-threads.c` - folosește 2 fire de execuție ce accesează o zonă de memorie partajată.
 - Contorizați timpul de execuție folosind `time`.
 - Care program se rulează mai repede? De ce? Folosiți `perf stat` pentru a determina acest lucru?
 - **Hints:**
 - Care dintre cele 2 programe obține cele mai multe `tlb-misses`?
 - Folosiți `perf list` pentru a determina evenimentul corespunzător cu `data tlb misses`.

Soluții

Resurse utile

- [GNU gprof 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?](#)

From:

<http://elf.cs.pub.ro/so/wiki/> - **Sisteme de Operare**

Permanent link:

<http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-12>

Last update: 2011/05/13 10:49