Profiling

Contents

- 1 Introducere
- 2 Tehnici de profiling
  - 2.1 Tehnica de instrumentare
  - 2.2 Tehnica de eantionare (sampling)
  - 2.3 Suporturi pentru profiler
- 3 Linux
  - 3.1 gprof
  - 3.2 Oprofile
    - 3.2.1 Arhitectura i modul de funcionare
    - 3.2.2 Utilizare
      - 3.2.2.1 Configurare
      - 3.2.2.2 Pornire
      - 3.2.2.3 Oprire
      - 3.2.2.4 Colectarea i analizarea datelor
      - 3.2.2.5 Exemplu de rulare
- 4 Windows
  - 4.1 Kernrate si KrView
  - 4.2 XPerf
- 5 Exerciii
  - 5.1 Prezentare
  - 5.2 Exerciii de laborator
- 6 Soluii
- 7 Resurse utile

Introducere

Un profiler este un utilitar de analiză a performanei care ajută programatorul să determine punctele critice (bottleneck) ale unui program. Acest lucru se realizează prin investigarea comportamentului acestuia, evaluarea consumului de memorie i 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 seciuni de cod la începutul i sfâritul funciei ce se dorete analizată. De asemenea, se rein i funciile apelate. Astfel, se poate estima timpul total al apelului în sine cât i al apelurilor de subfuncii. Dezavantajul major al acestor profilere este legat de modificarea codului: în funcii de dimensiune scăzuta i des apelate, acest overhead poate duce la o interpretare greită e rezultatelor.

Tehnica de eantionare (sampling)

Profiler-ele bazate pe sampling nu fac schimbări in codul programului, ci verifica periodic procesorul cu scopul de a determina ce funcție (instrucțiune) se executa la momentul respectiv. Apoi estimează frecventa si timpul de execuție al unei anumite funcții într-o perioada de timp.

Suporturi pentru profiler

În Linux (i nu numai), suportul pentru profilere este disponibil la nivel de:

- biblioteca C (GNU libc), prin informaii de timp de viaă al alocărilor de memorie
- compilator. Modificarea codului în tehnica de instrumentare se poate realiza uor în procesul de compilare, compilatorul fiind cel ce inserează seciunile de cod necesare.
- nucleu al sistemului de operare, prin punerea la dispoziie de apeluri de sistem
- hardware: multe procesoare sunt dotate cu contoare de temporizare (Time Stamp Counter - TSC) sau contoare de performană care numără evenimente ca cicluri de procesor sau TLB misses.

Linux

gprof

`gprof` este utilitarul de profiling folosit în combinaie cu `gcc`. Pentru folosirea `gprof` trebuie compilat programul cu suport de profiling prin folosirea opiunii `-pg`.

Astfel, pentru fiierul de mai jos:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAXLEN                1024
#define NEEDLE                100

static void init_array_rand(int *v, size_t len)
{
        int i;

        srand(time(NULL));
        for (i = 0; i < len; i++) {
                v[i] = rand();
        }
}

static int linear_search(int *v, size_t len, int needle)
{
        size_t i;

        for (i = 0; i < len; i++) {
                if (v[i] == needle)
                        return i;
```

```
        }

        return -1;
}

int main(void)
{
        int v[MAXLEN];

        init_array_rand(v, MAXLEN);
        printf("a fost gasit elementul %d pe linia %d\n", NEEDLE, linear_search(v, MAXLEN, NEEDLE

        return 0;
}
```

După compilare:

```
$ # se compilează programul
$ gcc -Wall -g -pg    test.c   -o test
$ # se execută programul i se obine un fiier 'gmon.out' cu informaii de profiling
$ ./test
gasit elementul 100 pe linia -1
$ ls
gmon.out  Makefile  test  test.c
$ # se execută 'gprof' pentru a extrage informaiile de profiling din 'gmon.out' (vor fi scrise în
$ gprof ./test > out.txt
$ cat out.txt
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00        1     0.00     0.00  init_array_rand
  0.00      0.00     0.00        1     0.00     0.00  linear_search

[...]
```

Se observă că timpii obtinuti nu sunt specificai în unităi măsurabile. Pentru aceasta se va executa codul din funcia `main` de un număr dat de ori:

```
#define NLOOPS                  10000
[...]

int main(void)
{
int i;
        int v[MAXLEN];

for (i = 0; i < NLOOPS; i++) {
                init_array_rand(v, MAXLEN);
             printf("gasit elementul %d pe linia %d\n", NEEDLE, linear_search(v, MAXLEN, NEEDLE))
}

return 0;
}
```

Informaia obinută este următoarea:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 53.15      0.10     0.10    10000    10.10    10.10  init_array_rand
 47.84      0.19     0.09    10000     9.09     9.09  linear_search
```

Oprofile

Oprofile este un sistem de profiling disponibil ca modul pentru kernel-ul de Linux (i integrat în variantele mai noi ale acestuia), capabil să analizeze atât kernel-ul, cât i aplicaiile utilizator. Folosete tehnica de sampling i se bazează pe informaiile oferite de contoarele din CPU.

Avantaje:

- overhead redus
- profiling al întregului sistem (inclusiv secțiuni "delicate" din kernel)
- prezentarea efectelor la nivel de hardware

Dezavantaje:

- necesita drepturi de utilizator privilegiat (root)
- nu poate fi folosit pentru cod compilat dinamic sau interpretat (Java, Python, etc)

Arhitectura i modul de funcionare

Procesoarele au nite countere speciale pe care le decrementează de fiecare dată când are loc un eveniment de un anumit tip (cache miss, tlb miss, branch miss-predictions, etc.). Când un astfel de counter se ajunge la valoarea zero, se trimite o întrerupere NMI care este interceptată de modulul de kernel al oprofile. Modulul de kernel resetează counterul la o valoare fixată de utilizator i salvează într-un buffer din kernel date despre

locaia care a generat întreruperea (proces/kernel, id-ul threadului, instrucinea curentă care a generat decrementarea counterului, etc.).

Oprofile e format din 3 componente majore:

- **modul de kernel** - un driver special care este notificat de la fiecare întrerupere NMI generată de procesor
- **daemon** - intermediar între modulul kernel i toolurile userspace. Preia date din kernel i le scrie (după procesare) în `/var/lib/oprofile/samples/`.
- **utilitare user space**:
  - `opcontrol` - permite configurarea evenimentelor monitorizate i a frecvenei cu care sunt eantionate, a proceselor pentru care se face monitorizarea, etc.
  - `opreport` - sumar al monitorizării
  - `opannotate` - adnotează pe surse sau pe codul dezasamblat al programului numărul de evenimente care au fost eantionate la fiecare instruciune.

Utilizare

Configurare

Primul pas este verificarea existenei modulului oprofile:

```
muttley:~# modinfo oprofile
```

Apoi este necesară încărcarea efectivă:

```
muttley:~# modprobe oprofile
# sau
muttley:~# opcontrol --init
```

Înainte de a începe colectarea datelor de profiling, trebuie configurat i pornit daemonul `oprofiled`. Pentru a afla informaii despre kernel, `oprofile` are nevoie de calea către imaginea vmlinux (necompresată) a kernelului activ:

```
muttley:~# opcontrol --vmlinux=/boot/vmlinux-`uname -r`
```

Dacă nu se dorete analizarea kernelului, ci doar a unor aplicaii, se folosete:

```
muttley:~# opcontrol --no-vmlinux
```

Pentru a afla tipurile de evenimente disponibile si detalii despre acestea (inclusiv valoarea minima a contorului asociat) se foloseste:

```
student@muttley:~$ opcontrol --list-events
#sau
student@muttley:~$ ophelp
```

`opcontrol` poate monitoriza maximum două tipuri de evenimente în acelai timp (deoarece procesoarele nu au mai multe countere de performană care pot fi folosite simultan). Pentru a alege evenimentele dorite se folosete opiunea `--event`. Sintaxa acesteia este:

```
# opcontrol --event=<TIP_EVENIMENT>:<COUNT>:<UNIT-MASK>:<KERNEL-SPACE-COUNTING>:<USER-SPACE-COUNT
```

Unde:

- TIP_EVENIMENT - reprezintă numărul evenimentului care se dorete monitorizat
- COUNT - reprezintă numărul de evenimente hardware de acest tip după care procesorul emite o întrerupere NMI.
- UNIT-MASK - o valoare numerică ce poate modifica comportamentul pentru counterul curent. Dacă nu este specificat se folosete o valoarea implicită (poate fi determinată cu `ophelp`).
- KERNEL-SPACE-COUNTING - poate lua două valori, 0/1, i specifică dacă se activează sau nu counterul atunci când se rulează cod kernel. Implicit are valoarea 1.
- USER-SPACE-COUNTING - paote lua două valori, 0/1, i specifică dacă se activează sau nu counterul atunci când se rulează cod user. Implicit are valoarea 1.

De exemplu: pe procesoare Core 2 duo, counterul DTLB_MISSES poate fi folosit cu oricare combinaie a următoarelor valori:

- 0x01: ANY Memory accesses that missed the DTLB.
- 0x02: MISS_LD DTLB misses due to load operations.
- 0x04: L0_MISS_LD L0 DTLB misses due to load operations.
- 0x08: MISS_ST TLB misses due to store operations.

```
# se dorete măsurarea toate evenimentele de tip DTLB_MISSES (0xf = 0x1|0x2|0x4|0x8)
# care au avut loc în timp ce rulam cod user space
# ignorând cele care au avut loc în kernel space.
# Se dorete cuantificarea fiecărui al 10000-lea eveniment de acest tip.
muttley:~# opcontrol --event=DTLB_MISSES:10000:0x0f:0:1
```

Pentru a vedea care sunt parametrii cu care este configurat în acest moment oprofile se poate folosi

```
muttley:~# opcontrol --status
Daemon not running
Event 0: PREF_RQSTS_DN:45000:0:1:1
Separate options: library thread
vmlinux file: none
Image filter: /home/gringo/labs/so/profiling/cache_trashing/with_padding
Call-graph depth: 6
Buffer size: 65536
```

Pornire

După configurare, trebuie pornit daemonul:

```
muttley:~# opcontrol --start
```

Oprire

```
muttley:~# opcontrol --shutdown
```

Colectarea i analizarea datelor

Pentru a nu fi influenai de eventuale date rămase de la o rulare anterioară, trebuiesc curaate datele salvate:

```
# se terg toate datele din buffer
muttley:~# opcontrol --reset
#      SAU
# se salvează bufferul curent în sesiunea cu numele ''sesiunea_anterioara''
muttley:~# opcontrol --save=sesiunea_anterioara
```

Odată pornit daemonul, `oprofile` va colecta date despre toate executabilele ce rulează pe acea maină (sau dacă a fost configurat cu `--image` va colecta date doar pentru executabilele din calea specificată). Acum trebuie executat i binarul pe care dorim să îl măsurăm. Deoarece `oprofile` încearcă să-i minimizeze impactul asupra performanelor sistemului, acesta va amâna pe cât posibil scrierea datelor din bufferele interne pe disk. Dacă toolurile userspace `opreport` sau `opannotate` nu găsesc informaii de profiling pe disk vor raporta o eroare:

```
student@muttley:~$ opannotate --source
opannotate error: No sample file found: try running opcontrol --dump or specify a session contain
```

Putem să forăm scrierea pe disc a bufferelor curente cu:

```
student@muttley:~$ opcontrol --dump
```

Exemplu de rulare

Avem următorul program care calculează numărul de numere divizibile cu 2 i cu 3 din intervalul $0..2^{30}$ (exerciiu pur teoretic :).

```
#include <stdio.h>

int main()
{
  int start = 0, stop  = (1 << 30);
  int div2 = 0, div3 = 0;
  int i;

  for (i = start; i < stop; i++) {
    if (i % 2 == 0)
      div2 ++;
    if (i % 3 == 0)
      div3 ++;
  }

  printf("div2=%d, div3=%d\n", div2, div3);
  return 0;
}
```

Dacă rulăm programul obinem:

```
student@muttley:~$ time ./simple
div2=536870912, div3=357913942

real    0m13.828s
user    0m12.525s
sys     0m0.020s
```

Dorim să îmbunătăim performanele programului i îl paralelizăm.

O variantă ar fi să incrementăm variabilele `div2` i `div3` din două threaduri diferite protejând accesul la variabilele `div2` i `div3` cu un mutex. Această implementare va fi mult mai ineficientă decât cea prezentată mai sus, majoritatea timpului de rulare fiind petrecut în rutinele de preluare i eliberare a mutexului.

O paralelizare mai bună se poate implementa folosind variabile separate pentru fiecare thread i integrarea rezultatelor pariale la final (folosind o paradigma de programare de tipul map-reduce).

```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

#define BIGNUM           (1 << 30) // limita superioară a intervalului de prelucrare
#define NR_CPUS           2
#define NR_THREADS        NR_CPUS
#define PER_CPU_INTERVAL (BIGNUM / NR_CPUS) //lungimea intervalului pe care va căuta fiecare thre

struct thd_data {
  int div2;
  int div3;
};

struct thd_data per_cpu_vec[NR_THREADS];
pthread_t threads[NR_THREADS];

/**
 * Se impune threadului `thdid` să ruleze doar pe procesorul `thdid % NR_CPUS`.
 * Dacă există procesoare >= threaduri, fiecare thread va rula pe un procesor separat.
 */
static void set_affinity(int thdid)
{
  int rc;
  cpu_set_t cmask;
  CPU_ZERO(&cmask);
  CPU_SET(thdid % NR_CPUS, &cmask);
  rc = sched_setaffinity(0 /*current thead*/, sizeof(cpu_set_t), &cmask);
  if (-1 == rc)
    perror("sched_setaffinity error");
}


static void * thd_func(void *param)
{
  int thdid = (int)param;
  int i;
  /* Fiecare thread operează pe o poriune distinctă a intervalului. */
  int start = thdid       * PER_CPU_INTERVAL;
  int stop  = (thdid + 1) * PER_CPU_INTERVAL;

  set_affinity(thdid);


  for (i = start; i < stop; i++) {
    if (i % 2 == 0)
      per_cpu_vec[thdid].div2 ++;
    if (i % 3 == 0)
      per_cpu_vec[thdid].div3 ++;
```

```
  }

  return NULL;
}


int main()
{
  int rc, i, sum;

  /* se creează threadurile */
  for (i = 0; i < NR_THREADS; i++) {
    rc = pthread_create(&threads[i], NULL, thd_func, (void*)i);
    if (-1 == rc) {
      perror("error creating thread");
      exit(-1);
    }
  }


  /* se ateaptă ca toate threadurile să-i termine execuia */
  for (i = 0; i < NR_THREADS; i++) {
    rc = pthread_join(threads[i], NULL);
    if (-1 == rc) {
      perror("error joining thread");
      exit(-1);
    }
  }


  /* se integrează rezultatele pariale ale fiecărui thread */
  sum = 0;
  for (i = 0; i < NR_THREADS; i++) {
    sum += per_cpu_vec[i].div2;
  }
  printf("div2=%d, ", sum);

  sum = 0;
  for (i = 0; i < NR_THREADS; i++) {
    sum += per_cpu_vec[i].div3;
  }
  printf("div3=%d\n", sum);
  return 0;
}
```

Programul paralelizat este mai complex, performanele lui fiind:

```
student@muttley:~$ time ./complex
div2=536870912, div3=357913942

real    0m38.865s
user    1m14.913s
sys     0m0.016s
```

Pentru programul paralelizat

- timpul total de rulare este cu 350% peste cel al variantei neparalelizate.
- timpul de procesor este cu 700% peste cel al variantei neparalelizate, folosind 2 core-uri în acelai timp

Intuitiv timpul de procesor ar fi trebuit să crească, dar timpul de rulare ar fi trebuit să scadă. Vom investiga problema folosind oprofile.

Partea computaională (for-ul) folosete puine date i ar trebui să încapă în cache-ul L1. Vom verifica dacă se fac accese în cache-ul L2.

```
# se terg datele unei rulări anterioare
muttley:~# opcontrol --reset
# se monitorizează toate cererile în cache-ul L2 (L2_RQSTS) făcute de executabilul "complex"
muttley:~# opcontrol --no-vmlinux --image=./complex --event=L2_RQSTS:7500
# se pornete serverul oprofiled
muttley:~# opcontrol --start
# se rulează aplicaia monitorizată
muttley:~# ./complex
div2=536870912, div3=357913942
# se închide serverul
muttley:~# opcontrol --dump; opcontrol --shutdown

# se preiau datele
muttley:~# opreport
CPU: Core 2, speed 1833 MHz (estimated)
Counted L2_RQSTS events (number of L2 cache requests) with a unit mask of 0x7f (multiple flags) c
   L2_RQSTS:90000|
  samples|      %|
------------------
     1660 100.000 p1


# se afiează informaii per-simbol
muttley:~# opreport --symbols
CPU: Core 2, speed 1833 MHz (estimated)
Counted L2_RQSTS events (number of L2 cache requests) with a unit mask of 0x7f (multiple flags) c
samples  %        symbol name
1660     100.000  thd_func


# se afiează detalii per instruciune i se afiează informaii despre fiierul i linia din care provi
muttley:~# opreport --debug-info --details
CPU: Core 2, speed 1833 MHz (estimated)
Counted L2_RQSTS events (number of L2 cache requests) with a unit mask of 0x7f (multiple flags) c
vma      samples  %        linenr info                 symbol name
08048600 1660     100.000  cache_trashing.c:40         thd_func
  0804863a 5         0.3012  cache_trashing.c:51
  08048648 597      35.9639  cache_trashing.c:52 --       per_cpu_vec[thdid].div2 ++;
  08048658 1         0.0602  cache_trashing.c:53
  0804865f 8         0.4819  cache_trashing.c:53
  08048681 2         0.1205  cache_trashing.c:53
  08048684 1         0.0602  cache_trashing.c:53
  08048694 1044     62.8916  cache_trashing.c:54 --       per_cpu_vec[thdid].div3 ++;
  0804869e 1         0.0602  cache_trashing.c:50
  080486a2 1         0.0602  cache_trashing.c:50


# se adnotează pe codul sursă date de profiling
muttley:~# opannotate --source
...
     2  0.1205 :  for (i = start; i < stop; i++) {
     5  0.3012 :    if (i % 2 == 0)
   597 35.9639 :      per_cpu_vec[thdid].div2 ++;
    12  0.7229 :    if (i % 3 == 0)
  1044 62.8916 :      per_cpu_vec[thdid].div3 ++;
```

```
...


# se adnotează pe codul dezasamblat al programului date de profiling
muttley:~# opannotate --assembly
...
     5  0.3012 : 804863a:  test   %eax,%eax
                : 804863c:       jne    8048652 <thd_func+0x52>
                : 804863e:       mov    (%ebx,%eax,4),%eax
                : 8048641:       mov    0x8049a48(,%eax,8),%edx
   597 35.9639 : 8048648:       add    $0x1,%edx
                : 804864b:       mov    %edx,0x8049a44(,%eax,8)
                : 8048652:       mov    0x8(%ebp),%eax
                : 8048655:       mov    -0x18(%ebp),%edx
     1  0.0602 : 8048658:       movl   $0x55555555,0x1c(%ebp)
     8  0.4819 : 804865f:       mov    %edx,(%eax)
...
     2  0.1205 : 8048681:       mov    -0x14(%ebp),%edx
     1  0.0602 : 8048684:       cmpl   $0x0,0x14(%ebp)
                : 8048688:       jne    8048648 <thd_func+0x9e>
                : 804868a:       mov    (%ebx,%eax,4),%eax
                : 804868d:       mov    0x8049a48(,%eax,8),%edx
  1044 62.8916 : 8048694:       add    $0x1,%edx
                : 8048697:       mov    %edx,0x8049a48(,%eax,8)
     1  0.0602 : 804869e:       addl   $0x1,-0x8(%ebp)
     1  0.0602 : 80486a2:       mov    -0x8(%ebp),%eax
```

Din rezultatele monitorizării se observă că se fac un număr mare de accese în cache-ul L2 când se accesează datele per-cpu, dei datele în loop încap în cache-ul L1.

```
struct thd_data {
  int div2;
  int div3;
};
struct thd_data per_cpu_vec[NR_THREADS];
```

Procesorul pe care a fost rulat exemplul anterior deine câte un cache L1 pentru fiecare core i un cache L2 partajat. Dacă mai multe core-uri modifică date care corespund unei aceleiai linii din cache-ul L2, de fiecare dată când unul din core-uri scrie în acea linie, linia va fi invalidată în cache-ul L1 al celorlalte core-uri. Astfel celelalte threaduri vor trebui să recitească din L2 toată linia curentă, cache-ul L1 ne mai ajutând în acest caz, toate acesele făcându-se la o viteză mai mică chiar decât cea a cache-ului L2.

Dacă se modifică structura de date specifică fiecărui thread, se poate elimina problema de "false sharing".

```
struct thd_data {
  int div2;
  int div3;
  char padding[NR_PADDING_BYTES];
};
struct thd_data per_cpu_vec[NR_THREADS];

muttley:~# time ./with_padding
div2=536870912, div3=357913942
real    0m6.432s
user    0m12.793s
sys     0m0.008s


muttley:~# opreport
```

```
CPU: Core 2, speed 1833 MHz (estimated)
Counted L2_RQSTS events (number of L2 cache requests) with a unit mask of 0x7f (multiple flags) c
    L2_RQSTS:7500|
  samples|      %|
------------------
        1 100.000 with_padding
```

În acest caz a avut loc doar un singur eantion de citire în L2 (în realitate au avut loc între 7500 i 2*7500-1 evenimente de acest tip).

- timpul total de rulare este 58% din cel al variantei neparalelizate
- timpul de procesor este 112% din cel al variantei neparalelizate.

Windows

Kernrate si KrView

Kernrate este un echivalent al oprofile pentru Windows. [1]

XPerf

[2]

Exerciii

Prezentare

Pentru a urmări mai uor noiunile expuse la începutul laboratorului folosii această prezentare (pdf) (odp).

Exerciii de laborator

- Folosii arhiva de resurse a laboratorului.
- Putei folosi fiierele tags pentru o parcurgere rapidă a surselor folosind vim (fiierul tags) sau Emacs (fiierul TAGS).

1. **(2 puncte)** Intrai în directorul `01-bubble-sort/`.
   - Analizai coninutul fiierului `bubble-sort.c`.
   - Completai funcia `bubble_sort` cu o implementare de Bubble sort.
   - Comparai, folosind `gprof`, rezultatele obinute în momentul utilizării funciei `bubble_sort` implementate în C i funciei `bubble_sort` implementate în assembly.

- Folosii **_apoi_** flag-ul `-O2` pentru a compila sursa folosind suportul de optimizări ale compilatorului i comparai rezultatele de profiling obinute cu cele anterioare.
- Care sunt pările din cod peste care se pierde mult timp?
- **Hints**:
  - Folosii funcia swap pentru interschimbarea a două valori
  - Folosii opiunea `-lb` a `gprof` pentru informaii sumare despre timpul petrecut în diversele zone de cod.

2. **(2 puncte)** Intrai în directorul `02-major/`.
   - Scriei 2 programe prin care să determinai, cu ajutorul `oprofile`, dacă limbajul C este column-major sau row-major (putei citi mai multe aici).
   - **Hints:**
     - Un program va completa o matrice iterând pe linii, celălalt pe coloane.
     - Folosii evenimentul `BSQ_CACHE_REFERENCE` cu masca `0x07`.
     - Folosii `opcontrol --image=program-name` pentru a urmări doar evenimentele care au loc când se rulează cod din programul `program-name`.
     - Folosii `opreport` pentru a afia informaii de debug detaliate
     - Nu uitai să tergei mesajele de log de la precedenta rulare a `oprofile`
3. **(2 puncte)** Intrai în directorul `03-find-char/`.
   - Analizai coninutul fiierului `find-char.c`.
   - Compilai fiierul `find-char.c` i rulai executabilul obinut.
   - Identificai, folosind `gprof`, care este funcia care ocupă cel mai mult timp de procesor i poate îmbunătăii performanele programului.
4. **(2,5 puncte)** Intrai în directorul `04-tlb/`.
   - Analizai coninutul fiierelor `tlbp.c`, respectiv `tlbt.c`. Ce se realizează în cadrul celor două programe?
   - Compilai cele două programe.
   - Contorizai timpul de execuie folosind `time`. Care program se rulează mai repede?
   - Folosii `oprofile` pentru a determina în rularea cărui program se obin cele mai multe TLB miss-uri.
   - **Hints**:
     - Folosii `ophelp` pentru a determina evenimentul ce poate fi folosit pentru detectarea evenimentelor de tip TLB miss.
     - Folosii `opcontrol --image=program-name` pentru a urmări doar evenimentele care au loc când se rulează cod din programul `program-name`.
5. **(2,5 puncte)** Intrai în directorul `05-hash/`.
   - În directorul `05-hash/` se găsete o implementare a unui algoritm pentru tabele distribuite scrisă de Alexandru pentru tema de la CPL.
   - Dei dimensiunea tabelului este dublă faă de numărul de cuvinte, programul are o comportare ineficientă. De ce? Reparai greeala. (Google is your best friend).

Soluii

Resurse utile

- Intel 64 and IA-32 Architectures Optimization Reference Manual Appendix B
- Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B Appendix A
- False sharing - [3]

- Documentatie kernrate
- gprof manual
- gcov manual