Măsurători

Mihai Budiu — mihaib+@cs.cmu.edu
http://www.cs.cmu.edu/~mihaib/

16 septembrie 1998

Subiect: Măsurători de performanţă

Cunoştinţe necesare: Noţiuni elementare de sisteme de operare, programare în C, noţiuni elementare despre limbaje de asamblare

Cuvinte cheie: timp, cicli pe instrucţiune, cache

Cuprins

Acest articol va construi un mic set de rutine pentru a măsura durata execuţiei unor bucăţele de program după care va folosi aceste rutine pentru a estima tot felul de parametri arhitecturali. Am spart articolul în două bucăţi: în prima voi discuta despre măsurători în general; apoi voi construi infrastructura necesară măsurătorilor, după care arăt cum putem măsura anumiţi parametri ai calculatorului. În a doua parte a articolului urmează să fac o muncă "detectivistă" care arată cum se pot folosi măsurătorile de performanţă pentru a trage concluzii despre anumiţi parametri arhitecturali care influenţează viteza doar în mod indirect. Ca exemplu, intenţionez să ilustrez cum se pot face tot felul de măsurători ale caracteristicilor cache-ului microprocesorului Pentium II. Majoritatea ideilor şi codului din acest articol au fost dezvoltate ca teme la cursul de arhitectura calculatoarelor pe care autorul l-a luat în toamna anului 1997 la CMU. Creditul pentru aceste idei i se cuvine unuia dintre profesori, Randy Bryant.

1 Introducere: măsurători de performanţă

Într-un articol anterior, despre natura cercetării în "sisteme" (PC Report din martie 1998), discutam despre importanţa evaluării programelor. Majoritatea articolelor de cercetare din domeniu conţin măsurători ample, care controlează feluriţi parametri şi studiază variaţia altora. Măsurarea unui sistem complex este foarte dificilă, şi există o sumedenie întreagă de tehnici care se pot utiliza. Importanţa măsurătorilor a fost recunoscută şi de proiectanţii de microprocesoare: toate procesoarele moderne oferă suport extins pentru măsurarea a tot felul de evenimente importante. Un articol recent din PC Report ("Tehnologia MMX, partea a II-a", Iulie 1998, de Valer Bocan) arăta lista tuturor măsurătorilor pentru care procesorul Pentium are suport din fabricaţie: o pagină întreagă.

Există foarte multe feluri de măsurători; o privire superficială poate distinge următoarele clasificări:

Măsurători statistice vs. exacte. Numărul de evenimente petrecute la rularea unui program este uriaşă; contorizarea fiecărui eveniment poate genera o cantitate enormă de informaţie, a cărei colectare poate interfera cu programul însuşi.

Probabil cele mai frecvente măsurători sunt cele făcute pentru a vedea unde un program îşi petrece cel mai mult timp. Acest tip de măsurătoare se numeşte *profiling*. Metoda obişnuită pentru astfel de măsurători este de a genera periodic întreruperi şi de a marca locul execuţiei programului în clipa întreruperii. Acesta este un exemplu tipic de măsurătoare statistică. Compilatoarele de C generează cod cu măsurători prin folosirea opţiunii `-p`. Programe ca `prof` şi `gprof` sunt folosite pentru a analiza rezultatele.

Măsurători reale vs. simulări. Unul din scopurile măsurătorilor este de a vedea dacă o nouă idee oferă într-adevăr mai multă eficienţă decît metodele tradiţionale. Dar, mai ales cînd este vorba de proiectarea de hardware, un sistem nu poate fi construit pînă nu suntem siguri că este bun; de asta nu putem fi siguri însă pînă cînd nu l-am măsurat. Soluţia tradiţională este de a construi un *simulator* al sistemului şi de a măsura parametrii de interes cu acesta.

Este inutil de zis că a construi un simulator foarte precis este o treabă extrem de complicată. Se întîmplă adesea ca unele dintre fenomenele care nu sunt luate în considerare de către simulare să aibă în realitate un impact foarte important asupra performanţei.

Măsurători cu instrumente externe vs. sisteme auto-instrumentate. În mod ideal măsurătoarea nu trebuie să disturbe sistemul însuşi de loc; în realitate lucrurile stau altfel. O măsurătoare care nu interferă foloseşte analizoare de magistrală (care înregistrează tot traficul pe magistrala calculatorului între procesoare, controlere şi memorie) sau capturează pachetele dintr-o reţea. O problemă cu astfel de măsurători este cantitatea enormă de informaţie pe care o generează şi care trebuie stocată pentru prelucrări ulterioare.

Pe de altă parte, atunci cînd măsurăm un program, putem modifica programul însuşi pentru a înregistra chiar el numai evenimentele importante. Problema este că aceste modificări afectează programul însuşi; nu numai din cauză că strîngerea informaţiei ia timp, dar şi pentru că aşezarea codului şi datelor în memorie se schimbă, de unde tot felul de efecte noi în cache-uri, în interferenţe cu memoria virtuală, etc.

Măsurători sintetice vs. măsurători în condiţii "reale". Cînd măsurăm un sistem, cu ce fel de date de intrare trebuie să facem rulările? Care sunt datele "cele mai tipice"? Putem distinge la marginile unui spectru larg teste complet sintetice care exersează un număr foarte mic de parametri, şi teste extrem de complexe, cu programe sau date "reale", folosite în medii industriale.

Înainte de a plonja în concret să menţionăm şi faptul că interpretarea rezultatelor testelor este adesea foarte dificilă. După cum vom vedea în partea a doua a acestui articol, mai multe fenomene pot da naştere unor efecte similare, iar atunci cînd putem avea de-a face cu interferenţa mai multor fenomene este foarte greu de depistat cauza reală a unor rezultate.

Partea I

Măsurători directe

2 O procedură pentru măsurarea timpului în Unix

După ce am vorbit despre tot felul de metode, care mai de care mai sofisticate, voi ilustra în acest articol o metodă foarte simplă de auto-instrumentare a unui program, care poate fi folosită pentru a măsura destul de precis chiar evenimente foarte scurte.

Măsurarea timpului este o treabă a sistemului de operare, aşa că avem nevoie de un oarecare suport din partea acestuia. Voi folosi pentru exemplificare sistemul de operare Unix; programele din acest text au fost testate pe mai multe sisteme, incluzînd Linux, SunOs, Solaris şi Digital Unix. Metodologia poate fi probabil însă folosită şi pe sisteme din familia Windows, în măsura în care contabilizarea timpului pentru programe se face într-un fel asemănător (acest subiect îmi este, din păcate, străin).

2.1 Funcţiile `setitimer` şi `getitimer`

Nucleul unui sistem Unix menţine pentru fiecare proces trei contoare independente, numite "ceasuri de interval" (interval timers). Utilizatorul are la dispoziţie două funcţii de bibliotecă pentru a citi şi scrie aceste ceasuri; scrierea se face cu `setitimer()` iar citirea cu `getitimer()`.

Timpul însuşi este reprezentat într-o structură care se numeşte `struct timeval`. Pe sistemele citate anterior structura aceasta este foarte simplă:

```
struct timeval {
   long  tv_sec;        /* secunde */
   long  tv_usec;       /* microsecunde */
};
```

Un "itimer" (interval timer) are două astfel de structuri înauntru:

```
struct  itimerval {
   struct  timeval it_interval; /* timer interval */
   struct  timeval it_value;    /* current value */
};
```

1. o structură numită `it_interval`, care conţine perioada cu care ceasul trebuie să "bată".

2. o structură numită `it_value` care arată cît timp mai trebuie să treacă înainte de următoarea bătaie a ceasului.

Cînd un astfel de ceas "expiră", nucleul trimite procesului în cauză un semnal. Din fericire pe noi nu ne vor interesa semnalele: vom folosi aceste ceasuri într-un mod mai simplu, pentru a măsura durata unor evenimente: setăm valoarea curentă a lui `it_value`, executăm codul de măsurat, şi apoi citim `it_value` din nou; făcînd diferenţa vom afla cît a durat execuţia.

2.2 Timp real şi timp virtual

Unix este un sistem de operare multitasking: mai multe procese se pot executa "simultan". De obicei nu avem la dispoziţie tot atîtea procesoare cît procese de executat, aşa că modul în care mai multe programe îşi "împart" calculatorul este prin *time-sharing*: o bucăţică de timp se execută unul, apoi programul este suspendat şi se execută un al doilea, etc.

Nucleul oferă deci două feluri de timere pentru procese (de fapt trei, dar numai două ne interesează pe noi acum): unul care măsoară timpul real scurs (numit şi timpul ceasului de pe perete: "wall-clock time") şi unul care măsoară timpul "virtual": numai timpul cît procesul măsurat a fost în execuţie.

Noi vom folosi ambele timere, în funcţie de ceea ce vrem să măsurăm.

2.3 Măsurarea timpului

Vom scrie trei funcţii de bază care manipulează aceste timere; iată declaraţiile lor în fişierul `etime.h`:

```
/* fisierul etime.h */

#ifndef _ETIME_H
#define _ETIME_H

void init_etime(void);  /* initializeaza ceasurile */
double get_etime(void); /* calculeaza timpul virtual scurs de la
                           initializarea ceasurilor, in secunde */
double get_ewtime(void);/* timpul real scurs de la initializarea
                           ceasurilor */
#endif
```

Aceste funcţii sunt relativ simplu de scris, dacă citim cu atenţie explicaţiile din manual ale funcţiilor `getitimer` şi `setitimer`. Am presupus că nici unul dintre programele măsurate nu se va executa mai mult de o zi, şi de aceea am pus lungimea intervalului de măsurat la 24 * 60 * 60 = 86400 secunde. Iată codul:

```
/* fisierul etime.c: manipulare de timere */

#include "etime.h"
#include <stdio.h>
#include <sys/time.h>

/* memoreaza timpul virtual ramas */
static struct itimerval init_virtual;

/* timpul real ramas */
static struct itimerval init_real;

#define MAX_ETIME 86400   /* o zi = un interval foarte mare. */
```

```
void init_etime(void)
{
    init_virtual.it_interval.tv_sec = 0;
    init_virtual.it_interval.tv_usec = 0;
    init_virtual.it_value.tv_sec = MAX_ETIME;
    init_virtual.it_value.tv_usec = 0;
    setitimer(ITIMER_VIRTUAL, &init_virtual, NULL);
    /* ultimul argument al lui setitimer citeste valoarea
       timerului inainte de a fi modificat; nu ne intereseaza,
       deci folosim NULL */

    init_real.it_interval.tv_sec = 0;
    init_real.it_interval.tv_usec = 0;
    init_real.it_value.tv_sec = MAX_ETIME;
    init_real.it_value.tv_usec = 0;
    setitimer(ITIMER_REAL, &init_real, NULL);
}

double get_etime(void)
        /* elapsed time: timp scurs */
{
    struct itimerval curent;

    getitimer(ITIMER_VIRTUAL, &curent);

    /* cite secunde au trecut de cind am setat init_virtual? */
    return (double)((init_virtual.it_value.tv_sec -
                     curent.it_value.tv_sec) +
                    (init_virtual.it_value.tv_usec -
                     curent.it_value.tv_usec) * 1e-6);
    /* 1e-6 = marimea unei microsecunde in secunde. */
}

double get_ewtime(void)
        /* elapsed wall time: timp scurs real */
{
    struct itimerval curent;

    getitimer(ITIMER_REAL, &curent);
    return (double) ((init_real.it_value.tv_sec - curent.it_value.tv_sec) +
                     (init_real.it_value.tv_usec - curent.it_value.tv_usec)*1e-6);
}
```

Deşi sunt multe litere, ce se întîmplă ar trebui să fie evident. Observaţi că ceasurile numără "în jos" (descresc de la valoarea iniţială spre 0), şi de aceea scad valoarea curentă din cea iniţială şi nu invers.

2.4 Măsurarea rezoluţiei ceasului

Buun. Avem deci o metodă pentru a măsura cît timp a trecut. Primul lucru pe care o să-l facem este să vedem ce *rezoluţie* are timerul însuşi. Cu alte cuvinte: cu ce frecvenţă bate ceasul? Nu ne putem aştepta ca valoarea timer-ului să fie ajustată foarte frecvent, pentru că aceasta ar lua foarte mulţi cicli din execuţia programelor însele.

Pentru a măsura rezoluţia ceasului ne vom folosi de faptul că **ceasul se mişcă relativ încet**. Mult mai încet decît durata executării unei funcţii sau a unui apel de sistem. Cu alte cuvinte, pot executa de multe ori funcţia `get_etime` între două bătăi ale ceasului. Acest lucru este fundamental pentru buna funcţionare a programelor de măsurare pe care le scriem, şi îl vom verifica puţin mai tîrziu. În orice caz, ştim din folclor (sau, dacă nu, din alte articole din PC Report) că un apel de sistem pe o maşină modernă durează de ordinul zecilor de microsecunde, pe cînd rezoluţia ceasului este în zona milisecundelor (de sute de ori mai mică).

Vom scrie deci un program foarte simplu care nu face decît să citească ceasul în mod repetat pînă sesizează o schimbare. Din cauză că citirile vor fi mult mai dese decît ticăirile ceasului, suntem siguri că vom vedea cu cît se schimbă ceasul la fiecare bătaie. Funcţia `delta()` calculează rezoluţia ceasului:

```
/* functia delta(): masuratoarea rezolutiei ceasului SO */

double delta(void)
{
    double curent, inainte,
           mindelta = 10;       /* valoare ridicol de mare */
    int repetam = 3;   /* masuram de mai multe ori, ca sa fim siguri */

    inainte = get_etime();
    while (repetam) {
        curent = get_etime();
        if ((curent - inainte) > 1e-6) {
            repetam--;
            if ((curent - inainte) < mindelta)
                /* avem o estimare mai buna pentru rezolutie */
                mindelta = curent - inainte;
            inainte = curent;
        }
    }
    return mindelta;
}
```

(Observaţi metoda de a compara valori `double`). Încercaţi-o:

```
/* exemplu: masurarea delta() */

#include <stdio.h>
```

```
#include "etime.h"

double delta(void);

int main(void)
{
    init_etime();
    printf("Rezolutia ceasului este %2.3fms\n", delta()*1000);
    return 0;
}
```

2.5 Măsurarea unor evenimente foarte scurte

Ce ne facem însă cu intervalele foarte scurte? Cum poţi să măsori un eveniment care durează o microsecundă dacă ai un ceas care are ca unitate de măsură de 10ms? Din fericire nu e prea greu: un calculator e foarte bun la a repeta aceeaşi operaţiune de milioane de ori. Ce trebuie să facem este să executăm operaţia de măsurat de — să zicem — un milion de ori, şi să împărţim rezultatul măsurătorii la un milion.

De fapt trebuie să fim ceva mai grijulii, din două motive:

- Măsurarea unei operaţiuni de o secundă ar dura prea mult în acest fel, de aceea vom face calculatorul să măsoare atîta timp cît îi trebuie pentru a fi garanta că este suficient de riguros. Dacă o măsurătoare ia în jur de o secundă, atunci ea nu mai trebuie repetată de loc.

- Trebuie să fim atenţi cînd măsurăm timpul *virtual*, (pentru că în interiorul unei secunde procesul ar putea fi întrerupt de mai multe ori din cauza time-sharing) şi cînd timpul *real* (pentru că asta vrem de fapt să aflăm). Trebuie să cronometrăm numai "timpul efectiv de joc".

Acestea fiind zise, vom purcede la a scrie o funcţie generică numită `ftime` care are trei argumente:

- Un pointer spre o altă funcţie, a cărei durată trebuie măsurată;

- Precizia cerută, în procente. Eroarea măsurătorii va fi limitată de această valoare;

- Un boolean care spune dacă măsurăm timp virtual (0) sau real (1).

Pentru a limita eroarea, trebuie să facem un pic de aritmetică. Să presupunem că vrem ca eroarea să fie sub 1 procent. Atunci dacă ceasul bate cu perioada de X, trebuie să măsurăm numai durate de cel puţin 101X. De ce? Orice măsurătoare este eronată cu cel mult +/−X, deci eroarea va fi sub 1 procent.

Mai general, dacă vrem o eroare mărginită de ϵ procente şi rezoluţia ceasului este Δ atunci trebuie să rulăm testul timp de $(1+\epsilon)\Delta/\epsilon$. Formula nu e prea greu de demonstrat riguros, dar datorită formatului PC Report formulele matematice sunt ceva mai greu de tehnoredactat. Voi adăuga demonstraţia ca anexă în varianta articolului care se găseşte în pagina mea de web.

```
/* functia ftime: masuratoarea duratei unei alte functii */

#include "etime.h"

/* puneti functia delta() aici */

double ftime(void (*de_masurat)(void), double epsilon, int timp_real)
{
    double durata;        /* cit timp trebuie executat testul */
    unsigned long repetitii;
    double inceput, sfirsit;
    static double Delta;
    static int initializat = 0;
    double (*masoara)(void);

    masoara = timp_real ? get_ewtime : get_etime;

    if (!initializat) {
        /* prima oara calculam Delta */
        initializat = 1;
        Delta = delta();
    }

    durata = (1 + epsilon) * Delta / epsilon;

    for (repetitii = 1; ; repetitii *= 2) {
        int i;
        inceput = masoara();
        for (i=repetitii; i > 0; i--)
            de_masurat();
        sfirsit = masoara();
        if (sfirsit - inceput >= durata) break;
    }
    return ((sfirsit - inceput) / (double)repetitii);
}
```

Funcţia este destul de simplă: va dubla numărul de repetiţii al executării funcţiei de măsurat pînă cînd în total repetiţiile depăşesc suficient durata necesară pentru a mărgini eroarea. Dacă vă îngrijorează numărul mare de execuţii al buclei, observaţi că numărul de execuţii al funcţiei de măsurat descrie o serie geometrică (se execută de 1, 2, 4, 8, etc. ori), deci numărul total de execuţii nu va depăşi niciodată 2*`durata`, unde `durata` este durata necesară pentru a limita eroarea (şi asta e uşor de demonstrat). Pentru calculatorul meu Δ este 10ms, deci pentru o eroare de 1% `durata` este 1s. Durata măsurătorii nu va depăşi deci 2 secunde (decît dacă funcţia de măsurat durează mai mult).

Ştiu că vă grăbiţi să încercaţi codul, aşa că daţi-i drumul:

```
/* prima masuratoare */
```

```
#include <stdio.h>
#include "etime.h"

/* puneti aici delta() si ftime() */

void f(void)
{
   /* scrieti aici codul de masurat, ce vreti */
}

int main(void)
{
    init_etime();
    printf("Durata executarii lui f este %2.2fsec\n", ftime(f, 0.01, 0));
    return 0;
}
```

2.6 Surse de eroare

Din păcate am văzut (din fugă) că un sistem care se auto-măsoară este supus la perturbaţii (un fel de principii de incertitudine à la Heisenberg). Nici programul nostru nu este scutit de astfel de probleme. Nu putem face foarte mult pentru a le corecta, dar este foarte bine să le avem în vedere. Iată unele posibile interferenţe:

Codul suplimentar: observaţi că ceea ce măsurăm este tot codul executat între atribuirea lui `inceput` şi atribuirea lui `sfirsit`, din funcţia `ftime()`. Asta înseamnă, pe lîngă corpul funcţiei `de_masurat` şi ceva din ciclul `for`, plus codul care cheamă funcţia şi execută întoarcerea. De asemenea, vom vedea că uneori va trebui să înghesuim în corpul funcţiei de măsurat cod suplimentar, pe care de fapt nu l-am vrea măsurat (de pildă pentru că funcţia de măsurat nu poate avea argumente aşa cum am scris-o). Aceste erori sunt însă mărunte; eroarea relativă depinde, desigur, de cît de mult durează funcţia de măsurat însăşi.

Execuţia în interiorul nucleului: o problemă mult mai gravă este faptul că contabilizarea timpului de către nucleu nu este întotdeauna foarte precisă. Nucleul trebuie să execute tot felul de funcţii pe care utilizatorul nu le-a chemat niciodată (de pildă funcţia care comută de la un proces la altul). Timpul acestor funcţii este atribuit pînă la urmă tot unor procese utilizator.

Sau să luăm un alt caz: apariţia unui pachet pe reţea. Nucleul trebuie să citească pachetul şi probabil să-l trimită pe o altă interfaţă, sau să-l livreze unui proces. Dar din momentul apariţiei pachetului pînă la livrare nu este încă evident cui îi este destinat acesta (poate unei alte maşini). Ori timpul de procesare al pachetului trebuie contabilizat şi el undeva. În general toate întreruperile (de disc, de la tastatură, etc.) suferă de problema asta.

Din această cauză, măsurătorile vor fi mai precise dacă maşina nu este deloc încărcată (nu execută prea multe procese) şi nu este conectată la reţea. Cît de mare este eroarea vă invit să experimentaţi făcînd măsurători pe două maşini: una aglomerată şi una liberă.

Efecte de cache: Un alt tip de interferenţă, foarte spectaculos, este interferenţa în cache a codului care măsoară cu cel măsurat. Vom vedea în partea a doua a articolului, cînd vom măsura parametrii cache-ului, că datele şi codul îşi împart cache-ul numit L2; din cauza asta pentru anumite plasamente ale datelor am putea avea conflicte între date şi codul însuşi, conflicte care ar putea să dispară în alte configuraţii! Astfel de efecte sunt extrem de dificil de controlat şi mărginit.

Conflicte de cache pot fi şi între sistemul de operare şi programul de măsurat etc. O cutie a Pandorei, ce mai.

3 Măsurători de performanţă

Hai să exersăm acum procedurile de mai sus pe nişte operaţiuni interesante. Vom măsura pentru început cea mai scurtă durată din calculator: frecvenţa de tact a microprocesorului! Aceasta este efectiv uriaşă: un procesor la 333Mhz are un ceas de 3ns, de 3 milioane de ori mai mică decît rezoluţia Delta (10ms) a ceasului!

3.1 Frecvenţa ceasului procesorului

Cum putem să măsurăm o durată atît de scurtă? În primul rînd, cum putem scrie un program atît de scurt? Nici nu putem. Dar vom face acelaşi truc: vom repeta aceeaşi instrucţiune foarte scurtă de multe ori.

Ce instrucţiune durează exact un ciclu de ceas? Practic, pe toate procesoarele RISC sau CISC o adunare de numere întregi durează exact un ciclu de ceas.

Va trebui să fim atenţi însă cînd scriem codul la următoarele aspecte:

1. Procedura cu adunările va trebui să fie compilată *cu optimizări* de către compilator. Codul în asamblare rezultat nu trebuie să conţină nici un fel de instrucţiuni în plus în afară de adunări, pentru că altfel nu măsurăm ce trebuie;

2. Din cauza asta nu putem folosi de fapt o buclă: instrucţiunile de incrementare şi test iau la fel de mult ca adunările însele. Va trebui să scriem cu mîna o mulţime de adunări;

3. Procesoarele moderne superscalare pot executa mai multe adunări simultan; pentru a evita acest lucru vom face ca fiecare adunare să folosească rezultatul celei anterioare (asta se numeşte o *dependenţă*; dependenţele în programe sunt unul dintre obstacolele majore împotriva creşterii performanţei. De data asta vor fi în folosul nostru);

4. Codul nu trebuie să fie optimizabil în aşa fel încît să fie simplificat prea tare (*dead-code ellimination*). De pildă, dacă facem numai adunări, dar nu facem nimic cu rezultatul, un compilator deştept s-ar putea prinde şi ar putea elimina toate adunările din program.

```
/* masurarea ceasului microprocesorului */

int ax;  /* niste variabile globale pentru a impiedica optimizari */
int bx;
int cx;

/* niste macrouri simpatice pentru a genera 100 de repetitii */
#define DOI(x) x x
```

```
#define PATRU(x) DOI(x) DOI(x)
#define CINCI(x) x PATRU(x)
#define DOUASCINCI(x) CINCI(CINCI(x))
#define OSUTA(x) PATRU(DOUASCINCI(x))

void add_test(void)
{
    register int a = ax;
    register int b = bx;
    register int c = cx;

    OSUTA(a += b; a += c;)      /* doua adunari dependente! */

    ax = a+b+c;     /* pentru ca scriem intr-o variabila globala
                       compilatorul nu va scoate nici o adunare
                       din program */
}

void add_dummy(void)
{
    register int a = ax;
    register int b = bx;
    register int c = cx;
    ax = a+b+c;
}

double mhz(double epsilon)
{
    double secs;

    secs = ftime(add_test, epsilon, 0) - ftime(add_dummy, epsilon, 0);
    return (200.0 / secs) / 1e6; /* add_test are 200 instr.
                                    mai mult decit add_dummy */
}
```

Am fost foarte atent să compensez diferenţa între costul celor 200 de instrucţiuni de adunare şi instrucţiunile suplimentare, calculînd durata unei proceduri `add_dummy`, care conţine numai codul parazit şi scăzînd cele două valori. Am testat codul ăsta pe o maşină rapidă: un Pentium II la 266Mhz astfel:

```
#include <stdio.h>
#include "etime.h"

/* puneti aici ftime(), delta(), mhz() si prietenii lor */

int main(void)
```

```
{
    init_etime();
    printf("Frecventa ceasului este %gMhz\n", mhz(0.02));
    return 0;
}
```

Cu o eroare tolerată de 2% am obţinut răspunsul: 266.3MHz. Destul de bine, nu?

3.2 Costul unui apel de procedură

Asta este deja o treaba foarte simplă de calculat. Pur şi simplu folosim programul de mai sus, numit "prima măsurătoare", fără a pune nimic în corpul funcţiei `f`. Pe sistemul meu timpul a ieşit 33.9ns, adică aproximativ 9 cicli de procesor.

3.3 Durata unui apel de sistem simplu

Este interesant de măsurat care este durata executării unui apel de sistem (un serviciu elementar oferit de nucleu). Măsurarea timpului însăşi foloseşte funcţia `getitimer()` care la rîndul ei foloseşte un apel de sistem.

Trebuie să fim însă foarte atenţi să măsurăm apeluri de sistem *idempotente*, adică a căror execuţie repetată va da mereu aceleaşi rezultate. De exemplu, probabil că nu putem măsura durata citirii dintr-un fişier folosind metoda noastră, pentru că numai prima citire va accesa discul; a doua va lua datele din cache. Aceasta este o limitare importantă a testelor repetitive, ca cel pe care l-am scris.

De asemenea, pentru că nu este foarte clar cui îi atribuie nucleul timpul petrecut în timpul unui apel de sistem, vom măsura timpul **real**. Dacă apelul nu este blocant, avem şanse ca în timpul testului să nu fie executat nici un alt proces. Această măsurătoare e musai sa fie executată pe o maşină ne-încărcată, pentru că altfel rezultatele vor fi puternic distorsionate de celelalte procese.

Oricum, pentru un apel de sistem foarte simplu, cum e `getpid()` (pentru multe alte detalii privitoare la acesta vedeţi şi articolul meu din PC Report ianuarie 1998, "Anatomia unui apel de sistem") putem rula următorul cod:

```
/* masurarea unui apel de sistem */

/* nu uitati sa includeti get_etime(), etc... */

#include <unistd.h>

int main(void)
{
    init_etime();
    printf("Un apel de sistem dureaza %fs\n", ftime(getpid, 0.02, 1));
    /* o sa obtineti un warning la compilare din cauza ca
       getpid intoarce de fapt un intreg; nu-i problema. */
    return 0;
}
```

Pe aceeaşi maşină, rulînd Linux 2.0.30, am obţinut un timp de 2.00 microsecunde pentru un apel de sistem. Am avut deci dreptate: un apel de sistem e mult mai scurt decît rezoluţia ceasului.

3.4 Durata comutării de procese

În fine, încheiem această parte a articolului cu o măsurătoare a duratei comutării între două procese.

Vom scrie două procese care ştim că se blochează reciproc, în aşa fel încît execuţia se pasează de la unul la celălalt. Din nou, măsurăm timpul real.

Programul acesta cere cîteva cunoştinţe elementare despre Unix; unele dintre funcţiile componente le-am explicat mai demult, de exemplu în articolul "Shell-ul" din PC Report iunie 1997.

Ideea este simplă: deschid două ţevi şi fac două procese: unul citeşte din una şi scrie în cealaltă cîte un caracter şi viceversa. Cînd un proces încearcă să citească dintr-o ţeavă goală se blochează; în acest fel la fiecare citire/scriere se produce şi o comutare de procese. Măsurătoarea se face numai de către procesul părinte.

```
/* masurarea timpului de comutare intre procese */

#include <unistd.h>

int p[2], r[2];
char c;

void t(void)
{
    write(r[1], &c, 1);
    read(p[0], &c, 1);
}

double forktest(void)
{
    int copil;

    if (pipe(p) < 0 || pipe(r) < 0) exit(1);
    copil = fork();
    if (copil) {
        double rv = ftime(t, 0.01, 1);
        kill(copil, 9);
        return rv;
    }
    else {
        while (1) {
            read(r[0], &c, 1);
            write(p[1], &c, 1);
        }
        return 0;
    }
```

```
}

int main(void)
{
    init_etime();
    printf("Comutarea intre 2 procese dureaza %gus\n", 1e6*forktest());
    return 0;
}
```

Pe maşina mea rezultatul afişat a fost: 17.8528us. Asta include evident şi costul a patru apeluri de sistem (cîte un `read` şi un `write` în fiecare din procese) şi două comutări (dus-întors). Asta înseamnă că o comutare de fapt durează cam 17 - 4*2 / 2 = 4.5 microsecunde. Asta este o performanţă excelentă pentru un sistem de operare, fiind şi una dintre explicaţiile succesului Linux.

4 Încheiere temporară

Nu pot decît să vă invit să exersaţi măsurătorile pe tot felul de alte probleme. De exemplu, este mult mai greu să măsuraţi timpul de creare al unui proces, din cauza ne-idempotenţei. Măsuraţi apoi tot felul de alte lucruri, ca timpul de acces la disc, timpul de transfer în reţea, etc. Vedeţi ce iese. În partea a doua vom continua explorările utilizărilor măsurătorilor pe nişte cazuri mult mai complicate.

Partea II
Măsurători indirecte

În numărul anterior din PC Report (septembrie 1998) am văzut cum putem măsura intervale de timp folosind instrumentele puse la dispoziţie de către sistemul de operare; de asemenea am folosit aceste instrumente pentru a ne face o idee despre durata unora dintre activităţile fundamentale, cum ar fi tactul procesorului, durata unui apel de procedură, durata unui apel de sistem, etc. Codul procedurilor din acel articol este disponibil şi din pagina de web a autorului, pentru cei care nu au cumpărat numărul anterior al revistei, sau pentru cei care sunt prea leneşi ca să tasteze.

În acest articol vom continua să folosim măsurătorile de timp, dar de data aceasta în scopuri indirecte. Activităţi de acest gen sunt foarte des întîlnite în proiectarea sistemelor de calcul: măsurarea directă a unor parametri este adesea imposibilă, şi ca atare proiectanţii trebuie să măsoare fenomene diferite, iar apoi printr-un raţionament de la efecte spre cauze să deducă motivele cărora rezultatele li se datorează.

Cu cît sistemele în cauză sunt mai complicate, cu atît mai dificil de interpretat devin rezultatele. Dacă un efect poate fi produs de cauze multiple, din care unele nu sunt sub controlul direct al experimentatorului, atunci devine foarte greu de explicat cu exactitate în ce măsura rezultatul este o interferenţă a părţilor componente şi ce contribuţie are fiecare dintre ele.

Chiar şi acest articol, care măsoară un sistem relativ simplu şi aflat în întregime (aproape) la dispoziţia experimentatorului, va avea pe alocuri dificultăţi în a explica unele dintre fenomene. Pentru că informaţiile pe care le voi extrage din măsurători sunt în mare parte nepublicate de către fabricanţi, nici nu avem o metodă directă de a le verifica; este posibil ca erori de interpretare să se fi strecurat în raţionamentele mele. Cititorii care le vor decela sunt invitaţi să le comunice autorului.

5 Măsurarea parametrilor cache-urilor microprocesorului

5.1 O digresiune

O figură marcantă în galeria celebrităţilor proiectanţilor de sisteme este Mahadev Satyanarayanan[1]. Domnului Satyanarayanan îi este atribuit un citat care afirmă că în proiectarea sistemelor există doar două unelte fundamentale:

- Cache-urile;
- Un nivel suplimentar de indirectare.

Aserţiunea este cel puţin spectaculoasă şi este rezultatul distilării experienţei de zeci de ani de proiectare a unor sisteme complexe. Eu personal nu sunt capabil să argumentez pro sau contra, dar îmi sprijin inserţia acestui citat pe autoritatea incontestabilă a domnului Satyanarayanan.

Despre cache-uri am publicat un articol amplu în PC Report din martie 1997[2]; cu toate acestea, sunt departe de a fi spus totul despre ele (chiar în textul de faţă ne vom întîlni cu multe lucruri noi). Pentru că de fapt restul articolului va fi consacrat în întregime cache-ului asociat microprocesorului, o să spun doar cîteva cuvinte despre noţiunea de indirectare, din dorinţa de completitudine.

Indirectarea este oferită de către "referinţe" (pointeri): în loc de a vorbi despre o valoare vorbeşti despre *locul* unde se află acea valoare. Noţiunea de variabilă este un exemplu de indirectare: cînd vorbesc despre X nu spun ce valoare are X, dar am la dispoziţie o metodă pentru a afla această valoare atunci cînd am nevoie. Un nume dat unui obiect este o indirectare pentru acel obiect. Numele simbolice din Internet sunt indirectări pentru adrese reale ale calculatoarelor, un callback este o indirectare pentru o funcţie care trebuie chemată, un număr de inod este un nivel de indirectare pentru un fişier pe disc în Unix, etc.

5.2 Cache-uri: o recapitulare sumară

Să revenim acum la cache-uri. Dacă vreţi un studiu detaliat al arhitecturii cache-urilor de microprocesor atunci trebuie să citiţi capitolul 5 din cartea "Computer Architecture: A Quantitative Approach", ediţia a IIa, Morgan Kaufmann, 1995, de David Patterson, de la universitatea Berkeley şi John Hennessy, de la universitatea Stanford[3]. Această carte (şi versiunea ei oarecum simplificată "Computer Organization and Design: The Hardware/Software Interface", de aceeaşi autori, 1998) este baza majorităţii cursurilor de arhitectura calculatoarelor din Statele Unite. Capitolul 5 numai are 100 de pagini şi rezumă circa 1600 de articole de cercetare publicate numai între 1989 şi 1995.

Pentru concreteţe voi studia cache-ul ataşat procesorului maşinii aflate pe biroul meu, un Pentium II la 266Mhz, vechi de un an (poate între timp detaliile arhitecturale s-au schimbat).

[1] Cea mai cunoscută realizare a sa este sistemul de fişiere distribuit numit Andrew File System, iniţial dezvoltat la universitatea Carnegie Mellon, actualmente vîndut de compania Transarc, o filială IBM. Un derivat AFS numit Episode este sistemul de fişiere de bază din mediul de calcul distribuit (DCE: Distributed Computing Environment) al organizaţiei OSF (Open Software Foundation).

[2] Textul articolului este disponibil în pagina de web a autorului.

[3] Berkeley şi Stanford sunt clasate între primele 4 universităţi din Statele Unite în privinţa cercetării în domeniul calculatoarelor.

Ştim că programul şi datele pe care acesta le prelucrează se află în mod normal în memoria calculatorului (iar atunci cînd calculatorul este stins, pe disc). O memorie DRAM tipică pentru un calculator modern are un timp de acces de ordinul a 50 de nanosecunde; asta înseamnă că din clipa în care procesorul vrea să ia un cuvînt din memorie pînă la livrarea lui trec 50 de nanosecunde.

Procesorul meu însă (pe lîngă faptul că este o maşină superscalară, deci poate executa mai multe instrucţiuni deodată) poate executa o instrucţiune în fiecare ciclu de ceas; la 266Mhz asta înseamnă ceva mai puţin de 4 nanosecunde pentru o instrucţiune.

Discrepanţa între aceşti timpi este foarte mare. Soluţia de a folosi memorii mult mai rapide, SRAM, nu este practică pentru că SRAMul este de cîteva zeci de ori mai scump decît DRAMul, deci preţul calculatorului ar creşte prea mult.

Din fericire procesorul tinde să re-folosească fiecare dintre instrucţiunile executate şi dintre datele prelucrate de mai multe ori în intervale scurte de timp; această proprietate (observată empiric) a programelor se numeşte "localitate", şi este cea care permite ingenioasa soluţie a cache-urilor: construim o memorie intermediară numită "cache" (de la "ascuns"), cu timp mic de acces, dar de capacitate mică. Atunci cînd procesorul vrea să prelucreze date le aduce în cache. Prima oară o să aştepte, dar a doua oară timpul de acces va fi redus simţitor. Cînd cache-ul se umple procesorul elimină dintre datele care probabil nu vor mai fi folosite în curînd.

Asta e ideea de bază, şi este extrem de practică. Noi înşine, oamenii, tindem să repetăm materia înaintea unui examen, pentru a împrospăta cunoştinţele din "cache" (memoria de scurtă durată şi capacitate mică), folosind aceeaşi observaţie despre localitate (şi anume că putem uita totul de îndată ce examenul a trecut).

O încercare de acces care găseşte datele în cache se numeşte "lovitură" (hit); un rateu în engleză se cheamă "miss". Evident, un rateu este întotdeauna mai costisitor (ca timp) decît o nimereală.

Dacă datele cerute nu sunt în cache, atunci de obicei procesorul este blocat (*stalled*) pînă cînd se face transferul din memoria lentă.

Pentru că în cache se pot afla simultan doar o parte din octeţii din memorie, fiecare cuvînt din cache este însoţit de o "etichetă" (*tag*) care arată de unde vine acel cuvînt.

5.3 Dimensiunea cache-ului

În primul nostru experiment vom încerca să măsurăm cît de mare este cache-ul asociat procesorului.

Cum facem asta? Simplu: citim din memorie date aflate una lîngă cealaltă. Dacă datele încap toate în cache, durata unei citiri *repetate* va fi în medie mică. Dacă datele nu încap însă, de fiecare cînd citim o dată nouă una veche va trebui să fie scoasă şi cea nouă adusă din memorie. În felul acesta, de îndată ce citim mai multe date decît încap în cache, trebuie să observăm o încetinire a vitezei de execuţie.

Am scris deci următorul program:

```
/* Masurarea ratei de transfer din memorie (cache) */

#include "ftime.h"       /* procedurile pentru masurat */
#define MAXTRANSFER (4 * 1024 * 1024) /* 4 Mb */

typedef int unitate;     /* unitatea de transferat: 4 octeti */

union {
```

```
   char array[MAXTRANSFER];
   unitate aliniament;
} de_transferat;

int TRANSFER;            /* citi octeti transferam */
register dummy;

#define UNROLL 64

#define ACCES(adresa)  a += *(adresa);
#define ACCES4(adresa) \
  ACCES(adresa+0) ACCES(adresa+1) \
  ACCES(adresa+2) ACCES(adresa+3)
#define ACCES16(adresa) \
  ACCES4(adresa+0) ACCES4(adresa+4) \
  ACCES4(adresa+8) ACCES4(adresa+12)
#define ACCES64(adresa) \
  ACCES16(adresa+0) ACCES16(adresa+16) \
  ACCES16(adresa+32) ACCES16(adresa+48)

void citire(void)
{
    register int i;
    register unitate a;
    register int *b = (int*)de_transferat.array;

    for (i = TRANSFER / sizeof(unitate); i >= 0; i -= UNROLL) {
        ACCES64(b)
        b += UNROLL;
    }
    dummy = a;
}

void citire_ajustare(void)
{
    register int i;
    register unitate a;
    register unitate *b = (unitate*)de_transferat.array;

    for (i = TRANSFER / sizeof(unitate); i >= 0; i -= UNROLL) {
        b += UNROLL;
    }
    dummy = a;
}
```

```
int main(void)
{
     int i;
     double Mhz, durata_ciclu;

     init_etime();
     Mhz = mhz(0.02);
     durata_ciclu = 1.0 / Mhz / 1e6;
     printf("Frecventa ceasului este %gMhz\n", Mhz);

     for (i=0; i < MAXTRANSFER; i++)
        de_transferat.array[i] = 0;

     for (TRANSFER=512; TRANSFER < MAXTRANSFER; TRANSFER <<= 1) {
          double t = ftime(citire, 0.01, 0) - ftime(citire_ajustare, 0.01, 0);
          printf("Cantitate %d, rata %2.2fMb/s, %2.2fcicli/cuvint\n",
                 TRANSFER,
                 (double)TRANSFER / t / (1024*1024),
                 t / durata_ciclu / TRANSFER);
     }
     return 0;
}
```

Ce se întîmplă trebuie să fie destul de clar pentru cei care au citit prima parte a articolului. Cu toate acestea trebuie să fim atenţi la anumite subtilităţi.

Folosim funcţiile `mhz()`, `init_etime()`, `ftime()`, dezvoltate data trecută. Facem măsurătoarea citind cuvinte adiacente dintr-un vector foarte mare. Pentru că citim cîte un întreg odată, care are probabil 4 octeţi, trebuie să luăm nişte măsuri de precauţie. Multe calculatoare nu permit citirea datelor ne-aliniate: orice cuvînt de 4 octeţi trebuie să fie accesat la o adresă multiplu de 4. Acesta este şi scopul acelei `union` din programul de mai sus: vectorul de caractere numit `array` va fi aliniat în acelaşi fel cu întregul numit `aliniament`, pe care compilatorul de C garantează să-l pună la o adresă multiplu de 4. Pentru Pentium aliniamentul nu contează de fapt, dar codul de mai sus funcţionează şi pentru alte procesoare.

O altă tehnică foarte interesantă şi demnă de reţinut este *desfăşurarea buclei* (loop unrolling). În loc să repetăm de un număr de ori o citire din memorie repetăm de 64 de ori mai puţin 64 de citiri din memorie.

De ce facem asta? Pentru că în felul acesta facem de 64 de ori mai puţine incrementări ale contorului buclei, şi facem de 64 de ori mai puţine teste de terminare a buclei. Din cauza asta măsurătoarea este mai precisă. Tehnica aceasta este folosită cu succes şi de către compilatoarele care optimizează. Pe lîngă faptul că numărul de instrucţiuni utile executate creşte, o buclă cu un corp mai lung oferă mai multe oportunităţi pentru "scheduling", dînd posibilitatea procesorului să execute mai multe instrucţiuni simultane mai des.

Variabila `dummy` este folosită pentru a împiedica compilatorul să optimizeze prea tare codul (întreaga buclă ar deveni inutilă altfel).

Esenţială este execuţia buclei `for` care iniţializează toată matricea `de_transferat.array` cu 0: dacă nu facem treaba asta sistemul de operare nici măcar nu va aloca pentru noi memoria respectivă şi orice citire din acea memorie va da automat un rezultat 0, fără ca memoria să fie citită de fapt. (Mi-au ieşit peri albi pînă am înţeles de ce măsurătorile care nu iniţializau matricea dădeau rezultate aberante.)

Iată şi rezultatele măsurătorii:

Octeţi	MB/s	Cicli/cuvînt
512	961.50	1.06
1024	972.92	1.05
2048	934.09	1.09
4096	958.68	1.06
8192	971.47	1.05
16384	933.42	1.09
32768	474.40	2.15
65536	474.23	2.15
131072	474.15	2.15
262144	426.70	2.39
524288	316.05	3.23
1048576	228.57	4.47
2097152	228.57	4.47
4194304	224.56	4.55

Observăm imediat un rezultat foarte interesant: viteza de transfer are 3 paliere, la 1Gb/sec, jumătate de Gb/s şi un sfert de Gb/sec. De ce cele 3 valori diferite?

Din cauză că viteza procesoarelor creşte — datorită avansului tehnologiei — mult mai repede decît viteza de răspuns a memoriei, calculatoarele moderne au de fapt 2 nivele de cache, unul mai mic şi mai rapid, numit L1, de obicei din SRAM, şi unul mai mare şi ceva mai lent (L2). La procesorul Pentium II cache-ul L1 este pe acelaşi integrat cu procesorul, iar cache-ul L2 este în acelaşi soclu. Cache-ul L2 este de de obicei DRAM lent, dar are un bus extrem de lat şi scurt, deci poate susţine o rată de transfer relativ mare comparativ cu memoria de bază.

Măsurătorile de mai sus indică un cache L1 de 16Kb şi un cache L2 de 512Kb.

Rata de transfer nu descreşte subit la paliere ci se degradează pe măsură ce ocupăm mai mult din cache pentru că programul însuşi şi sistemul de operare folosesc părţi din cache, care concurează pentru spaţiu în cache cu matricea `array`. Efectul este însă destul de pronunţat.

5.4 *Write back* sau *write through*?

Atunci cînd procesorul scrie date, ce se întîmplă cu cache-ul? La întrebarea asta atît de simplă există o multitudine de răspunsuri posibile.

Prima alegere pe care o are de făcut un designer este între a păstra valoarea scrisă numai în cache (asta se numeşte *write-back*) sau de a propaga simultan valoarea scrisă în nivelul următor al ierarhiei de memorie (*write-through*).

Avantajul metodei din urmă este că în fiecare clipă ştim că ceea ce se găseşte în cache la o anumită adresă este o copie fidelă a ceea ce se găseşte în memorie (copiile sunt *coerente*). Asta simplifică oarecum lucrurile cînd avem de-a face cu mai multe microprocesoare care accesează aceeaşi memorie; de asemenea avem certitudinea că în cazul cînd anumite cuvinte sunt scoase din cache pentru a face loc altora valoarea lor nu mai trebuie salvată în nivelul inferior.

Pe de altă parte write-through nu poate scrie mai repede decît poate absorbi nivelul următor din ierarhia de memorii (e clar: pe o şosea nu pot intra mai multe maşini decît ies decît pentru foarte puţin timp).

Am rescris testul anterior astfel:

```
#define ACCES(a) (*(a) = 0);

....

void atinge(void) {
    int i, a;
    for (i=0; i < TRANSFER/sizeof(unitate); i++)
        a += de_transferat.array[i];
    dummy = 0;
}

int main(void)
{
     for (...) { /* ca mai sus */
          atinge();
          .... /* ca mai sus */
     }
}
```

Am schimbat macro-ul `ACCES` pentru a scrie în loc de a citi. În plus am scris o funcţie mică, numită `atinge()`, care citeşte fiecare cuvînt din array înainte de a face măsurătoarea. Vom vedea în secţiunea următoare de ce vrem să atingem fiecare cuvînt înainte de a măsura.

Iată şi rezultatele:

Octeţi	MB/s	Cicli/cuvînt
512	849.79	1.20
1024	873.34	1.17
2048	856.00	1.19
4096	861.86	1.18
8192	857.57	1.19
16384	855.45	1.19
32768	237.11	4.31
65536	237.07	4.31
131072	237.05	4.31
262144	213.34	4.79
524288	158.02	6.46
1048576	88.88	11.50
2097152	76.19	13.41
4194304	75.29	13.57

Din nou avem trei paliere dar, exceptîndu-l pe primul, performanţa este substanţial mai mică. Cu toate acestea putem afirma că cele două nivele sunt write-back. De ce? Dacă L1, de pildă, era write-through, am fi avut exact aceeaşi rată de transfer pentru tot domeniul 512 octeţi — 512 Kb.

De ce e rata de scriere mai mică decît cea de citire? Sunt mai multe posibile motive la mijloc:

- Pentru că datele sunt modificate, atunci cînd o valoare este scoasă din cache pentru a aduce o alta ea trebuie şi salvată în nivelul următor din ierarhia de memorie. Cu alte cuvinte, cînd scriu octetul cu numărul 32768, octetul cu numărul 0 trebuie salvat din cache-ul L1 în L2. Scrierea va implica deci două acţiuni: o scriere din L1 în L2 în paralel cu o scriere din microprocesor în L1;
- În general citirea se poate implementa mai rapid. De ce? Atunci cînd vreau să citesc de la adresa X din memorie, trebuie făcute următoarele operaţii:

  1. Din adresa X se extrage posibila adresă a octetului din cache;
  2. Adresa octetului care se află în cache în acel loc (tag-ul) este comparată cu X;
  3. Dacă în cache se află într-adevăr octetul de la adresa X octetul este transferat spre microprocesor. Altfel este iniţiată o citire din nivelul următor al ierarhiei.

  Diferenţa între citire şi scriere este că la citire paşii 2 şi 3 de mai sus pot fi iniţiaţi simultan. Pentru scriere însă paşii 2 şi 3 trebuie efectuaţi strict unul după altul.
- Statistic s-a constatat că în general citirile din memorie sunt cam de 4 ori mai frecvente decît scrierile (de exemplu codul programului însuşi este de obicei imutabil; este doar citit de microprocesor şi niciodată modificat). Din cauza asta proiectanţii optimizează procesorul pentru a face citirile mai rapid.

5.5 *Write allocate on write miss?*

O altă alegere pe care o pot face proiectanţii este dacă atunci cînd se scrie într-un cuvînt care *nu* se află în cache cuvîntul trebuie adus în cache sau scrierea trebuie făcută direct în nivelul următor. Dacă cuvîntul este totuşi adus, cache-ul se cheamă "write-allocate on write miss", din motive evidente. Altfel cache-ul se numeşte "write-around". În general, un cache write-through este şi write-around şi invers, un cache write-back este write-allocate.

Pentru a verifica ipoteza este suficient să scoatem din programul anterior apelul funcţiei `atinge()`; măsurătorile ne confirmă că Pentium are un cache write-allocate pentru că performanţa este foarte asemănătoare.

5.6 I-cache şi D-cache

O întrebare la care este mult mai greu de răspuns este dacă există unul sau două cache-uri separate pentru program şi date. Modul de acces la instrucţiuni este foarte diferit de modul de acces la date, aşa că are sens ca două cache-uri diferite să se ocupe de cele două zone diferite. În plus, în general, datele şi codul sunt puse în memorie în zone complet diferite.

Pentru a putea răspunde la această întrebare ar trebui să scriem un program care se auto-modifică şi care poate controla propria sa aşezare în memorie; treaba este posibilă dar relativ încîlcită, aşa că o lăsăm ca un exerciţiu pentru cititorul cu înclinaţii de hacker.

O căutare pe serverul de web de la Intel ne va indica faptul că există două cache-uri L1, ambele de 16K, unul pentru cod şi unul pentru date. Numele tradiţionale sunt I-cache (instruction-cache) şi D-cache (data-cache).

L2 este însă un cache *unificat*: 512K pentru ambele foloase.

5.7 Asociativitatea cache-ului

Gradul de asociativitate al unui cache arată cît de uşor poate fi localizat un cuvînt în cache. Cele mai simple cache-uri (numite *direct-mapped*) caută un cuvînt într-un mod foarte simplu: ignoră primii biţi din adresă şi indexează în cache cu adresa obţinută. Dacă mărimea cache-ului este M, atunci toate cuvintele de memorie aflate la o adresă de forma $kM + o$, cu k întreg, se vor afla în acelaşi loc în cache, la adresa o (desigur, nu toate simultan; cel mult unul dintre ele la un moment dat). Cache-urile directe sunt foarte simplu de construit, sunt mici (ca circuite hardware) şi foarte rapide.

Varianta alternativă este ca un cuvînt din memorie să se poată afla într-unul din mai multe locuri din cache. Dacă avem un cache de mărime M *asociativ pe n căi*[4] atunci există n locuri în care un cuvînt de la o adresă se poate afla. Şi anume cuvîntul de la adresa X se poate afla la oricare din adresele multiple de X mod M/n.

De exemplu pentru $M = 16$ (ridicol de mic), $n = 4$, cuvîntul de la adresa 1 din memorie se poate afla în cache în oricare din adresele 1, 5, 9 şi 13.

Un cache asociativ este mult mai flexibil decît unul direct. De exemplu, în cache-ul asociativ de mai sus, se pot afla simultan cuvintele din memorie de la adresele 1 şi 17, pe cînd într-un cache direct cu aceeaşi capacitate ($M = 16$) nu.

Cache-ul extrem, pentru care $n = M$, se numeşte *complet asociativ* (fully associative). Cu cît gradul de asociativitate (n) creşte, cu atît complexitatea circuitului creşte. Cu tehnologia curentă este complet ne-practic de implementat în hardware un cache complet asociativ mai mare de 256 de cuvinte.

Din această raţiune la Pentium II cache-ul L1 este asociativ pe 4 căi, iar L2 este direct. Cum am aflat asta? Am scris un mic test care încearcă să-l ghicească pe n. Ne-am bazat exact pe observaţia de mai sus.

Pentru cache-ul ipotetic de mai sus, cu $M = 16$, am putea face următoarele teste:

- Cît de repede putem citi octeţii 1 şi 17?
- Cît de repede putem citi octeţii 1, 17, 17+16=33, 17+32=49?
- etc.

Dacă primul test indică o rată de transfer la fel de mare ca transferul unor date consecutive, atunci ştim că $n >= 2$, pentru că 1 şi 17 încap simultan înauntru. Dacă viteza este substanţial mai mică ştim însă că $n = 2$.

Dacă $n >= 2$, al doilea test ne va spune dacă $n >= 4$ în exact acelaşi fel. Aţi prins ideea?

```
/* masurarea asociativitatii cache-ului L1 */

#define L1 (16 * 1024)   /* tocmai am masurat astea */
#define L2 (512 * 1024)

int ASOC;
int BLOC;
```

[4] Întotdeauna n divide pe M.

```
void asociativitate(void)
{
    register int i;
    register unitate a;
    register unitate *b = (unitate*)de_transferat.array;
    for (i=ASOC; i > 0; i--) {
        ACCES64(b);
        b += BLOC;
    }
    dummy = a;
}


void asociativitate_ajustare(void)
{
    register int i;
    register unitate a;
    register unitate *b = (unitate*)de_transferat.array;
    for (i=ASOC; i > 0; i--) {
        b += BLOC;
    }
    dummy = a;
}

int main(void)
{
     int i;
     double Mhz, durata_ciclu;

     init_etime();
     Mhz = mhz(0.02);
     durata_ciclu = 1.0 / Mhz / 1e6;
     printf("Frecventa ceasului este %gMhz\n", Mhz);

     BLOC = L1;
     for (i=1; i <= 16; i <<= 1) {
         double t;

         ASOC = i;
         t = ftime(asociativitate, 0.01, 0)
             - ftime(asociativitate_ajustare, 0.01, 0);
         printf("Distanta %d, %2.2cicli/cuvint\n",
                ASOC, t / durata_ciclu / ASOC / UNROLL);
     }
     return 0;
```

```
}
```

Testele măsoară cuvinte aflate la `BLOC` octeţi unul de altul în RAM, cuvinte despre care ştim că se vor bate pe aceeaşi linie din cache. Făcînd pe `BLOC = L1` măsurăm asociativitatea lui L1 (şi similar pentru L2). Încercăm felurite asociativităţi, toate puteri ale lui 2. Ne oprim la 16 în programul de mai sus, dar dacă asta nu e suficient putem continua testele.

Rezultatele numerice pentru L1 sunt următoarele:

Distanţa	cicli/cuvînt
1	1.12
2	1.10
4	1.47
8	2.14
16	2.12

Asta ne confirmă într-adevăr că avînd un asociativitatea cache-ului L1 este 4.

Cum putem măsura asociativitatea cache-ului L2 fără a fi încurcaţi de faptul că L1 este asociativ el însuşi? Trebuie să controlăm accesele în aşa fel încît rateurile să se producă şi în L1 şi în L2, deşi L1 va încerca să păstreze cît mai multe date.

Din moment ce ştim deja asociativitatea lui L1 am scris un program care încarcă toate cele 4 linii unde se poate afla o adresă în L1, după care referă o altă linie care se bate cu prima pentru o linie în L2. Rezultatul arată cam aşa:

```
/* masurarea asociativitatii cache-ului L2 */

void asociativitate_L2(void)
{
    register int i;
    register unitate a;
    register unitate *b = (unitate*)de_transferat.array;
    for (i=ASOC; i > 0; i--) {
        ACCES16(b);
        ACCES16(b+L1);
        ACCES16(b+L1+L1);
        ACCES16(b+L1+L1+L1);
        b+=L2;
    }
    dummy = a;
}


void asociativitate_L2_ajustare(void)
{
    register int i;
    register unitate a;
    register unitate *b = (unitate*)de_transferat.array;
    for (i=ASOC; i > 0; i--) {
```

```
        b += L2;
    }
    dummy = a;
}

int main() { /* etc. */ }
```

A trebuit să măresc MAXTRANSFER la 17Mb ca să nu ies din marginile array-ului (cînd `ASOC = 8`, 8 cuvinte la 512Kb distanţa înseamnă 16Mb). Măsurătorile au arătat că pentru L2 gradul de asociativitate este 1, deci cu alte cuvinte L2 este direct-mapped.

5.8 Cache-ul victimă

Cache-urile asociative costă foarte mult; cu cît creşte gradul de asociativitate cu atît complexitatea hardware este mai mare, dar cu atît performanţele pe o gamă largă de programe vor fi mai bune. Proiectanţii de cache-uri au găsit o soluţie ingenioasă care rezolvă cazuri patologice de acces la memorie cu un cost minim. Ideea este de a adăuga unui cache asociativ pe n căi un cache extrem de mic (pînă în 10 cuvinte, poate chiar unul singur), dar complet asociativ. Atunci cînd un cuvînt este evacuat din cache-ul primar el va fi pus în acest "depozit" în speranţa că va fi folosit din nou în curînd. Acest minuscul cache se numeşte, din motive, sper evidente, cache-victimă. Simulările arată că pentru o gamă largă de programe cache-ul victimă este extrem de eficace în a reduce rateurile care nu pot fi evitate de gradul de asociativitate. Faţă de costul redus al hardware-ului beneficiile sale sunt remarcabile.

Pentium II este un procesor foarte modern, deci este destul de plauzibil să fie echipat cu un cache-victimă pentru L1. Cum putem să determinăm dacă e adevărat şi să măsurăm capacitatea sa?

Simplu: ne bazăm chiar pe observaţia precedentă, cum că acest cache este folosit pentru a ţine informaţiile care scapă din memoria asociativă. Atunci vom concepe un test care face din ce în ce mai multe cuvinte să se bată pentru cache-ul victimă, pînă cînd vom observa o degradare a performanţei.

De data asta nu avem voie să atingem mai multe cuvinte consecutive; trebuie să citim exact 5, 6, 7, etc. cuvinte care se bat pe aceeaşi linie, iar atunci cînd observăm o degradare a performanţei putem trage concluzia că am depăşit capacitatea victimei.

```
Testul pentru cache-ul victima:

          ----------
         |__________|  ^
         |__________|  | grad de asociativitate
         |__________|  v
         |__________|  ^ bataie pentru victima
         |__________|  v
```

Primul test pe care l-am făcut, cu 5 cuvinte, a arătat deja o descreştere substanţială a ratei de transfer, deci am tras concluzia ca Pentium de fapt nu are cache victimă.

```
/* Masurarea cache-ului victima */
void victima(void)
```

```
{
    register unitate a;
    register unitate *b = (unitate*)de_transferat.array;
    ACCES(b);
    ACCES(b+L1);
    ACCES(b+L1+L1);
    ACCES(b+L1+L1+L1);
    ACCES(b+L1+L1+L1+L1);
    dummy = a;
}

void victima_ajustare(void)
{
    register unitate a;
    register unitate *b = (unitate*)de_transferat.array;
    dummy = a;
}

int main(void)
{
    /* etc. */
    t = ftime(victima, 0.01, 0) - ftime(victima_ajustare, 0.01, 0);
    printf("%2.2fcicli/cuvint\n", t / durata_ciclu / 5);
}
```

5.9 Mărimea liniei din cache

Din motive de economie şi localitate octeţii din cache nu sunt independenţi, ci sunt grupaţi în *blocuri* sau *linii*. Astfel, dacă unul dintre octeţii dintr-un bloc este referit de către procesor, toţi ceilalţi octeţi din linie sunt şi ei aduşi în cache. Economia care se obţine în acest fel este de etichete de adresa (tags): toţi octeţii dintr-o linie au aceeaşi etichetă.

(La scriere lucrurile pot varia: unele procesoare pot suporta modificarea şi salvarea numai unora dintre octeţii dintr-o linie).

Cum putem măsura mărimea liniei? În ceea ce priveşte L2 nu am reuşit să concep un test suficient de discriminator, mai ales pentru că probabil linia din L2 este mai mică decît cea din L1, deci nu pot să evit interferenţele! Cititorul care concepe un test este rugat să mi-l aducă la cunoştinţă.

Mărimea liniei din L1 poate fi determinată după cum arată următoarea figură:

```
Determinarea marimii liniei de cache L1:
                    _ ...
                   |_| ^
                   |_| | ASOC (grad de asociativitate
                   |_| |           incluzind victima)
       _............|_|.v.
      |_|.............
      <---LINIE------->
       ^
       | Se bate octetul asta cu cei ``verticali'' pentru o linie?
       Daca da, atunci marimea liniei este cel putin LINIE, altfel
       marimea este mai mica.
```

Testul acesta trebuie făcut cu grijă, pentru că dacă o linie este mai mare decît bus-ul pe care se transportă datele nu toţi octeţii dintr-o linie vor veni deodată. De exemplu dacă linia din L1 are 16 octeţi (128 de biţi) şi dacă transferul între L2 şi L1 se face pe cîte 64 de biţi, o linie trebuie adusă de fapt în doi paşi.

Măsurătoarea pe care am pus-o la cale se desfăşoară astfel: pun un cuvînt în cache, si după aia accesez alte 4 cuvinte. Dacă linia octetului cu pricina concurează cu liniile celorlalţi 4 am o idee despre lungimea liniei.

Iată şi programul:

```
int LINIE;

void linie(void)
{
    register unitate *b = (unitate*)de_transferat.array;
    register unitate a;
    ACCES(b);
    ACCES(b+LINIE);
    ACCES(b+LINIE+L1);
    ACCES(b+LINIE+L1+L1);
    ACCES(b+LINIE+L1+L1+L1);
    dummy = a;
}


void linie_ajustare(void)
{
    register unitate a = 0;
    dummy = a;
}

int main(void)
{
     /* init etc. */
```

```
    for (LINIE=20; LINIE >= 1; LINIE-=1) {
        double t;

        t = ftime(linie, 0.01, 0) - ftime(linie_ajustare, 0.01, 0);
        printf("Linie %d, %2.2fciclii/cuvint\n",
               LINIE, t / durata_ciclu / 5);
    }
}
```

Iată rezultatele:

Linie	cicli/cuvînt	Linie	cicli/cuvînt
20	2.29	10	2.32
19	2.26	9	2.32
18	2.29	8	3.44
17	2.32	7	4.44
16	2.57	6	3.27
15	2.57	5	3.22
14	2.29	4	3.36
13	2.32	3	3.39
12	2.29	2	3.36
11	2.35		

Valorile obţinute sugerează dimensiunea liniei de 8* `sizeof(unitate)` = 32 de octeţi = 128 de biţi. Valorile sunt relativ mari pentru că execut numai 5 instrucţiuni de citire în întreaga buclă, deci nu amortizez suficient costul buclei (ca mai înainte prin "unrolling").

5.10 Cache-uri blocante

Închei acest articol cu o provocare pentru cititor. Măsurători făcute de mine indică faptul că cache-ul Pentium nu este blocant. Va invit să verificaţi această aserţiune.

Procesoarele moderne pot lansa în execuţie mai multe instrucţiuni, care adesea operează asupra unor date diferite. În acest caz ar fi detrimental performanţei ca în momentul în care o instrucţiune dă un rateu în cache toate celelalte să fie forţate să aştepte. Un cache ne-blocant permite execuţia altor instrucţiuni în timp ce serveşte din memoria lentă rateul uneia. În felul acesta se poate întîmpla ca datele cerute de două instrucţiuni să sosească în ordine inversă cererilor!

Cache-urile ne-blocante sunt nişte dispozitive hardware extrem de sofisticate. Dar atunci cînd ai 15 milioane de tranzistori la dispoziţie (cît conţine Pentium II) nu te dai în lături de la nimic pentru o creştere a performanţei în execuţie.

6 Concluzii

Calculatoarele sunt nişte sisteme din ce în ce mai complicate; miniaturizarea şi tehnologiile înalte au permis realizarea fizică a unor sisteme extrem de sofisticate. Balanţa dintre părţile componente este în continuă schimbare: dacă în urmă cu 20 de ani viteza de acces a memoriilor şi viteza de calcul a procesoarelor erau relativ egale, în ziua de azi procesoarele evoluează cu o rată de creştere a performanţei de 55% pe an, faţă de memorii care cresc cu numai 7% în aceeaşi perioadă. Designerii sunt deci forţaţi să inventeze tot felul de noi dispozitive care să crească performanţa întregului sistem (care tinde să fie la fel de lent ca cea mai lentă componentă).

Din cauza asta măsurarea performanţei unui sistem este crucială pentru a înţelege ce îmbunătăţiri trebuie aduse. Adesea măsurătorile se pot face numai indirect, şi sunt influenţate de o multitudine de parametri independenţi. Proiectarea unor măsurători interesante şi interpretarea rezultatelor devin adevărate meşteşuguri.

Una peste alta, arhitecţii calculatoarelor nu vor rămîne prea curînd şomeri.

A-propos, ce meserie v-aţi ales?

A Derivarea duratei de execuţie

Pentru început vom calcula valoarea minimă şi maximă a timpului real de execuţie în funcţie de valoarea timpului măsurat şi de precizia ceasului.

Să facem următoarele notaţii:

- Δ: rezoluţia ceasului sistemului de operare;
- T_s: timpul de pornire al măsurătorii, cum e raportat de `get_etime()`;
- T_f: timpul final al măsurătorii, raportat de `get_etime()`;
- T: timpul măsurat de execuţie;
- t_s: timpul real de pornire ("timp absolut": Newtonian);
- t_f: timpul real final;
- t: timpul real de execuţie;

Ştim că următoarele relaţii sunt adevărate:

$$\begin{aligned} T_s \leq \quad & t_s \quad < T_s + \Delta \\ T_f \leq \quad & t_f \quad < T_f + \Delta \end{aligned}$$

Atunci:

$$\begin{aligned} t = t_f - t_s \quad & < \quad T_f + \Delta - T_s = T + \Delta \\ t = t_f - t_s \quad & > \quad T_f - (T_s + \Delta) = T - \Delta \end{aligned}$$

Deci avem:

$$T - \Delta < t < T + \Delta \tag{1}$$

Acum vom demonstra următoarea relaţie între T, Δ şi eroarea maximă relativă dorită ϵ: $T \geq \frac{1+\epsilon}{\epsilon}\Delta$.

Fie k un întreg pozitiv astfel încît $T = k\Delta$; putem presupune fără a pierde din generalitate că avem $k \neq 1$. Din 1 avem

$$T - \Delta < t < T + \Delta \Leftrightarrow -\Delta < t - T < \Delta.$$

Împărţind prin $T = k\Delta$ avem:

$$-\frac{1}{k} < \frac{t-T}{T} < \frac{1}{k} \tag{2}$$

Termenul din mijloc este chiar eroarea relativă pe care vrem s-o mărginim. Acum exprimăm eroarea relativă ca funcţie de raportul $r = \frac{t-T}{T}$:

$$\frac{t-T}{t} = \frac{(t-T)/T}{t/T} = \frac{r}{1-r} \tag{3}$$

Din ecuaţiile 2 and 3 obţinem:

$$\begin{aligned} \frac{t-T}{t} &< \frac{1/k}{1-1/k} = \frac{1}{k-1} \\ \frac{t-T}{t} &> \frac{-1/k}{1-1/k} = -\frac{1}{k-1}. \end{aligned}$$

sau

$$\left|\frac{t-T}{t}\right| < \frac{1}{k-1}.$$

Pentru că vrem ca eroarea relativă să fie mărginită de ϵ, trebuie să avem:

$$\frac{1}{k-1} \leq \epsilon \Leftrightarrow k \geq \frac{1+\epsilon}{\epsilon} \Leftrightarrow T \geq \frac{1+\epsilon}{\epsilon}\Delta.$$

QED.