Program de testare a performanțelor de transfer a memoriei cache

Student:Pop Adrian

Grupa:30233

An universitar:2021-2022

Cuprins:

1.Introducere:

1.1 Context și Obiective:

Scopul acestui proiect este de-a elabora un program care va prezenta informații legate de PC-ul folosit și care va putea testa și prezenta performanțele memoriei cache a acestuia. Rezultatele vor putea fi prezentate prin grafice sau prin mesaje . Există diferite tipuri de teste ce pot fi alese de către utilizator: pentru citire, pentru scriere și pentru testarea timpului mediu de acces.

Pe lângă faptul că programul poate fi folosit independent doar pentru testarea performantei memoriei cache, acesta poate fi folosit și în cadrul unui program de testare pentru procesor sau chiar pentru toate componentele PC-ului.

1.2 Specificații:

Interfața grafică prin care utilizatorul poate rula testele și vizualiza rezultatele este concepută în Java prin intermediul IDE-ului Intellij IDEA, iar partea din program ce rulează testele și produce fișierele cu rezultate este concepută în C/C++ folosind IDE-ul Microsoft Visual Studio.

1.3 Obiective:

Se vor evidenția vitezele de transfer pentru date aflate în memoria cache, în memoria internă și în memoria virtuală. Pentru testele de citire și scriere se vor trasa grafice in care se indica dependenta vitezei de transfer in raport diferiti parametri. Programul va putea rula o multitudine de teste de citire/scriere în memorie, va calcula și va face o medie între teste pe aceeași dimensiune de date și informațiile recoltate vor fi introduse într-un grafic ce va reprezenta timpul de lucru în funcție de dimensiune.

Pentru testul de hit și miss se vor realiza mai multe operații de scriere și se vor verifica timpii de finalizare pentru fiecare operație. Într-un final se va calcula timpul minim de acces al memoriei.

2.Studiul bibliografic:

Memoria cache reprezintă o componentă hardware sau software care este folosită pentru a stoca date pentru a le putea accesa mai rapid. Când sunt accesate date în urma unei operații, acestea se stochează în memoria cache pentru a economisii timp ce ar fi fost folosit pentru a căuta mai adânc într-o memorie mai lentă. Astfel cu cât se accesează mai des date din memoria cache, cu atât crește și performanța sistemului.

Pentru a facilita aspectul de viteză, o memorie cache ar trebuii să aibă dimensiuni mici pentru a nu irosii timp căutând date care nici nu ar fi stocate. De asemenea, o memorie cache poate să implementeze diferite strategii de stocare a datelor astfel încât datele care sunt accesate mai recent sunt stocate aproape una de cealaltă. O altă strategie reprezintă stocarea de date ce sunt folosite alături de alte date.

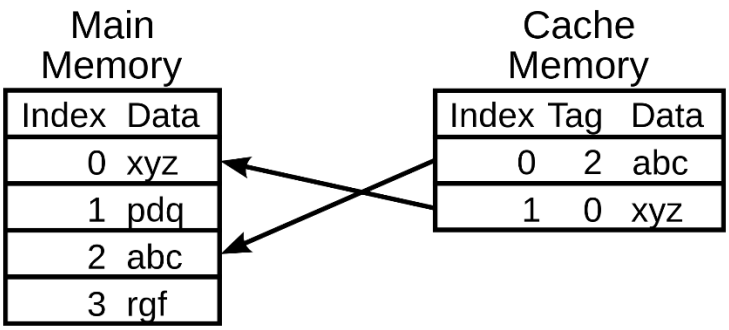


Fig 1. Structură obișnuită a unei memorii cache

Un exemplu de memorie cache se regăsește în cadrul procesorului. Memoria cache a procesorului reprezintă un cache de tip hardware care este situat între procesorul unui PC și memoria principală. Aceasta este situată aproape de procesor și este mai rapidă și mai mică decât memoria principală. Astfel aceasta este folosită pentru a stoca date accesate recent din memoria principală pentru ca procesorul să nu irosească timp căutând date într-o memorie mai lentă. Majoritatea cache-urilor pentru procesor prezintă o structură ierarhică de mai multe nivele, fiecare cu dimensiune mai mare ca cea precedentă: Nivel 1,2,3, uneori și 4.

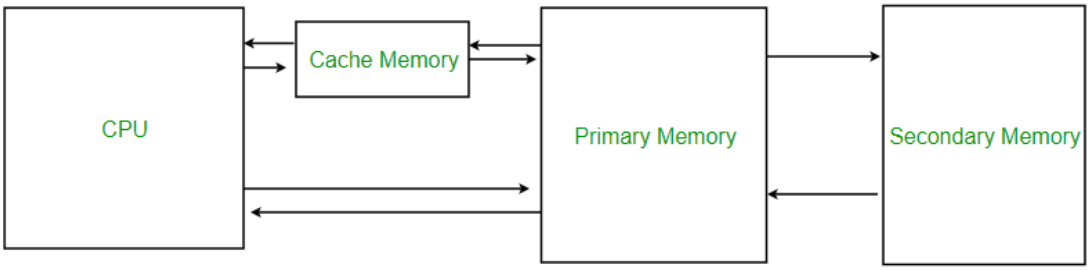


Fig 2. Procesorul și memoriile asociate acestuia

Atunci când se dorește să se facă loc pentru alte date, memoria cache poate să șteargă una din intrările existente. În vederea acestui lucru s-au creat mai multe euristici de înlocuire a datelor din cache. Fiecare are avantajele și dezavantajele sale, neexistând o metodă perfectă de înlocuire. Printre euristicile de înlocuire se regăsesc:

- ”**First-in-First-Out**” (=”primul venit primul ieșit”) unde se va scoate din cache prima informație introdusă, indiferent dacă aceasta este accesată des sau nu.
- ”**Least Recently Used**” (=”Cea mai puțin recent folosit”) unde se va elimina informația care n-a mai fost verificată de cel mai mult timp. Aceasta este una dintre cele mai populare euristici
- ”**Least Frequently Used**” (=”Cea mai puțin frecvent folosită”) unde, similar cu LRU, se va elimina infirmația care a fost accesată cel mai puțin.
- **Algoritmul lui Bélády** este considerat cel mai eficient algoritm de înlocuire a datelor. Acesta constă în ștergerea informației care este accesată cel mai puțin în viitor. Deoarece este imposibil de a vedea în viitor care date sunt accesate, acest algoritm nu are o

implementație practică, dar rezultatul obținut în urma experimentării poate fi comparat cu rezultatele obținute în urma aplicării unui alt algoritm cache.

O metodă folosită pentru a îmbunătății performanța este de-a verifica și ignora regiunile din cadrul memoriei care sunt accesate cel mai puțin. Pentru a verifica performanța unei memorii cache, se verifică numărul de cache hits și cache misses. Un cache hit are loc atunci când datele căutate se găsesc în cadrul memoriei cache. Un cache miss are loc atunci când nu s-a găsit informația căutată și se caută în următoarea memorie. Cea mai importantă metrică de verificare a performanței o reprezintă Timpul Minim de Accesare a Memoriei, care este calculată folosind formula:

$$AMAT = H + MR \cdot AMP$$

unde H reprezintă timpul mediu pentru un cache hit, MR reprezintă rata de cache miss a memoriei și AMP reprezintă penalizarea medie pentru un cache miss.

3. Design:

Programul pentru benchmark este alcătuit din 2 programe separate: un program care este folosit pentru a creea interfața grafică cu care interacționează utilizatorul și un program care aplică testele și prelucrează rezultatele obținute de la acestea.

Din interfața grafică utilizatorul va putea vedea specificațiile legate de procesorul și de memoria cache a PC-ului de pe care este rulat programul. Pe fereastra deschisă se regăsesc 3 butoane ce corespund la unul din testele ce le poate rula programul.

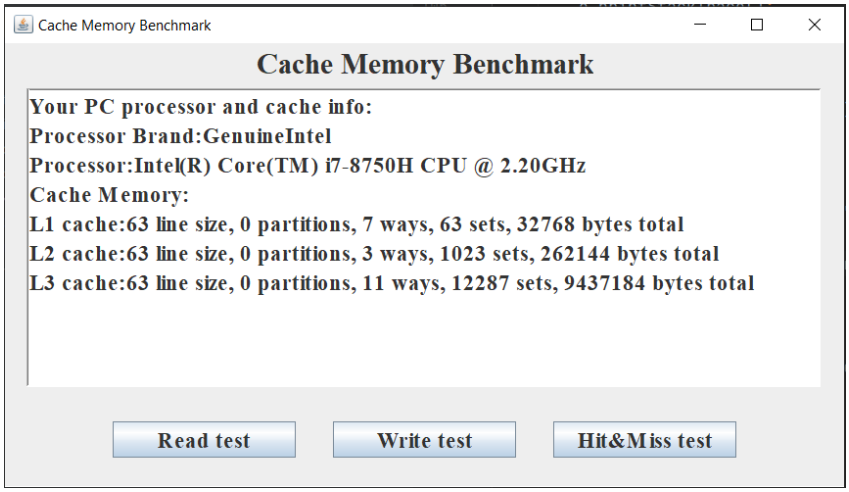


Fig 3.Fereastra principală a programului

Atunci când este apelat un test de citire sau scriere, se va rula executabilul celui de-al doilea program, se vor aplica și cronometra testele respective și se va genera un grafic ce va reprezenta performanța memoriei cache în funcție de diferite dimensiuni de date.

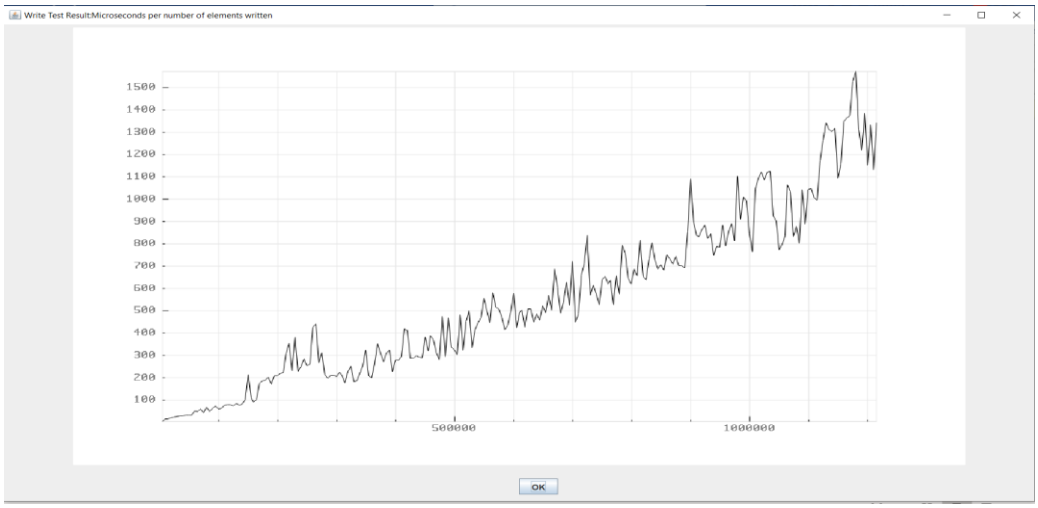


Fig 4. Fereastra rezultată în urma unui test de scriere

Atunci când este apelat testul de hit&miss, se va rula executabilul celui de-al doilea program, se vor aplica și cronometra teste de scriere pentru fiecare nivel al memoriei și rezultatele se vor nota datele experimentale într-un fișier text.

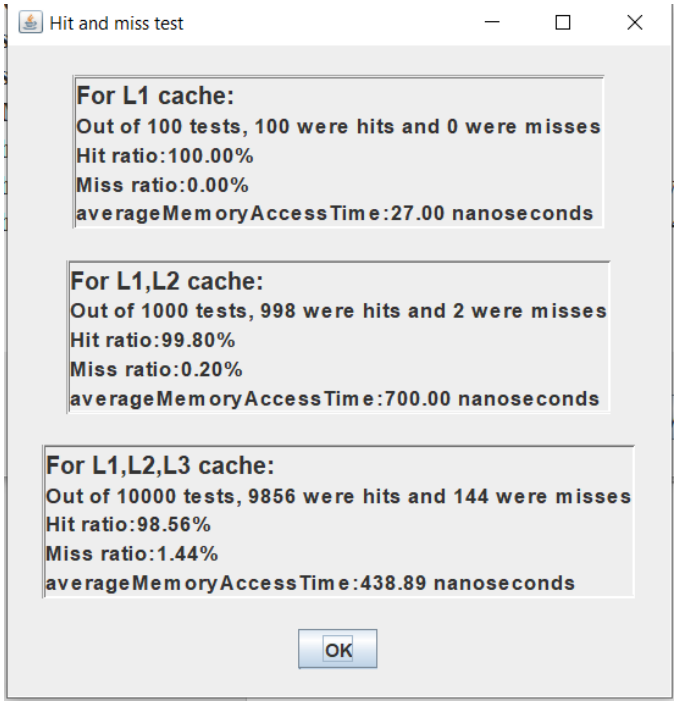


Fig 4. Fereastra rezultată în urma unui test de hit&miss

4.Implementare:

Aplicarea algoritmilor de testare a memoriei cache se face în cadrul programului scris în C/C++. La începutul fiecărei rulări se apelează funcția getPCInfo(). Funcția getPCInfo este folosită pentru a extrage date legate de PC-ul utilizatorului folosind funcția CPUID. Această funcție este apelată numai în contextul limbajului Assembly. Modul de lucru al funcției diferă în funcție de valoarea stocată în registrul EAX, iar rezultatele obținute în urma apelării funcției CPUID sunt stocate în regiștrii EBX, ECX și EDX. Atunci când este apelată funcția getPCInfo(), se va rula de mai multe ori funcția CPUID în diferite configurații pentru a obține date legate de procesor și memorie cache care sunt scrise într-un fișier text.

```
void getPCInfo() {
    FILE* f = fopen("PCinfo.txt", "w");
    char vendor[13];
    char processor[50];
    int a[4];
    __asm {
        mov eax, 0x0
        cpuid
        mov a[0], ebx
        mov a[4], edx
        mov a[8], ecx
    }
    short poz = 0;
    for (int i = 0;i < 3;i++) {
        memcpy(vendor + poz, &a[i], 4);
        poz += 4;
    }
    vendor[12] = '\0';
    fprintf(f, "Processor Brand:%s\n", vendor);
```

Pentru a rula un test de scriere, se va apela funcția testWritingCache(). Inițial se calculează memoria totală a memoriei cache. Testele constau din creearea de liste de variabile de tip Integer generate aleator. Dimensiunea acestor liste va începe de la 5000 de elemente și se va incrementa cu câte 5000 atăta timp cât memoria ocupată de lista respectivă este mai mică decât memoria totală a cache-ului. Pentru fiecare dimensiune se vor rula câte 5 teste unde se vor parcurge fiecare element din listă pentru al incrementa. Pentru măsurarea timpului necesar pentru a incrementa toate elementele din listă se va folosii ceasul din librăria chrono. Cronometrarea se începe înainte de parcurgere și se va încheia după finalizarea parcurgerii. Timpii sunt salvați în microsecunde.

```
void testWritingCache() {
    std::vector<double> x,y;
    int totalMemory = 0, buffer;
    for (int i = 0;i < cache;i++)
        totalMemory += lvlSize[i];
    totalMemory /= 8;
    double previousAverage=-1;

    for (int size = 5000;size <= totalMemory;size += 5000) {

        double average = 0;
        std::vector<double> numbers = numberGen(size, size/4);
        int* times = (int*)malloc(5 * sizeof(int));
        for (int j = 0;j < 5;j++) {
            auto begin = std::chrono::high_resolution_clock::now();
            for (long long i = 0;i < size;i++)
                numbers[i] += 1;
            auto end = std::chrono::high_resolution_clock::now();
            auto elapsed = std::chrono::duration_cast<std::chrono::microseconds>(end - begin);
            times[j]= elapsed.count();
        }
```

După terminarea celor 5 teste și stocarea timpilor de finalizare pentru fiecare, se va apela funcția fixNumbers() pentru a filtra din rezultate. Deoarece nu se poate influența direct modul de lucru al PC-ului, există situații în care procesul în care se execută programul să fie oprit temporar în favoarea rulării unui alt proces, lucru ce duce la întârzierea finalizării unor operații. Pentru a lua în calcul doar testele cele mai similare în rezultat, se folosește funcția fixNumbers() ce va șterge din lista de rezultate valorile care au număr de cifre diferit față de majoritatea valorilor din listă. De exemplu pentru lista de rezultate 65,78,101,100,70 se vor elimina valorile 101 și 100.

```
void fixNumbers(int* numbers) {

    for (int i = 0;i < 5;i++) {
        for(int j=0;j<4-i;j++)
            if (numbers[j] > numbers[j + 1]) {
                int aux = numbers[j];
                numbers[j] = numbers[j + 1];
                numbers[j + 1] = aux;
            }
    }
    int temp = numbers[4];
    int k = 0;
    while (temp) {
        temp /= 10;
        k++;
    }
    timeN = 5;
    int* frecv = (int*)calloc(k+1,sizeof(int));
    for (int i = 0;i < 5;i++) {
        temp = numbers[i];
        k = 0;
        while (temp) {
            temp /= 10;
            k++;
        }

        frecv[k]++;
    }
    int maj = frecv[1],frecvMaj=1;
    for (int i = 2;i < k+1;i++)
        if (frecv[i] > maj) {
            maj = frecv[i];
            frecvMaj = i;
        }
    int dif = 1;
    while (frecvMaj) {
        dif *= 10;
        frecvMaj--;
    }

    int i = 0;
    while (i < timeN) {
        if (numbers[i] < dif / 10 || numbers[i]>=dif) {
            for (int j = i;j < timeN - 1;j++)
                numbers[j] = numbers[j + 1];
            numbers = (int*)realloc(numbers, (timeN - 1)*sizeof(int));
            timeN--;
        }
        else
            i++;
    }
}
```

După preluarea rezultatelor cele mai relevante, se face media între acestea. Pentru a evita spike-uri de rezultate în urma testelor, se verifică dacă diferența dintre media curentă și cea anterioare nu este prea mare. În funcție de valoarea mediei anterioare, diferența de procentaj poate să difere de la 300% până la 6%. Dacă diferența dintre medii este prea mare, atunci nu se

trece la următoarea dimensiune. După terminarea testelor, lista rezultatelor și lista dimensiunilor este transmisă funcției makeGraph.

```
        for (int i = 0;i < timeN;i++)
            average += times[i];
        average /= timeN;
        if (previousAverage == -1 || abs(previousAverage - average) * 100 / previousAverage < percent) {
            previousAverage = average;
            int area = 0, temp = 10, lim[] = { 50,250,400,1000,1400 };
            while (temp < previousAverage) {
                temp += 10;
                if (temp >= lim[area])
                    area++;
            }
            switch (area)
            {
            case 0: {percent = 300; break;}
            case 1: {percent = 100; break;}
            case 2: {percent = 50; break;}
            case 3: {percent = 25; break;}
            case 4: {percent = 15; break;}
            case 5: {percent = 6; break;}
            }
            y.push_back(average);
            x.push_back(size);
        }
        else
            size -= 5000;
    }
    char* name=(char*)malloc(40*sizeof(char));
    sprintf(name, "MicrosecondsAveragePerSizeWriting");
    makeGraph(x, y, name);
```

Funcția makeGraph se folosește de funcțiile din biblioteca PbPlots.h pentru a genera un fișier de tip .png cu graficul mediilor testelor pentru fiecare dimensiune de date.

Funcția testReadingCache se folosește de un algoritm care este aproape identic cu cel folosit de funcția testWritingCache, cu anumite diferențe. Pentru fiecare dimensiune n se vor scrie într-un fișier n elemente de tip Integer generate aleator. Dimensiunile vor începe de la 5000 și se vor incrementa cu 10000 până când se va atinge sau depășii dimensiunea memorie cache.Timpul ce va fi măsurat reprezintă timpul necesar pentru a citii toate elementele din fișier. După realizarea celor 5 teste, se va apela funcția fixNumbers, se va face media rezultatelor, dar nu se va verifica diferența față de media anterioară deoarece spike-urile la testul de citire sunt mai puțin pronunțate.

```
void testReadingCache() {
    std::vector<double> x, y;
    long long totalMemory = 0;
    int buffer;
    for (int i = 0;i < cache;i++)
        totalMemory += lvlSize[i];
    totalMemory /= 8;
    double previousAverage = -1;

    for (int size = 5000;size <= totalMemory;size += 10000) {

        double average = 0;

        x.push_back(size);
        std::vector<double> numbers = numberGen(size, size/4);
        int* times = (int*)malloc(5 * sizeof(int));
        FILE* f = fopen("forReading.txt", "w+");
        for (int i = 0;i < size;i++)
            fprintf(f, "%.0f ", numbers[i]);
        for (int j = 0;j < 5;j++) {
            auto begin = std::chrono::high_resolution_clock::now();
            for (int i = 0;i < size;i++)
                fscanf(f, "%d", &buffer);
            fclose(f);
            auto end = std::chrono::high_resolution_clock::now();
            auto elapsed = std::chrono::duration_cast<std::chrono::microseconds>(end - begin);
            times[j] = elapsed.count();
        }
        fixNumbers(times);
        for (int i = 0;i < timeN;i++)
            average += times[i];
        average /= timeN;
        y.push_back(average);
    }
    char* name = (char*)malloc(40 * sizeof(char));
    sprintf(name, "MicrosecondsAveragePerSizeReading");
    makeGraph(x, y, name);
```

Testul pentru rata de cache hit și cache miss și pentru Timpul Minim de Accesare a Memoriei se face apelând funcția testHitMiss. Se va realiza câte un set de teste pentru dimensiunea totală ocupată de un nivel cache alături de dimensiunile celorlalte nivele anterioare. La începutul fiecărui set se alocă dinamic cantitatea de memorie pe care se vor realiza testele. La fiecare test se va alege o locație aleatoare din memoria alocată și se va realiza o operație de scriere prin incrementare. Timpul necesar realizării operației este măsurat în nanosecunde. Datorită faptului că operația se poate realiza mai rapid decât ar putea să măsoare programul, un timp măsurat de 0 nanosecunde va fi considerat ca fiind un cache hit, și un timp măsurat ca fiind mai mare sau egal cu 100 nanosecunde va fi considerat ca fiind un cache miss.

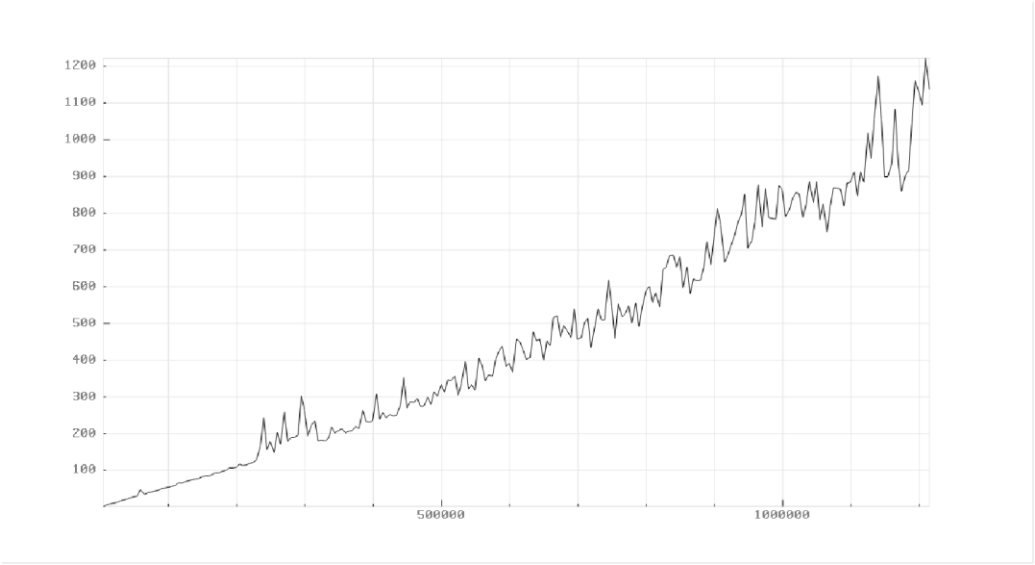
```
void testHitMiss() {
    std::vector<double> x, y;
    int totalMemory = 0, limit = 100, nrHit = 0,nrMiss=0,inc=0;
    int buffer, timeMiss = 0, hitTime = 0;
    FILE* f = fopen("HitMiss.txt", "w");
    for (int i = 0;i < cache;i++) {
        nrHit = 0;
        nrMiss = 0;
        timeMiss = 0;
        totalMemory += (lvlSize[i]/8);
        int* tester= (int*)calloc(totalMemory+100,sizeof(int));
        for (int j = 0;j < limit;j++) {
            int loc = rand() % (totalMemory + 100);
            auto begin = std::chrono::high_resolution_clock::now();
            tester[loc]++;
            auto end = std::chrono::high_resolution_clock::now();
            auto elapsed = std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin);
            int time = elapsed.count();
            if (time > 100) {
                nrMiss++;
                timeMiss += time - 100;
            }
            else {
                hitTime += time;
                nrHit++;
            }
        }
```

După finalizarea unui set de teste se calculează rata de cache hit și rata de cache miss, se calculează penalizarea pentru cache miss dacă au avut loc cache miss-uri, și se va calcula Timpul Mediu de Acces a Memoriei. Toate aceste valori se vor scrie într-un fișier.

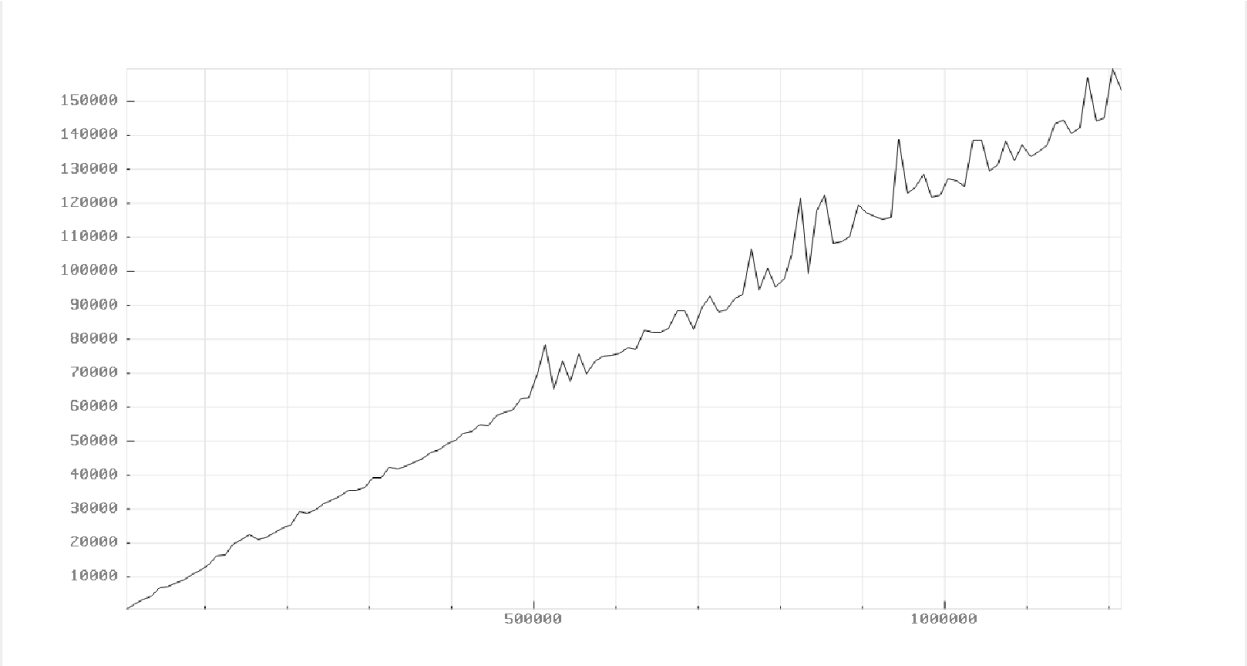
```
        float hitRatio, missRatio,averageMemoryAccesTime;
        hitRatio = (float)nrHit / limit;
        missRatio = 1 - hitRatio;
        fprintf(f,"%d %d %.2f %.2f", nrHit, nrMiss,hitRatio*100,missRatio*100);
        hitTime /= nrHit;
        if (timeMiss) {

            averageMemoryAccesTime = hitTime + missRatio * ((float)timeMiss / nrMiss);
            fprintf(f," %.2f %.2f\n", (float)timeMiss / nrMiss,averageMemoryAccesTime);
        }
        else {
            averageMemoryAccesTime = hitTime;
            fprintf(f," %.2f\n",averageMemoryAccesTime);
        }
        limit *= 10;
    }
    fclose(f);
}
```

5.Testare:

Aplicând o evaluare pentru scriere, aceasta se va realiza în aproximativ 10-15 secunde. Media timpilor de lucru pentru fiecare test variază în zonele anumitor dimensiuni de date, dar se poate observa o creștere accentuată a timpilor în microsecunde odată cu creșterea numărului de variabile de tip întreg. Variația între timpi devine mai mare spre finalul setului de teste, dar variațiile nu prezintă diferențe considerabile care să inducă în eroare utilizatorul.

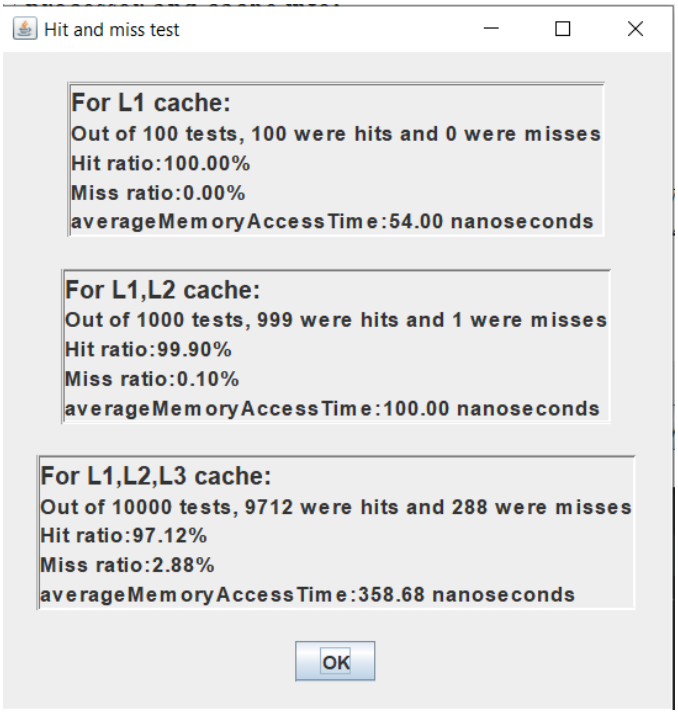


Aplicând o evaluare pentru citire, aceasta se va realiza în aproximativ 2 minute. La fel ca și la scriere, timpii de lucru variază între teste, dar, față de scriere, variațiile nu sunt la fel de accentuate. Creșterea timpilor în microsecunde este mai rapidă decât la scriere.



La anumite teste se pot observa spike-uri accentuate ce ajung la valori mai mari sau mai mici decât alte teste efectuate precedent. Acesta lucru este cauzat fie de teste care s-au efectuat mai lent din cauza valorilor pe care s-au realizat operatii, fie de teste unde procesul în care au avut loc a fost oprit pentru a se executa alte procese ale PC-ului.

Aplicând o evaluare pentru hit&miss se poate observa că pentru teste aplicate pe nivelul 1 al cache-ului nu s-a realizat niciun miss, pentru nivelul 1 și 2 s-a realizat un miss, iar pentru toată dimensiunea cache-ului au avut loc 288 de miss-uri din 10000 de operații. Rezultatele pot să mai difere între rulări, dar diferența dintre valori nu este foarte drastică.



6.Concluzii:

Scopul proiectului a fost de-a elabora un program de tip benchmark pentru testare a cache-ului procesorului. Programul elaborat pentru acest este unul simplu, compact, ce rulează o serie de teste simple pentru a putea cronometra și calcula performanțele memoriei cache. Deoarece fiecare PC are capabilități și specificații diferite, rezultatele pot să difere între PC-uri, dar testele rulate în contextul unui singur PC vor produce rezultate similare.

7.Bibliografie:

1) https://en.wikipedia.org/wiki/CPU_cache
2) https://en.wikipedia.org/wiki/Cache_replacement_policies
3) https://en.wikipedia.org/wiki/Cache_(computing)
4) https://en.wikipedia.org/wiki/Cache_hierarchy
5) https://www.felixcloutier.com/x86/cpuid

6) https://github.com/InductiveComputerScience/pbPlots