Tehnike optimizovanja koda

Bojan Nestorović 1084/2014
Matematički fakultet
Univerzitet u Beogradu

Šta su i čemu služe tehnike optimizovanja koda?

- Predstavljaju manje promene koda
- Za poboljšanje performansi
- Poželjno je merenje performansi prilikom svake promene
- Smanjuju “čitljivost koda”

Kategorije optimizovanja

- Logički pristup
- Optimizovanje petlji
- Transformiranje podataka
- Optimizovanje izraza
- Druga optimizovanja

1. Logički pristup

- Prestati sa proverom kad znate odgovor - IF
  - if (a > 5 && a < 10)

    ako smo utvrdili da je a < 5 onda ne trebamo pitati da li je a < 10, pa se ovo može zameniti sa:

    if (a > 5)
        if(a < 10)

    (neki jezici ovo rade automatski, kao npr c++)

1. Logički pristup

- Prestati sa proverom kad znate odgovor - FOR
  - flag = false;

```
for ( i = 0; i < count; i++)
        if ( niz[i] < 0 ) flag= true;
```

Kada se pronađe prvi negativan broj, iz petlje se može izaći jer nije potrebna dalja iteracija. To se može uraditi pomoću break(goto), korišćenjem while-a, postavljanjem uslova u for-u i postavljanjem brojača na kraj.

```
for ( i = 0; i < count; i++){
    if (niz[i] < 0 ) flag = true;
    break;
}
```

```
for ( i = 0; i < count && !flag; i++){
    if (niz[i] < 0 ) flag = true;
}
```

1. Logički pristup

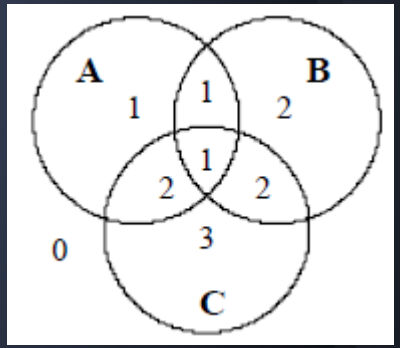
- Provera najverovatnijih slučajeva prvo
  - prvo treba proveravati one uslove koji imaju više šanse da budu tačni
  - pogotovo se koristi kod switch/case i if/then/else naredba
  - koristan pristup, ali otežava razumljivost koda

```
switch ( letter ) {
        case ‘a’, ‘e’, ‘i’, ‘o’, ‘u’: …
        ...
        case ‘w’, ‘z’, ‘x’, ‘q’: …
}
```

1. Logički pristup

- Korišćenje lookup tabela
  - korišćenje lookup tabela može biti brže nego prolaženje kroz mnogo logičkih uslova

```
if ( (a && !c) || (a && b && c) )
        category = 1;
else if ( (b && !a) || (a && c && !b))
        category = 2;
else if ( c && !a && !b)
        category = 3;
else category = 0;
```



1. Logički pristup

- Korišćenje lookup tabela
  - umesto toga može se koristiti:

```
int table[2][2][2] = {
// !b!c   !bc  b!c  bc
      0,    3,    2,    2     // !a
      1,    2,    1,    1     // a
```

otežava razumevanje, ali se to može nadoknaditi dobrom dokumentacijom

1. Logički pristup

- Korišćenje lenjeg izračunavanja
  - Ideja: izračunava se ono samo što je potrebno u tom trenutku, ostalo se zanemaruje i ostavlja za kasnije(ukoliko uopšte bude bilo potrebno)

    2<1 && a++

    Ovde nije potrebno izračunati a++ jer će uvek 2<1 biti netačno

2. Optimizovanje petlji

- Unswitching - izbegavati ponavljanje provere uslova u petljama za koje će uvek biti isti odgovor

```
for (i=0; i<n; i++)
        if (type == 1)      sum1 += a[i];
        else      sum2 += a[i];


if (type == 1)       for (i=0; i<n; i++)
                             sum1 += a[i];
else                 for (i=0; i<n; i++)
                             sum2 += a[i];
```

2. Optimizovanje petlji

- Spajanje više for petlji u jednu

```
for (i=0; i<n; i++)
        sum[i] = 0.0;
for (i=0; i<n; i++)
        rate[i] = 0.03;

for (i=0; i<n; i++) {
       sum [i] = 0.0;
       rate[i] = 0.03;
}
```

2. Optimizovanje petlji

- Odmotavanje petlje
  - ideja je da se smanji broj iteracija na štetu veličine koda

```
for (i=0; i<n; i++) {      a[i] = i; }
for (i=0; i<(n-1); i+=2) {
        a[i] = i;
        a[i+1] = i+1;
}
if (i == n-1)
       a[n-1] = n-1;
```

2. Optimizovanje petlji

- Smanjenje posla unutar petlji

```
for ( i = 0; i < rateCount; i++ ) {
      netRate[ i ] = baseRate[ i ] * rates*discounts*factors;  }

int temp = rates*discounts*factors;
for ( i = 0; i < rateCount; i++ ) {
      netRate[ i ] = baseRate[ i ] * temp;
```

2. Optimizovanje petlji

- Korišćenje kontrolnih promenljivih koje pomažu u prekidanju izvršavanja petlji ukoliko je tražena vrednost pronađena

```
i=0;
found = FALSE;
while ((!found) && (i<n)) {
        if (a[i] == testval)
       found = TRUE;
       else
        i++;
}
if (found) … //Value found
```

2. Optimizovanje petlji

- Stavljanje veće petlje unutar manje:

```
for (i=0; i<100; i++) // 100*1 = 100
    for (j=0; j<10; j++) // 100*10 = 1000
        dosomething(i,j);
```

1100 iteracija petlji

```
for (j=0; j<10; j++) // 10*1=10
    for (i=0; i<100; i++) // 100*10=1000
        dosomething(i,j);
```

1010 iteracija petlji

2. Optimizovanje petlji

- Smanjenje složenosti, tj. menjanje skupih operacija kao što su množenje sa sabiranjem

```
for (i=0; i<n; i++)
        a[i] = i*conversion;

sum = 0;
for (i=0; i<n; i++) {
        a[i] = sum;
        sum += conversion;
}
```

3. Transformiranje podataka

- Korišćenje integera umesto float-a:
  - matematičke operacije sa integerom su mnogo brže
  - koristiti u odgovarajućim situacijama gde se ne gubi tačnost
  - Koristiti float umesto double-a itd…
  - Testirati performanse i tačnost rezultata pri ovakvim konverzijama

3. Transformiranje podataka

- Korišćenje što je manje moguća dimenzija niza
  - samo ako je moguće pravljenje takvih promena bez opasnosti po podatke
  - jednodimenzioni nizovi umesto dvodimenzionih

```
for (i=0; i<rows; i++)
        for (j=0; j<cols; j++)
            a[i][j] = value;

for (i=0; i< rows * cols; i++)
      a[i] = value;
```

3. Transformiranje podataka

- Izbegavanje repetitivnog pristupa istom elementu:

```
for (i=0; i<rows; i++)
        for (j=0; j<cols; j++)
                a[j] = b[j] + c[i];

for (i=0; i<rows; i++) {
        temp = c[i];
        for (j=0; j<cols; j++)
                a[j] = b[j] + temp;
}
```

3. Transformiranje podataka

- Korišćenje dodatnih indeksa
  - sortiranje indeksa u nizu pre nego elemenata, pogotovo kada su elementi niza veliki
  - primer korišćenja dodatnih indeksa:

    C-ovska niska karaktera(string):

    da bi se izračunala dužina niske(string-a) ide se kroz nisku dok se ne dodje do terminalne nule

    Visual basic string:

    dužina string-a se nalazi kao prvi bajt sakriven u string-u,

    mnogo efikasnije od C-ovske niske.

3. Transformiranje podataka

- Korišćenje keširanja
- Čuvanje podataka pre nego ponovno izračunavanje
    - npr kod dužina niza, dužina string-a
    - izbegavanje ponovnog izračunavanja unutar petlji
    - izbegavanje ponovnog pristupa istom izračunavanj

4. Optimizovanje izraza

- Korišćenje algebarskih identiteta i manje skupih operacija
    - !a && !b je isto kao !(a || b), ali je ili mnogo jeftinije
    - sqrt(x) < sqrt(y) == x < y , ne koristi se dodatno izračunavanje
    - levo/desno šiftovanje umesto množenje/deljenje sa 2,4,8…
    - sabiranje umesto množenja, množenje umesto stepenovanja
    - efikasno polinomijalno izračunavanje - umesto običnog

      izračunavanja koristiti hornerovu šemu ili neki drugi efikasan

      metod:

      A*x*x*x + B*x*x + C*x + D = (((A*x)+B)*x)+C)*x+D

4. Optimizovanje izraza

- Inicijalizovanje u vreme kompajliranja
    - poznata konstanta parametra funkcije se može zameniti svojom vrednošću

```
log2val = log(val) / log(2);


const double LOG2 = 0.69314718;
log2val = log(val) / LOG2;
```

4. Optimizovanje izraza

- Izbegavanje korišćenja sistemskih ugrađenih matematičkih funkcija ukoliko nije potrebna velika tačnost
    - npr. ukoliko nam je potrebna vrednost logaritma kao ceo broj, ne koristiti sistemsku funkciju nego napraviti svoju
    - za log 2, može se računati broj šiftovanja potrebnih ukoliko je potreban celobrojni rezultat

4. Optimizovanje izraza

- Korišćenje tačnih tipova
    - izbegavanje nepotrebnih kastovanja
    - korišćenje konstantih brojeva sa pokretnim zarezom umesto float-a
    - korišćenje celobrojnih konstanti umesto int-a

4. Optimizovanje izraza

- Pre-izračunavanje rezultata
    - čuvanje podataka u tabelama/konstantama umesto računanja u toku izvršavanja programa
    - čak i velika izračunavanja pre izvršenja programa su bolja nego izračunavanja u toku izvršavanja
    - Npr:
        - stavljanje tabela u fajlove
        - korišćenje konstanti u kodu
        - korišćenje keširanja

4. Optimizovanje izraza

- Eliminisanje učestalih podizraza
    - svako izračunavanje što se ponavlja više puta može se izračunati jednom

```
a = b + (c/d) - e*(c/d) + f*(c/d);


z = c/d;
a = b + z + e*z +f*z;
```

5. Druga optimizovanja

- Korišćenje inline funkcija
    - Izbegavanje poziva funkcija i stavljanje tela funkcija direktno na mestu pozivanja(makroi)
    - Neki jezici poput c++ direktno podržavaju inline funkcije
    - Moderni kompajleri sami ponekad koriste inline funkcije

5. Druga optimizovanja

- Pisanje koda u jezicima niskog nivoa
    - delove kritičnog dela koda napisanog u programskom jeziku višeg nivoa prepisati u nekom jeziku nižeg nivoa, npr u asembleru
    - npr. jezik nižeg nivoa za python je c++, a za c++ asembler.
    - preporučljivo je raditi ovakvo optimizovanje samo u trenucima kada se ne piše nov kod
    - veoma dobri rezultati, ali može se lako napraviti greška