Ubrzavanje koda. Binarno traženje

BERNARD BRIŠKI

Uvod

- problem efikasnosti programa

- kompromis između efikasnosti i održavanja programa

- kako ubrzati kod?

1) lociranje „skupih” dijelova koda

2) minimalna modifikacija tih dijelova

Nulti primjer

- Chris Van Wyk izradio svoj program u C-u za analizu slika
- kompliciranije slike zahtijevale 10-ak minuta za obradu
- profiliranje programa – tehnika koja pokazuje koliko se vremena izvršava svaka funkcija
- problematična funkcija *malloc* (70% vremena)
- uzrok: najviše vremena se potroši kroz prolaske kroz memoriju da bi se našao određeni zapis
- rješenje: princip *cache memorije*
- 20 dodatnih linija programa – ubrzanje programa za 45% !

Primjer profiliranja

Func Time	%	Func+Child Time	%	Hit Count	Function
1413.406	52.8	1413.406	52.8	200002	malloc
474.441	17.7	2109.506	78.8	200180	insert
285.298	10.7	1635.065	61.1	250614	rinsert
174.205	6.5	2675.624	100.0	1	main
157.135	5.9	157.135	5.9	1	report
143.285	5.4	143.285	5.4	200180	bigrand
27.854	1.0	91.493	3.4	1	initbins

Složeniji primjeri

1) Ostatak pri cjelobrojnom dijeljenju

2) Funkcije, makronaredbe i *inline* kod

3) Sekvencijalno pretraživanje

4) Računanje sferne udaljenosti

Operator %

```
for i = [0, gcd(rotdist, n))
    /* move i-th values of blocks */
    t = x[i]
    j = i
    loop
        k = j + rotdist
        if k >= n
            k -= n
        if k == i
            break
        x[j] = x[k]
        j = k
    x[j] = t
```

-Problem rotiranja n-dimenzionalnog vektora za i mjesta ulijevo

-Uočimo da se ovaj odsječak

```
k = j + rotdist;
if (k >= n)
    k -= n;
```

može zamijeniti sljedećim odsječkom:

```
k = (j + rotdist) % n;
```

Operator %

-Testiranja su pokazala da je u pravilu gornji kod bolji od koda koji sadržava %

-rotdist = 1, vrijeme izvođenja programa se smanjilo sa 119n nanosekundi na 57n nanosekundi

-rotdist = 10, rezultati su bili iznenađujući – oba programa su trebala 206n nanosekundi

-Zašto su za **rotdist** = 10 ovakvi rezultati?

Za **rotdist** = 1, algoritam pristupa memoriji sekvancijalno i operator % dominira nad vremenom. Za **rotdist** = 10, algoritam pristupa svakom desetom podatku u memoriji i dohvaćanje podatka u cacheu postaje dominantnije

Funkcije, makronaredbe i *inline* kod

-funkcija **max** vraća maksimalnu vrijednost od 2

```
float max(float a, float b)
{   return a > b ? a : b; }
```

-drugi mogući način je makronaredba

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

Funkcije, makronaredbe i *inline* kod

```
float maxsum3(l, u)
    if (l > u)  /* zero elements */
        return 0
    if (l == u)  /* one element */
        return max(0, x[l])

    m = (l + u) / 2
    /* find max crossing to left */
    lmax = sum = 0
    for (i = m; i >= l; i--)
        sum += x[i]
        lmax = max(lmax, sum)
    /* find max crossing to right */
    rmax = sum = 0
    for i = (m, u]
        sum += x[i]
        rmax = max(rmax, sum)

    return max(lmax+rmax, maxsum3(l, m), maxsum3(m+1, u))
```

-problem: naći potpolje zadanog polja koje ima maksimalnu sumu

- makronaredba može smanjiti brzinu izvođenja programa sa 89n nanosekundi na 47 nanosekundi

- n = 10000, vrijeme izvođenja programa se poveća s 10 milisekundi na čak stotine sekundi, što je faktor usporavanja 10000

-Call-by-name semantika koju koristi ova makronaredba uzrokuje da se ona rekurzivno poziva više od 2 puta, i s time se povećava asimptotsko vrijeme izvršavanja.

- pisanje funkcije *inline* nije dalo neko ubrzanje

Sekvencijalno pretraživanje

-Problem: imamo niz od n podataka (radi jednostavnosti, uzmimo da su podaci *integeri),* ne nužno sortiranih. Potrebno provjeriti nalazi li se u nizu element t

```
int ssearch1(t)
    for i = [0, n)
        if x[i] == t
            return i
    return -1
```

- Prosječno vrijeme izvođenja ovog koda je 4.06n nanosekundi. Budući da je u tipičnom uspješnom pretraživanju potrebno pretražiti samo pola polja, program troši 8.1n nanosekundi ako se prođe kroz svaki element liste.

Sekvencijalno pretraživanje

-ako postavimo stražara na kraj liste, uštedit ćemo jednu usporedbu kroz koju svaki put moramo proći

```
int ssearch2(t)
    hold = x[n]
    x[n] = t
    for (i = 0; ; i++)
        if x[i] == t
            break
    x[n] = hold
    if i == n
        return -1
    else
        return i
```

-ubrzanje za 5%

Sekvencijalno pretraživanje

-odmotavanjem petlje dolazimo do ubrzanja od 56%

```
int ssearch3(t)
    x[n] = t
    for (i = 0; ; i += 8)
        if (x[i  ] == t) {          break }
        if (x[i+1] == t) { i += 1; break }
        if (x[i+2] == t) { i += 2; break }
        if (x[i+3] == t) { i += 3; break }
        if (x[i+4] == t) { i += 4; break }
        if (x[i+5] == t) { i += 5; break }
        if (x[i+6] == t) { i += 6; break }
        if (x[i+7] == t) { i += 7; break }
    if i == n
        return -1
    else
        return i
```

Računanje sferne udaljenosti

- Problem: Ulazni podatak je skup *S* od 5000 točaka na površini Zemaljske kugle; svaka točka je reprezentirana svojom geografskom dužinom i geografskom širinom. Nakon što odaberemo strukturu podataka po želji u kojoj će se nalaziti te točke, slijedi još jedan ulazni podatak – niz *A* od 20000 točaka, također reprezentiranih geografskim širinama i dužinama. Potrebno je za svaku točku tog niza naći točku u *S* koja je najbliža. Udaljenost se mjeri kao kut koji zatvaraju polupravci koji prolaze tim točkama čija je početna točka centar kugle
- Geografske širine i dužine predstavljaju problem pri računanju jer moramo koristiti trigonometrijske funkcije
- Dodavanjem x,y i z koordinata u strukturu točke, u mogućnosti smo zamijeniti brojne skupe trigonometrijske funkcije s jednostavnim aritmetičkim
- Iako ovaj pristup zahtjeva više memorije, ipak znatno ubrzava program

Binarno pretraživanje

- Problem: Imamo sortirano polje nekih podataka (ponovno ćemo radi jednostavnosti uzeti da je to polje *integera*). Zanima nas je li neki *integer t* element tog polja. Budući da je polje sortirano, tu činjenicu iskoristimo na sljedeći način: pogledamo je li srednji po redu element polja upravo naš *t.* Ako jest, našli smo *t* u polju i gotovi smo s pretraživanjem. Ako nije, prebacujemo se na jedan od sljedeća dva slučaja:

1) ako je srednji član niza veći od *t,* sortiranost niza nam govori da se *t* ne može nikako naći u gornjoj polovici niza, jer su u njoj samo veći elementi od srednjeg člana niza, a onda su veći i od *t.* Pretraživanje nastavimo po istom principu na donjoj polovici niza.

2) ako je srednji član niza manji od *t,* po sličnom rezoniranju, zaključujemo da pretraživanje nastavljamo na gornjoj polovici niza, jer se na donjoj polovici niza nalaze svi elementi manji od *t.*

Koju god polovicu izabrali, na njoj nastavljamo pretraživanje slično kao gore, uspoređujemo srednji član tog podniza sa *t* i onda ovisno o odnosu, ili završavamo pretraživanje ako smo našli naš element, ili nastavljamo pretraživanje na donjem ili gornjem podnizu podniza koji smo pretraživali

Binarno pretraživanje #1

```
l = 0; u = n-1
loop
    /* invariant: if t is present, it is in x[l..u] */
    if l > u
        p = -1; break
    m = (l + u) / 2
    case
        x[m] <  t:  l = m+1
        x[m] == t:  p = m; break
        x[m] >  t:  u = m-1
```

Binarno pretraživanje #2

```
l = -1; u = n
while l+1 != u
    /* invariant: x[l] < t && x[u] >= t && l < u */
    m = (l + u) / 2
    if x[m] < t
        l = m
    else
        u = m
/* assert l+1 = u && x[l] < t && x[u] >= t */
p = u
if p >= n || x[p] != t
    p = -1
```

- promatramo problem nalaženja prvog pojavljivanja *t* u nizu

- prva linija koda inicijalizira invarijante

- izvođenjem petlje, invarijante se mijenjaju *if-else* linijom koda

- nakon što se prekine izvođenje programa, znamo da ako se *t* nalazi u nizu, njegovo prvo pojavljivanje će biti na poziciji *u*.

- zadnje dvije linije koda postavljaju *p* na indeks prvog pojavljivanja *t* u nizu *x* ako se nalazi u nizu i na -1 ako se *t* ne nalazi u nizu.

Binarno pretraživanje #3

```
i = 512
l = -1
if x[511] < t
    l = 1000 - 512
while i != 1
    /* invariant: x[l] < t && x[l+i] >= t && i = 2^j */
    nexti = i / 2
    if x[l+nexti] < t
        l = l + nexti
        i = nexti
    else
        i = nexti
/* assert i == 1 && x[l] < t && x[l+i] >= t */
p = l+1
if p > 1000 || x[p] != t
    p = -1
```

- koristimo činjenicu da je n = 1000

- reprezentiramo granice sa *l* kao donjom granicom i sa *i* kao inkrementom tako da vrijedi *l + i = u.*

- u svakom koraku, *i* je potencija od 2.

- zbog n=1000, prije same petlje je potrebno osigurati da je radijus koji pretražujemo veličine 512, kao najveća potencija od 2 ispod 1000, tj. *l* i *l+i* skupa reprezentiraju radijus od 1 do 511 ili od 488 do 1000

Binarno pretraživanje #4

```
i = 512
l = -1
if x[511] < t
    l = 1000 - 512
while i != 1
    /* invariant: x[l] < t && x[l+i] >= t && i = 2^j */
    i = i / 2
    if x[l+i] < t
        l = l + i
/* assert i == 1 && x[l] < t && x[l+i] >= t */
p = l+1
if p > 1000 || x[p] != t
    p = -1
```

- ovaj program je modifikacija prethodnog

- *If* uvjet je pojednostavljen, varijable *nexti* više nema, isto kao što nema više ni onih linija koda koje su koristile *nexti*

Binarno pretraživanje #5

```
l = -1
if (x[511]   < t) l = 1000 - 512
    /* assert x[l] < t && x[l+512] >= t */
if (x[l+256] < t) l += 256
    /* assert x[l] < t && x[l+256] >= t */
if (x[l+128] < t) l += 128
if (x[l+64 ] < t) l += 64
if (x[l+32 ] < t) l += 32
if (x[l+16 ] < t) l += 16
if (x[l+8  ] < t) l += 8
if (x[l+4  ] < t) l += 4
if (x[l+2  ] < t) l += 2
    /* assert x[l] < t && x[l+2  ] >= t */
if (x[l+1  ] < t) l += 1
    /* assert x[l] < t && x[l+1  ] >= t */
p = l+1
if p > 1000 || x[p] != t
    p = -1
```

- ako malo bolje razmislimo, svako izvršavanje programa #4, za pretraživanje bilo koje vrijednosti u nizu, uvijek će se *i* kretati isto

- umjesto da uvijek računamo *i,* jednostavno možemo tu liniju koda dijeljenja *i* sa 2 zamijeniti sa stvarnim vrijednostima

Testiranje binarnog pretraživanja

- prva testiranja su pokazala da se za *n* = 1000 vrijeme izvođenja posljednjeg programa u odnosu na standardni program binarnog pretraživanja smanjilo sa 350 nanosekundi na 125 nanosekundi (testiranja takva da je *t* za prvi test bio na poziciji *x*[0], pa na poziciji *x*[1] itd)

- testiranja takva da se *t* pojavljuje na nasumičnim indeksima, a ne redom, dalo je drugačije rezultate: obično binarno pretraživanje je trebalo 418 nanosekundi, dok je profinjena verzija trebala 266 nanosekundi, što je ubrzanje za 36 posto.

- razlog ovakvom odstupanju je što su prva testiranja dala algoritmima preferirani uzorak komada memorije kojem pristupa i mogućnost predviđanja skoka.

- zaključak: čak se i binarno pretraživanje može ubrzati

Osnovni principi

Uloga efikasnosti. Mnogi zahtjevi softwarea su jednako važni kao efikasnost, neki čak i važniji. Potrebno je sačuvati brigu o efikasnosti za one programe kojima je to bitno.

Alati za mjerenje. Kada je efikasnost bitna, prvi korak je profiliranje. Profiliranje se vrši na kritičnim dijelovima koda, za ostale dijelove koda vrijedi postulat: "*Ako nije potrgano, ne treba popravljati*".

Pogodna implementacija. Problem efikasnosti se može riješiti na mnoge načine. Prije ubrzavanja koda, uvijek bi se trebali uvjeriti u to da drugi pristupi ne donose efikasnije rješenje

"Kada je više, manje". Ponekad ljudi pretjeraju u traženju efikasnijeg rješenja, i to dovodi do suprotnog. Trebalo bi pri svakom poboljšanju programa najprije provjeriti na reprezentativnom uzorku je li taj program uopće brži nego originalni.

Sažetak naučenih principa kroz primjere

1) Iskorištavanje čestih slučajeva – princip cachiranja u Van Wykovom programu

2) Iskorištavanje algebarskih identiteta – zamjena operatora % sa jednostavnom usporedbom

3) Rušenje hijerarhije procedure – zamjena funkcije sa makronaredbom

4) Kombiniranje testova i odmatanje petlje – postavljanje stražara u sekvencijalnom pretraživanju i povećanje inkrementa s 1 na 8

5) Povećavanje strukture – u primjeru sa računanjem sferne udaljenosti

6) Kombiniranje testova u binarnom pretraživanju je smanjilo broj usporedbi u unutarnjoj petlji. Iskorištavanje algebarskog identiteta je promijenilo granice tako da smo umjesto donje i gornje granice gledali donju granicu i inkrement. Odmatanje petlje unaprijedilo je program tako da zanemaruje svaki višak u petlji.

Za dodatne principe ubrzavanja …

- preporučujem pročitati Jon Bentley, Programming pearls (2. Izdanje), Appendix 4

Literatura

Bentley, Jon, Programming pearls (2. Izdanje), column 2, str 11-21

Bentley, Jon, Programming pearls (2. Izdanje), column 8, str 78-87

Bentley, Jon, Programming pearls (2. Izdanje), column 9, str 87-98