

Kódoptimalizálás

Fordítóprogramok előadás (A,C,T szakirány)

A fordítóprogramok szerkezete



A szintézis menete „valójában”

- 1 Optimalizálási lépések végrehajtása az eredeti programon (vagy annak egyszerűsített változatán).
- 2 Kódgenerálás.
- 3 Gépfüggő optimalizálás végrehajtása a generált kódon.

A kódoptimalizálás célja

- Hatékonyabb program létrehozása:
 - nagyobb sebesség
 - kisebb méret
- Ezek a célok időnként ellentmondanak egymásnak!

Követelmény a kódoptimalizálással szemben

- „Az optimalizált programnak ugyanúgy kell működnie, mint az eredetinek.” - Ez sok mindent jelenthet!
 - ugyanarra a bemenetre ugyanazt a kimenetet adja?
 - eseményvezérelt környezetben ugyanúgy viselkedik?
 - párhuzamos környezetben ugyanúgy viselkedik?
 - stb.
- Az adott nyelv szemantikája (jelentése) dönti el, hogy egy optimalizálási lépés megengedhető-e.

Kódoptimalizálási lépések osztályozása

- Mit optimalizálunk?
 - az eredeti programot (vagy annak egyszerűsített változatát): itt még vannak ciklusok, elágazások, kifejezésértékelés stb.
 - a tárgyprogramot: jellemzően assembly kódot
- Mi az átalakítás hatóköre?
 - lokális: kis programrészek átalakítása
 - globális: a teljes program szerkezetét kell vizsgálni
- Mit használ ki az átalakítás?
 - általános optimalizálási stratégiák: algoritmusok javítása
 - gépfüggő optimalizálás: az adott architektúra sajátosságait használja ki

Lokális optimalizálás: alablokk fogalma

Definíció: alablokk

Egy programban egymást követő utasítások sorozatát alablokknak nevezzük, ha

- az első utasítás kivételével egyik utasítására sem lehet távolról átadni a vezérlést
(assembly programokban: ahová a `jmp`, `call`, `ret` utasítások „ugranak”; magas szintű nyelvekben: eljárások, ciklusok eleje, elágazások ágainak első utasítása, `goto` utasítások célpontjai)
- az utolsó utasítás kivételével nincs benne vezérlés-átadó utasítás
(assembly programban: `jmp`, `call`, `ret` magas szintű nyelvekben: elágazás vége, ciklus vége, eljárás vége, `goto`)
- az utasítás-sorozat nem bővíthető a fenti két szabály megsértése nélkül

Alablokk szerepe az optimalizálásban

- A definíció következményei:
 - egy alablokknak pontosan egy belépési pontja van (az első utasítás)
 - az utasításai szekvenciálisan hajtódnak végre
 - ha rá kerül a vezérlés, akkor az összes utasítása pontosan egyszer hajtódik végre
- Lokális optimalizálás: egy alablokkon belüli átalakítások

Alablokkok meghatározása

- Jelöljük meg:
 - a program első utasítását
 - azokat az utasításokat, amelyekre távolról át lehet adni a vezérlést
 - a vezérlés-átadó utasításokat követő utasításokat
- Minden megjelölt utasításhoz tartozik egy alablokk, ami a következő megjelölt utasításig (vagy az utolsó utasításig) tart.

Alablokkok: példa

```
main: mov eax, [Cimke1]
      cmp eax, [Cimke2]
      jz igaz
      dec dword [Cimke1]
      inc dword [Cimke2]
      jmp vege
igaz: inc dword [Cimke1]
      dec dword [Cimke2]
vege: ret
```

Tömörítés

- Cél: minél kevesebb konstans és konstans értékű változó legyen!
- *konstansok összevonása*: a fordítási időben kiértékelhető kifejezések kiszámítása

Eredeti kód

```
a := 1 + b + 3 + 4;
```

Optimalizált kód

```
a := 8 + b;
```

- *konstans továbbterjesztése*: a fordítási időben kiszámítható értékű változók helyettesítése az értékükkel

Eredeti kód

```
a := 6;
b := a / 2;
c := b + 5;
```

Optimalizált kód

```
a := 6;
b := 3;
c := 8;
```

Azonos kifejezések többszöri kiszámításának elkerülése

Eredeti kód

```
x := 20 - (a * b);
y := (a * b) ^ 2;
```

Optimalizált kód

```
t := a * b;
x := 20 - t;
y := t ^ 2;
```

- Ez csak látszólag növeli a program méretét!
 - az `a*b` kifejezés kiértékelése is több assembly utasítás
 - a `t` változó lehet egy regiszter is
- Megvalósítás a gyakorlatban:
 - az utasításokból egy címkézett, irányított körmentes gráfot építünk,
 - ebből generálható az optimalizált kód

Változó továbbterjesztése

Eredeti kód

```
x := a + b;
y := x;
z := y;
```

Optimalizált kód

```
x := a + b;
y := x;
z := x;
```

- Ha az `y` változóra a továbbiakban már nincs szükség, akkor `y := x` törölhető! (De ez a törlés már globális optimalizálás...)
- Ez is megoldható az előzőleg említett gráfos módszerrel.

Ablakoptimalizálás

- Ez egy módszer a lokális optimalizálás egyes fajtáihoz.
- Ablak:
 - egyszerre csak egy néhány utasításnyi részt vizsgálunk a kódból
 - a vizsgált részt előre megadott mintákkal hasonlítjuk össze
 - ha illeszkedik, akkor a mintához megadott szabály szerint átalakítjuk
 - ezt az „ablakot” végigcsúsztatjuk a programon
- Az átalakítások megadása:
 - $\{minta \rightarrow helyettesítés\}$ szabályhalmazzal
 - a mintában lehet paramétereket is használni

Ablakoptimalizálás: példa

- ablak mérete: 1 utasítás
- szabályhalmaz:


```
{mov reg,0 → xor reg,reg
  add reg,0 → ; elhagyható}
```

Eredeti kód

```
add eax,0
mov ebx,eax
mov ecx,0
```

Optimalizált kód

Ablakoptimalizálás: példa

- ablak mérete: 1 utasítás
- szabályhalmaz:


```
{mov reg,0 → xor reg,reg
  add reg,0 → ; elhagyható}
```

Eredeti kód

```
add eax,0
mov ebx,eax
mov ecx,0
```

Optimalizált kód

```
; elhagyott utasítás
```

Ablakoptimalizálás: példa

- ablak mérete: 1 utasítás
- szabályhalmaz:


```
{mov reg,0 → xor reg,reg
  add reg,0 → ; elhagyható}
```

Eredeti kód

```
add eax,0
mov ebx,eax
mov ecx,0
```

Optimalizált kód

```
; elhagyott utasítás
mov ebx,eax
```

Ablakoptimalizálás: példa

- ablak mérete: 1 utasítás
- szabályhalmaz:


```
{mov reg,0 → xor reg,reg
  add reg,0 → ; elhagyható}
```

Eredeti kód

```
add eax,0
mov ebx,eax
mov ecx,0
```

Optimalizált kód

```
; elhagyott utasítás
mov ebx,eax
xor ecx,ecx
```

Tipikus egyszerűsítések ablakoptimalizáláshoz

- felesleges műveletek törlése: nulla hozzáadása vagy kivonása
- egyszerűsítések: nullával szorzás helyett a regiszter törlése
- nulla mozgatása helyett a regiszter törlése
- regiszterbe töltés és ugyanoda visszaírás esetén a visszaírás elhagyható
- utasításismétlések törlése: ha lehetséges, az ismétlések törlése

Globális optimalizálás

- a teljes program szerkezetét meg kell vizsgálni
- ennek módszere az **adatáram-analízis**:
 - Mely változók értékeit számolja ki egy adott alablokk?
 - Mely változók értékeit melyik alablokk használja fel?
- lehetővé teszi:
 - az azonos kifejezések többszöri kiszámításának kiküszöbölését akkor is, ha különböző alablokkokban szerepelnek
 - konstansok és változók továbbterjesztését alablokkok között is
 - elágazások, ciklusok optimalizálását

Kódkiemelés

Eredeti kód

```
if( x < 10 )
{
  a = 0;
  b++;
}
else
{
  b--;
  a = 0;
}
```

Optimalizált kód

```
a = 0;
if( x < 10 )
{
  b++;
}
else
{
  b--;
}
```

Kódsüllyesztés

Eredeti kód

```
if( x < 10 )
{
  x = 0;
  b++;
}
else
{
  b--;
  x = 0;
}
```

Optimalizált kód

```
if( x < 10 )
{
  b++;
}
else
{
  b--;
}
x = 0;
```

Mi lenne, ha itt *kódkiemelést* alkalmaznánk?

Ciklusok kifejtése

Eredeti kód

```
for( int i=0; i<4; ++i )
{
  a += t[i];
}
```

Optimalizált kód

```
a += t[0];
a += t[1];
a += t[2];
a += t[3];
```

Mérlegelni kell, hogy a méret és a sebesség mennyire fontos...

Parciális kifejtés

Eredeti kód

```
for( int i=0; i<4; ++i )
{
  a += t[i];
}
```

Optimalizált kód

```
for( int i=0; i<4; i+=2 )
{
  a += t[i];
  a += t[i+1];
}
```

Ciklusok összevonása

Eredeti kód

```
for( int i=0; i<4; ++i )
{
    t1[i] = 0;
}
for( int i=0; i<4; ++i )
{
    t2[i] = 1;
}
```

Optimalizált kód

```
for( int i=0; i<4; i++ )
{
    t1[i] = 0;
    t2[i] = 1;
}
```

Frekvenciaredukálás

- költséges utasítások „átköltöztetése” ritkábban végrehajtódó alaplókkba
- példa:
 - ciklusinvariánsnak nevezzük azokat a kifejezéseket, amelyeknek a ciklus minden lefutásakor azonos az értékük
 - a ciklusinvariánsok (esetenként) kiemelhetők a ciklusból

Eredeti kód

```
cin >> a;
cin >> h;
while( h > 0 )
{
    cout << h*h*sin(a);
    cin >> h;
}
```

Optimalizált kód

```
cin >> a;
cin >> h;
double s = sin(a);
while( h > 0 )
{
    cout << h*h*s;
    cin >> h;
}
```

Erős redukció

- a ciklusban lévő költséges művelet (legtöbbször szorzás) kiváltása kevésbé költségessel

Eredeti kód

```
for( int i=a; i<b; i+=c )
{
    cout << 3*i;
}
```

Optimalizált kód

```
int t1 = 3*a;
int t2 = 3*c;
for( int i=a; i<b; i+=c )
{
    cout << t1;
    t1 += t2;
}
```