Optimalizálás

Optimizálás

Szerzünk valódi adatokat

1. Mérünk
2. Profilozunk
3. Gyorsítunk
4. Mérünk
5. Megnézzük, hogy elég jó-e

Miért lehet valami lassú?

1. Rossz algoritmus
2. Rossz szerkezetű utasítások
3. Rossz memória kezelés

1. Rossz algoritmus

Számolás

- Megfelelő O() algoritmus?
- Nem lehet gyorsítani dinamikus programozással?
- Tényleg kell ennyi mutex?
- Sok kis fájlolvasás
- Rossz hálózati kezelés
- Túl sok szál
- Sok apró adatbázis művelet?
- ...

1. Rossz algoritmus

Adatszerkezetek:

Hogy akarjuk használni?

- TreeMap vs HashMap?
- Láncolt lista vs Vector?
- std::string vs char*
- ...

2. Utasítások

Mi van a program alatt?

- CISC processzor
- Pipeline + Branch prediction
- Vectoros/SIMD utasítások
- Speciális utasítások

Használjuk a beépített függvényeket

memcpy
memmove
memcmp
memset

strcpy
strchr, strstr

math.h függvények

Ne használjunk gyorsítandó kódban

- Try - catch - throw
- RTTI
- Float-double konverzió
- Bitfieldek
  - Lassú a hozzáférés
- Szinkronizációs objektumok
- Virtuális függvények
  - Nem lehet eldönteni fordítási időben, hogy mi hívódik
  - Plusz idő a pointer dereferencia
- Dinamikus linkelés
- Áthívás cpp fájlok között
  - A fordító nem tud optimalizálni

Ne használjunk gyorsítandó kódban

- Pointerek változókra
  - A fordító néha nem tudja kimatekozni, hogy hova mutat a pointer
- Felesleges mellékhatások
  - Fordító optimalizálást akadályoz
- Objektumok inicializálása
- Memória másolás feleslegesen
  - Főleg copy constructor, operator=
- Memória foglalás/felszabadítás
  - Lásd inplace new

Loop unrolling

Ciklus helyett leírjuk többször az utasításokat:

```
for (int i = 0; i < 4; ++i) {
  doSomething(i);
}
```

helyett:

```
doSomething(0);
doSomething(1);
doSomething(2);
doSomething(3);
```

Loop unrolling

Ciklus helyett leírjuk többször az utasításokat:

```
for (int i = 0; i < 4; ++i) {
  doSomething(i);
}
```

Meg tudja csinálni a fordító is!

Kis eljárásoknál lehet, hogy gyorsabb unrolling nélkül

Ha nem fix a ciklusszám

```
for (int i = 0; i < n; ++i) {
   doSomething();
}
```

helyett:

```
switch (n) {
   case 0: doSomething();
   case 1: doSomething();
   case 2: doSomething();
   case 3: doSomething();
}
```

Eljáráshívás elkerülése

inline-olás / makrók

ne hívjunk felesleges függvényeket

Ha mégis kell:

- Nagy objektumok referencia/pointer szerinti átadása
- Template-ek polimorfikus függvények helyett
- fastcall
- 64 bites fordítás (hatékonyabb paraméter átadás)

Lookup tables

Összetett függvényeket előre ki lehet számoltatni

Köztes értékeket interpolálni

A számolás történhet program induláskor vagy még buildeléskor.

Assembly betétek

Nem hordozható, de gyors

Vannak nagyon spéci x86 utasítások:

pl:

F2XM1: *2^x -1*

FYL2XP1: *y*log2(x+1)*

Pár apróság
A szorzás gyorsabb mint az osztás

double a,b;

a = b / 1.2345

helyett:

a = b * (1.0 / 1.2345) // Fordítási időben számolható

Pár apróság
int <-> floating point

float -> int lassabb mint az **int -> float**

unsigned int -> float lassabb mint **signed int -> float**

ha u<2^31

```
unsigned int u;
double d;
d = (double)(signed int)u;
```

Pár apróság
out of order execution

y = a + b + c + d;

Függőségek vannak

y = ((a+b)+c)+d;

y = (a + b) + (c + d);

Nincsenek függőségek

(a + b) és (c + d) majdnem egyszerre hajtható végre

SIMD utasítások

Single **I**nstruction **M**ultiple **D**ata

MMX, SSE*, AVX

Nagy(64, 128, 256 bit), többféleképpen felhasználható regiszterek.

Külön utasítások vannak rá **`<*mmintrin.h>`**

A fordítók automatikusan is vektorizálhatnak

PI:

```
struct RGBA {
   unsigned char r;
   unsigned char g;
   unsigned char b;
   unsigned char a;
};

struct RGBA color = ...;
uint32_t* p = (uint32_t*)&color;
*p = 0xFF0000FF;
```

Generált assembly túrás

```
int i;
for (i = 0; i < 100; i++) {
  a[i] = r + i/2;
}
```

```
$B1$2:                              ; top of loop
 mov ebx, eax                       ; compute i/2 in ebx
 shr ebx, 31                        ; shift down sign bit of i
 add ebx, eax                       ; i + sign(i)
 sar ebx, 1                         ; shift right = divide by 2
 add ebx, DWORD PTR [edx]           ; add what r points to
 mov DWORD PTR[ecx+eax*4],ebx         ; store result in array
 add eax, 1                         ; i++
 cmp eax, 100                       ; check if i < 100
 jl $B1$2                             ; repeat loop if true
```

```
int i;
int Induction = r;
for (i = 0; i < 100; i += 2) {
  a[i] = Induction;
  a[i+1] = Induction;
  Induction++;
}
```

```
$B2$2:                          ; top of loop
 mov DWORD PTR [eax], ecx       ; a[i] = Induction;
 mov DWORD PTR [eax+4], ecx     ; a[i+1] = Induction;
 add ecx, 1                     ; Induction++;
 add eax, 8                     ; point to a[i+2]
 cmp edx, eax                   ; compare with end of array
 ja $B2$2                         ; jump to top of loop
```

Memória

Mi van a program alatt?

- Regiszterek
  - Helyben van 8
  - 8/16 integer register
  - 8/16 floating point register
- Cache
  - Pár órajelciklus beolvasni
  - Elő is tud olvasni
- Kód cache
  - Ebben vannak a végrehajtható utasítások
- Memória
  - **~100 órajelciklus beolvasni**

Tárolási típusok

- Globális / statikus:
  - Külön memóriarészben tárolva
- Lokális:
  - Vagy regiszterben vagy cache-ben tárolva
  - alloca
- Dinamikus:
  - new, delete, malloc, free
  - lassú
  - töredezett memóriát okozhat
  - pointeren keresztül férünk hozzá a memóriához
  - Lehet rossz az align
  - De: Lehet nagy memóriaterületeket foglalni.

Adatszerkezetek bejárása

Ne legyen cache miss

int vector[N][M]

Sorfolytonosan járjuk be, ne "oszlop szerint"

Bízzunk a C++ iterátorokban

Adatszerkezetek illesztése

- Igazítsuk a cache határra az objektumok kezdetét
- Igazítsuk cache méretre az objektumok méretét

Megcsinálja nekünk a fordító.

Kivéve ha memória felhasználásra optimalizál.

Kód tárolás

- Legyen a code cache hatékony
- Összefüggő kód közös cpp fájlba
- Öröklődésnél figyelni
- Linker scriptek

Mit tud helyettünk a fordító megcsinálni?

Alapszabály: Amit ki lehet találni fordítási időben azt ki tudja optimalizálni.

- Függvény inline-olás
  - Kitalálja hogy optimális
- Konstansok előre kiszámítása
  - a = b + 2.0 / 3.0; helyett a = b + 0.66666666...
  - Zárójelezésre figyelni!
  - Egyszerűbb függvényeket is végre tud hajtani
- Pointerek helyettesítése
  - void Plus2(int& p) {p=p+2;}
  - Csak akkor, ha függvénynek nincs mellékhatása
- Gyakori kifejezések kiértékelése

Mit tud helyettünk a fordító megcsinálni?

- Jumpok eltűntetése
  - Felesleges if ágak eltűntetése
  - returnok, utasítások if ágakba behúzása
- Változók registerbe pakolása
- Utasítások átrendezése
  - A processzor különböző részei dolgozhatnak egyszerre
- Live range analysis
  - Kitalálja, hogy meddig él egy változó
  - Ha már nincs használva a regiszterét/memóriáját használhatja másik változónak
  - {}-vel segíthetünk

Mit tud helyettünk a fordító megcsinálni?

- Komoly algebrai átrendezések is, pl:
  - (a&b)|(a&c) = a&(b|c)
  - a*x*x*x + b*x*x + c*x + d = ((a*x+b)*x+c)*x+d
- Ciklus invariáns dolgok cikluson elé tétele

Amiről azt gondolnánk, hogy lassú, de nem

- Adatok pointerekkel elérése. Így is-úgy is stack relatív lesz.
- A double ugyanolyan gyors mint a float

Refactoring

Refactoring

A meglévő forráskód belső átalakítása a meglévő viselkedés megtartása mellett.

Egyszerű átalakítások sorozata, ami jobb kódot eredményez.

Mivel az átalakítások egyszerűek, nehezebb elrontani.

A rendszer végig működőképes.

Refactoring

Mikor?

- karbantartás:
  - új funkció
  - hiba javítás (de nem egyszerre!)
- ha meg akarunk/kell érteni valamit

De mikor ne?

- kiadás előtt (általában ilyenkor változtatjuk, de ne!)
- hiba van és nem tudjuk, hogy miért

Karbantartás

A karbantartás oka:

- tudjon többet, mást (funkcionális)
- legyen gyorsabb, bírjon többet (nem funkcionális)
- hibajavítás

A karbantartás akadálya:

- régi a forráskód

Régi forráskód / legacy code

- mindig ezzel fogunk dolgozni és ebből van több
- nem akarunk foglalkozni vele, valaki másé a felelősség
- mi írtuk, de már nem emlékszünk rá
- rosszul dokumentált (megjegyzések, kódszerkezet)
- nincs rá teszt

- működik! (nem érdemes kidobni)

Refactoring és a tesztelés

Egyszerű, egyenértékű átalakítások.

Mi a bizonyíték? Tesztek:

1. Minden teszt zöld
2. Változtatunk a kódon
3. Teszt javítás!! (ha nem fordul, vagy megváltozott a szerkezet)
4. Minden teszt zöld

Ha nem létezik még teszt írjunk.

Ha hibát találunk közben arra is írjunk.

Refactoring és a jobb kód

Az átalakítások célja a jobb kód.

Jobb, ha:

- megszüntettünk egy codesmellt
- javítottunk a metrikán

Általában igaz, hogy nem kevesebb, de egyszerűbb kódunk lesz.

Általános módszerek

- Átnevezés (rename)
- Kiemelés (extract)
- Beágyazás (inline, encapsulate)
- Mozgatás (pull up, push down, move)
- Csere (replace)

http://refactoring.com/catalog/

Code metrics

Code metrics

- tárgyilagos mérések
- személyes vélemények
- szükséges, de nem elégséges feltételek
- felhívják a figyelmet lehetséges hibákra

LoC - Lines of Code

Erősen vitatott.

- függ a nyelvtől, stílustól.
- a több sor nem feltétlenül jelent nehezebb megértést

A teljes projektre értemezve lehet érdekes. Átfogó képet ad a méretről.

Folyamány: kód/megjegyzés arány, kód/teszt arány

ABC metrika

Assignments (A), Branches (B), Conditions (C)

|ABC| = sqrt((A*A)+(B*B)+(C*C))

Az értékadások, függvényhívások (és példányosítások) valamit a feltételvizsgálatok száma.

A függvény bonyolultságáról ad egy nagyságrendet.

Ciklomatikus komplexitás

Egy függvényben bejárható utak számáról, így a bonyolultságról ad képet.

$M = E - N + 2P$, ahol

E: egy blokkból egy másikba vezető lépések száma

N: a blokkok száma

P: a kilépési pontok száma

(McCabe, 1976)

Lefedettség / Code coverage

- Az összes függvény meghívódik egyszer
- Az összes függvény az összes helyről
- Minden utasítás (pl. egy if legalább egyszer)
- Minden ág egyszer (egy if összes utasítása)

- Minden ág az összes lehetséges módon (minden feltétel legalább egyszer igaz és egyszer hamis)
- Az összes útvonal

Code smells

Code smells

A "beteg" kód "tünetei".

Egyszerű dolgok elnevezve.

Felhívják a figyelmet lehetséges hibákra.

Ha úgy érzzük, hogy valami hack, akkor az is.

Kódkettőzés

Tünetek és megoldások:

- az osztály két metódusa is ugyanazokat a lépéseket hajtja végre
  => függvény kiemelés
- két alosztály ugyanazt az adattagot definiálja
  => adattag kiemelés
- két alosztály közel azonos lépéseket végez egy függvényben
  => sablonfüggvény

Hosszú függvény

Tünet:

egy függvény nem fér ki a képernyőre és/vagy nehéz olvasni, értelmezni

Megoldás:

emeljük ki összetartozó részeket új függvényekbe

Sok paraméteres függvény

Tünetek és megoldások:

- a függvénynek mindig több összetartozó paraméterre van szüksége
  => paraméter objektum
- a függvénynek paraméterként egy másik függvény visszatérési értékét adjuk
  => paraméter cseréje függvényhívással
- a függvény viselkedése egy paramétertől függ (pl. boolean flag)
  => paraméter helyett új függvények

Literálok mértéktelen használata

Tünet:

A kód sok pontján (feltételek, ciklusok, kifejezések) literálok szerepelnek

Megoldás:

Cseréljük le a "varázs számokat" nevesített konstansokra vagy függvényhívásokra.

Megjegyzések

Tünet:

egy függvényhez túl sok megjegyzés tartozik.

Megoldás:

alakítsuk át a kódot, hogy önleíró legyen és ne legyen szükség a megjegyzésekre

Eszközök

- http://google-styleguide.googlecode.com/svn/trunk/cpplint/cpplint.py
- http://cppcheck.sourceforge.net/
- clang

Források

- http://www.agner.org/optimize/optimizing_cpp.pdf
- http://www.codinghorror.com/blog/2006/05/code-smells.html
- http://www.refactoring.com/
- http://www.refactoring.com/rejectedExample.pdf
- BerkeleyX: Software as a Service https://www.edx.org/courses/BerkeleyX/
-