Optimalizace

Slova proroka

- Předčasná optimalizace je zdrojem veškerého zla.
  - Donald Ervin Knuth

Úrovně optimalizace

3 základní úrovně optimalizace:

- Nejvyšší úroveň: Volba algoritmu (quicksort vs. bublinkové třídění)
- Střední úroveň: překladač s pomocí programátora
- Nejnižší úroveň: volba instrukcí strojového kódu – překladač dělá automaticky, i při vypnutých optimalizacích

Co má smysl?

- Nejvyšší úroveň:
  může mít zásadní dopad na výkon aplikace
  (Jediná opravdu významná úroveň)
- Střední úroveň:
  (Má někdy význam, ale jen menší)
- Nejnižší úroveň:
  Prakticky bezvýznamná

Nejnižší úroveň

- Příklad: instrukce

```
and dword ptr[iProm], 0
```

je menší o 3B, ale je 3× pomalejší než instrukce se stejným významem

```
mov dword ptr[iProm], 0
```

Střední úroveň

- Použití registrů
- Šíření konstant a kopií
- Eliminace mrtvého kódu
- Eliminace společných podvýrazů
- Optimalizace cyklů
- Plánování instrukcí
- Redukce síly
- Rozvoj inline
- Sdílení řetězců
- Vypuštění rámce zásobníku
- Vypuštění kontroly zásobníku
- Překryvy v zásobníku
- Předpokládáme, že nedochází k aliasingu
- Sestavování na úrovní funkcí

Registrové proměnné

- Klíčové slovo register
- Dnešní překladače nabízejí 3 možnosti:
  - Ignorovat a dělat podle úvahy překladače
  - Dodržet podle možností
  - Zakázat
- Optimální (a implicitní) je první možnost

Šíření kopií a konstant

- Šíření konstant: náhrada

```
x = 234;   →        x = 234;
y = x;              y = 234;
```

ušetří zbytečný přístup do paměti

- Šíření kopií: náhrada

```
i = par;            i = par;
Func(i);   →        Func(par);
i = j;              i = j;
```

První příkaz je pak zbytečný a lze ho odstranit

Eliminace mrtvého kódu a paměti

- Předchozí kód lze zkrátit na

```
Func(par);
i = j;
```

- `i` byla do té doby „mrtvá paměť"
- Mrtvý kód: nemůže být spuštěn
- Obvykle produkt jiné optimalizace
- Odstranění: zmenšení výsledného kódu

Eliminace společných podvýrazů

- Základní schéma:

```
x = y + z;
w = y + z;
```

lze nahradit

```
pom = y+z;
x = pom;
w = pom;
```

někdy zmenšení, téměř vždy urychlení

Optimalizace cyklů

- Eliminace konstantního kódu:

```
for(i = 0; i < 10; i++)
  A[i] = x+y;
```

lze nahradit

```
pom = x+y;
 for(i = 0; i < 10; i++)
  A[i] = pom;
```

Plánování instrukcí

- Pro superskalární procesory (více jader)
- Přerovnání instrukcí tak, aby nezávislé mohly běžet paralelně

```
add ax, i       ; instrukce A
movsx eba, ax   ; instrukce B
xor ecx, ecx    ; instrukce C
```

A a B nemohou běžet paralelně, proto se přehodí pořadí na A, C, B

Redukce síly

- Redukce aritmetické složitosti
- Náhrada dělení mocninou 2 bitovým posunem vpravo:

```
y = y/16  → y = y >> 4
```

Redukce síly: na úrovni instrukcí

```
mov    ecx, 16                ; 1
mov    eax, dword ptr[y]      ; 1
cdq                           ; 3
idiv ecx                      ; 46
mov dword ptr[y], eax         ; 1

------------------------------

sar    dvord ptr [y], 4       ; 3
```

Rozvoj inline

- Různé problémy: nelze získat adresu, nehodí se pro virtuální metody
- Překladač může odmítout rozvoj udělat nebo ho může udělat, ale zároveň vytvořit instanci funkce
- To může způsobit nárůst výsledného kódu a tím zpmalení

Sdílení řetězců

- Stejné řetězce na různých místrch programu lze sloučit do jednoho
- Nebezpečí: Změna jednoho způsobí změnu všech

Vypuštení rámce zásobníku

- Rámec zásobníku – v podstatě definice lokálních souřadnic v zásobníku pro podprogram
- ```
  push ebp                    ; vstup
  mov ebp, esp
  sub esp, lokální_prom
  ```

- ```
  mov esp, ebp                ; výstup
  pop ebp
  ```

- **Vypuštění (zůstane pouze vytvoření lokálních proměnných): zkrátí se kód i čas**

Kontrola zásobníku

- Alokace velkého lokálního pole: paměť se přidělí, ale nepotvrdí („nezkomituje“)
- Kontrola: program zavolá funkci, která přistoupí k jednomu prvku pole na každé stránce a tím způsobí potvrzení (potvrdí se stránka následující za použitou)
- Je-li pole zpracováváno postupně, není třeba

Překryvy v zásobníku

- Pro lokální proměnné, jejichž dobu života se nepřekrývají, se použije stejné místo
- ```
  int i;
  f(i);
  // ..
  int j;
  //  zde už se nepoužívá i – proto
  // překladač uloží j na stejné místo
  // jako i
  ```
  ```

Sestavování na úrovni funkcí

- Obvykle linker připojuje celou knihovnu
- Může ale připojit pouze opravdu použité funkce
- Zmenšení kódu, zvýšení rychlosti