Dead Code Elimination

LLVM

Dead Code Elimination

1. Dead Code Elimination (DCE) je optimizaciona tehnika koja uklanja instrukcije i delove koda koji ne uticu na konacni rezultat programa, cime se smanjuje velicina i kompleksnost generisanog IR-a.

2. Glavni cilj: povecanje efikasnosti i performansi programa bez promene semantike izvornog koda.

3. ***Analiza zivosti podataka*** (liveness analysis)

4. U LLVM-u se DCE tipicno implementira kao FunctionPass, gde se analiza i brisanje instrukcija vrsi nezavisno za svaku funkciju, koristeci podatke o zavisnostima izmedju vrednosti i operanada (lokalna varijanta)

Implementacija - baza

- Osnova implementacije se zasniva na sledecem:
  “Ukoliko je instrukcija non-void tipa i nije call instrukcija, smatraj je inicijalno kao dead instrukciju - ukoliko se pokaze suprotno tokom dalje analize, sacuvaj je”
- Za promenljivu kazemo da je ziva ukoliko je njena vrednost “citana” odnosno koriscena u nekoj drugoj instrukciji.
- Implementacija se svodi na prolaz kroz sve instrukcije, markiranje zivima one koje se koriste kao operandi nekih drugih funkcija. U slucaju store instrukcija, proglasavamo njen nulti operand zivim.
- Load instrukcija uvodi dodatne zavisnosti izmedju promenljivih - sacuvane u posebnoj mapi VariablesMap.

Prosirenje 1: Dead Store Elimination

- Osnovna implementacija bi ostavila sve store instrukcije ciji je prvi operand ziva promenljiva, bez obzira na to da li su medjusobno prepisane.
- Prosirenje se zasniva na sledecem: “*Na istu memorijsku lokaciju može da se upiše više puta, ali samo poslednja upisana vrednost pre sledećeg load-a ima smisla. Sve ranije upisane vrednosti — ako nisu pročitane — mogu se bezbedno obrisati.*“
- Ovaj propust bi ostavio sve dead store instrukcije u nasem optimizovanom output.ll fajlu.

Implementacija prosirenja

1. Prolazimo redom kroz sve instrukcije, za svaku memorijsku adresu, pratimo koja instrukcija predstavlja njen “poslednji store” uspomoc mape LastStore
2. Kada naidjemo na novi store, proveravamo da li za tu adresu postoji store
3. Ako postoji, brisemo taj stari: od njega do trenutne instrukcije se nije desilo nijedno citanje (load).
4. Postavljamo LastStore[ptr] na trenutnu store instrukciju
5. Kada naidjemo na load instrukciju, uklanjamo poslednju store instrukciju iz mape njenog parametra - ta vrednost je procitana i ne treba da se brise
6. Na kraju prolaza, brisemo sve store instrukcije koje su ostale u LoadStore jer ni njihove vrednosti nisu nikada procitane (bile bi sklonjene iz mape)

Primer

```
int main() {
    int x = 77;          // provera starih funkcionalnosti
    int a = 5;
    a = 10;              // ← ovaj store prepisuje prethodni (a=5)
    a = 15;              // ← ovaj store prepisuje prethodni (a=10)
    int b = 20;
    b = 25;              // ← ovaj store prepisuje prethodni (b=20)
    int c = a + b;       // ← koristi poslednje vrednosti a i b
    return c;
}
```

Optimizovani output.ll

```
define dso_local noundef i32 @main() #4 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 15, ptr %1, align 4
  store i32 25, ptr %2, align 4
  %4 = load i32, ptr %1, align 4
  %5 = load i32, ptr %2, align 4
  %6 = add nsw i32 %4, %5
  store i32 %6, ptr %3, align 4
  %7 = load i32, ptr %3, align 4
  ret i32 %7
}
```

Prosirenje 2 - Empty Block Elimination

- Nakon core prolaza optimizacije koja uklanja sve dead instrukcije, moze se desiti da neki blokovi ostanu prazni - imali su sve instrukcije bez uticaja na rad programa
- Ovakvi blokovi ne uticu na semantiku programa
- Ovo prosirenje uklanja upravo takve blokove iz finalnog, output.ll fajla

Implementacija

1. Prolazimo po svim blokovima nakon prve faze optimizacije koja je obrisala dead instrukcije
2. Preskacemo EntryBlock jer njega ne smemo brisati
3. Ukoliko block ima samo jednu instrukciju i ona je unconditional BranchInstr, Brisemo taj block tako sto dohvatamo njegovo successor-a, zamenjujemo sva pojavljivanja tekuceg bloka njegovim successor-om
4. Na samom kraju, ubacujemo taj block u listu EmptyBlockova i ukoliko je ona neprazna, prolazimo i brisemo ceo njen sadrzaj

Prosirenje 3 - nastavak

- Rezultat prethodnog prosirenja je zaista brisao sve prazne blockove koji su nakon prvog prolaza izgubili sve svoje dead instrukcije.
- Medjutim, prethodna implementacija je, zbog ovog koda:
  BB.replaceAllUsesWith(Succ); (BB block koji se brise)
  Ostavljala u finalnom .ll fajlu conditional branch instrukcije kod kojih su True i False grane vodile na isti blok
- Takve instrukcije su bile besmislene, jer ne postoji uslovno grananje — oba puta vode na isto mesto.
- Pomocu IRBuilder klase umesto redundantnog conditional brancha kreira se novi jednostavan, unconditional branch i brise stari.

Primer - izvorni kod

```c
int main() {
    int x = 5;
    int y = 10;

    if (x < y) {
        int a = 1;
        int b = a + 2;
        b = b + 3;  // ni 'a' ni 'b' se nikad ne koriste van ovog bloka
    }

    return x;
}
```

Primer - neoptimizovani ll fajl

```
define dso_local noundef i32 @main() #4 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 5, ptr %2, align 4
  store i32 10, ptr %3, align 4
  %6 = load i32, ptr %2, align 4
  %7 = load i32, ptr %3, align 4
  %8 = icmp slt i32 %6, %7
  br i1 %8, label %9, label %14
```

```
9:
  store i32 1, ptr %4, align 4
  %10 = load i32, ptr %4, align 4
  %11 = add nsw i32 %10, 2
  store i32 %11, ptr %5, align 4
  %12 = load i32, ptr %5, align 4
  %13 = add nsw i32 %12, 3
  store i32 %13, ptr %5, align 4
  br label %14

14:
  %15 = load i32, ptr %2, align 4
  ret i32 %15
```

Primer - optimizovani fajl

```
define dso_local noundef i32 @main() #4 {
  %1 = alloca i32, align 4
  store i32 5, ptr %1, align 4
  br label %2

2:                                                ; preds = %0
  %3 = load i32, ptr %1, align 4
  ret i32 %3
}
```