Konstrukcija kompilatora

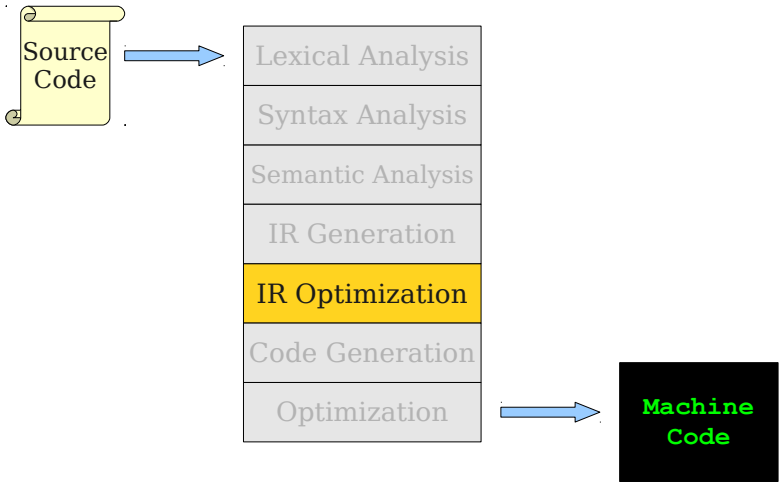
— Optimizacije —

Milena Vujošević Janičić

Matematički fakultet, Univerzitet u Beogradu

Sadržaj

Where We Are



Šta optimizujemo

- Optimizatori mogu da pokušaju da poboljšaju kod u skladu sa različitim željenim osobinama.
- Šta je to što možemo da želimo da optimizujemo?
- **Vreme izvršavanja** (napraviti program da bude što brži, na račun memorije i energije)
- **Upotreba memorije** (napraviti što je manji program, na račun vremena izvršavanja i energije)
- **Upotreba energije** (napraviti program da troši što manje energije, biranjem jednostavnih instrukcija, na račun brzine izvršavanja i memorije)
- Postoje i razni drugi mogući zahtevi — minimizovati pozive funkcija, redukovati korišćenje jedinice za rad u pokretnom zarezu...

Vrste optimizacija

- Postoje tri vrste optimizacija: lokalna, globalna i međuproceduralna
- Lokalna optimizacija radi u okviru jednog osnovnog bloka
- Globalana optimizacija radi na grafu kontrole toka funkcije
- Međuproceduralna optimizacija radi na celokupnom grafu kontrole toka koji prati pozive funkcija i njihovu međusobnu interakciju
- Međuproceduralna optimizacija se nekada naziva i Whole program optimization (WPO)

Međuproceduralna optimizacija

- Različiti programski jezici se različito kompiliraju
- Na primer, C/C++ i Fortran se kompiliraju po jedinicama prevođenja (translation units) koje odgovaraju datotekama koje dobijaju iz izvornih fajlova nakon preprocesiranja
- Ukoliko je na raspolaganju samo jedinica prevođenja, onda na osnovu nje ne možemo da dobije informacije potrebne za WPO, jer jedna jedinica prevođenja ne sadrži informacije za ceo program
- Za takve programske jezike, WPO optimizacija se radi u fazi linkovanja, jer su tada na raspolaganju sve potrebne informacije
- link-time optimization (LTO)

1 Primeri međuproceduralnih optimizacija

Primeri međuproceduralnih optimizacija

- Umetanje koda
- Izdvajanje koda
- Raspoređivanje funkcija
- Eliminacija mrtvog koda
- ...

1.1 Umetanje koda

Umetanje koda

- Umetanje koda — inlining
- Duplikacija koda sa ciljem dobijanja na efikasnosti

```
int s = 0;
for(int i=0; i<1000000;i++) {
   s += f(i);
}
...
int f(int i) {
   return i*i;
}
```

```
int s = 0;
for(int i=0; i<1000000;i++) {
   s += i*i;
}
```

Umetanje koda

- Prednosti:
  - Brže izvršavanje programa
  - Otvaranje prostora za nove optimizacije
- Mane:
  - Veći kod kada se telo iste funkcije umetne na više mesta
- Kako na osnovu analize koda zaključiti koje funkcije umetnuti a koje ne?
  - Umetanje „pravih” funkcija povećava efikasnost
  - Umetanje „pogrešnih” funkcija povećava veličinu izvršive datoteke a ne doprinosi efikasnosti

1.2 Izdvajanje koda

Izdvajanje koda — *outlining*

```
int global;
...
int a, b, c, d;
...
int s1 = a + b;
s1--;
global++;
f(s1, global);

int s2 = c + d;
s2--;
global++;
f(s2, global);
```

```
int global;
...
int a, b, c, d;
...
int s1 = outline(a, b);
int s2 = outline(c, d);

int outline(int i1, int i2) {
  int s = i1 + i2;
  s--;
  global++;
  f(s, global);
  return s;
}
```

Izdvajanje koda

- Smanjuje količinu memorijskog prostora koji zauzima program, ali pritom potencijalno povećava njegovo vreme izvršavanja.
- Pronalazi segmente koda (uzastopne nizove instrukcija) koji se ponavljaju u programu, izdvaja ih u zasebnu funkciju i menja pojavljivanja tog segmenta sa pozivom ka novoj funkciji.
- Ova optimizacija je posebno korisna za uređaje sa malom količinom memorije, najčešće uređajima sa ugrađenim računarom poput pametnih satova, mp3 plejera ili urađaja zasnovanih na razvojnom sistemu Arduino.
- Vreme izvršavanja programa može da bude oštećeno ako se izdvoji deo koda koji se baš često izvršava.
- Pogledajte Master rad Vladimir Vuksanović

1.3 Raspoređivanje funkcija

Raspoređivanje funkcija

- Raspoređivanje funkcija — code positioning
- Ako funkcija A poziva funkciju B puno puta, dobro je da one u memoriji budu raspoređene jedna pored druge
- Operativni sistem učitava memoriju po stranicama, a broj stranica u fizičkoj memoriji je ograničen - ako A poziva B puno puta, a one su locirane na različitim stranicama to može da utiče loše na vreme izvršavanja
- Neka funkcija A poziva funkciju B koja poziva funkcije C, D. Memorija je linarna, da li treba poređati funkcije ABCD ili ABDC ili ACBD?
- Graf poziva funkcija u programu (engl. *call graph*) može da bude veoma kompleksan — kome dati prednost?
- Primer algoritma: Profile guided code positioning

2 Optimizacije vođene profilima

Optimizacije vođene profilima

- Profili su informacije o ponašanju programa na osnovu kojih možemo da donesemo kvalitetne odluke
  - gde i kako da umećemo funkcije da bismo dobili najviše na efikasnosti,
  - gde da uradimo izdvajanje koda a da ne izgubimo na efikasnosti,
  - kako da rasporedimo funkcije
- Profili sadrže i razne druge informacije koje mogu da budu korisne za druge optimizacije

JIT vs AOT kompilacija

Jedna od prednosti koju JIT kompajleri imaju u odnosu na AOT kompajlere je sposobnost analize ponašanja aplikacije u vreme izvršavanja. Na primer, HotSpot prati koliko puta je svaka grana `if` iskaza izvršena. Ove informacije se prenose svom JIT kompajleru (npr. *Graal*) kao informacije koje nazivamo profilom. Profil je sažetak načina na koji je određeni metod izvršen tokom vremena izvršavanja. JIT kompajler zatim pretpostavlja da će se metod i dalje ponašati na isti način i koristi informacije iz profila da bi taj metod bolje optimizovao.

JIT vs AOT kompilacija

AOT kompajleri tipično nemaju informacije o profilisanju i obično su ograničeni na statički prikaz koda koji kompajliraju. To znači da, osim heuristika, AOT kompajler vidi svaku granu svakog `if` iskaza kao podjednako verovatnu za izvršavanje, svaki metod je podjednako verovatno da će biti pozvan kao i bilo koji drugi, i svaka petlja će se ponoviti isti broj puta. Ovo stavlja AOT kompajler u nepovoljan položaj — bez informacija o profilu, teško je generisati mašinski kod istog kvaliteta kao JIT kompajler.

Šta sadrži profil?

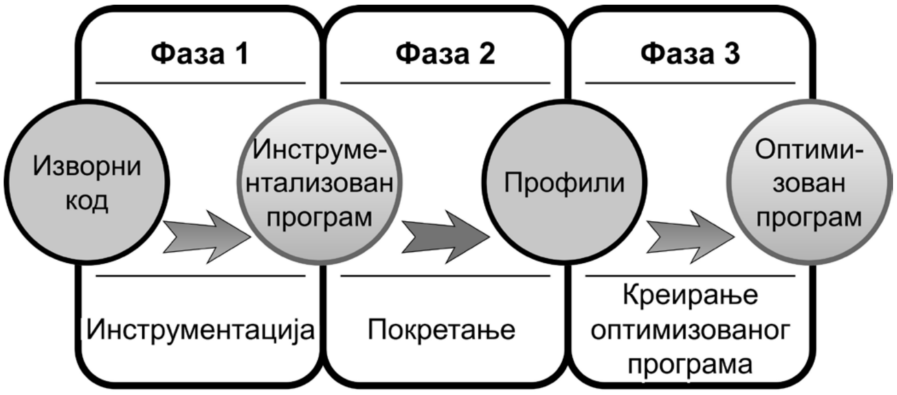
U praksi, profil je sažeti zapis o tome koliko puta su se određeni događaji desili tokom vremena izvršavanja. Ovi događaji su izabrani na osnovu toga koje informacije će biti korisne za kompajler kako bi doneo bolje odluke. Primeri takvih događaja su:

- Koliko puta je svaki metod pozvan?
- Koliko puta je u `if` iskazu vrednost provere evaluirana kao tačna, a koliko puta kao netačna?
- Kod virtuelnih poziva, koliko puta je pozvan koji metod?
- Koliko puta je jedan metod pozvan u nekom konkretnom kontekstu?
- Koliko puta se desilo neko zaključavanje?
- ...

Kako nabaviti profile u JIT kompilaciji?

Kada se aplikacija pokreće na JVM-u, profilisanje aplikacije obavlja okruženje za vreme izvršavanja, bez dodatnih koraka od strane programera. Iako je ovo nesumnjivo jednostavnije, profilisanje koje obavlja okruženje za vreme izvršavanja nije besplatno — ono uvodi dodatne troškove u izvršavanju koda koji se profilira — kako u smislu vremena izvršavanja, tako i u smislu korišćenja memorije. Ovo uzrokuje probleme sa zagrevanjem — aplikacija će dostići predvidljive vrhunske performanse tek nakon što prođe dovoljno vremena da ključni delovi aplikacije budu profilisani i JIT kompajlirani. Za dugotrajne aplikacije, ovaj trošak se obično isplati, donoseći kasnije poboljšanje performansi. S druge strane, za kratkotrajne aplikacije i aplikacije koje moraju da počnu sa dobrim, predvidljivim performansama što je pre moguće, ovaj trošak je kontraproduktivan.

Kako nabaviti profile u AOT kompilaciji?



Kako nabaviti profile u AOT kompilaciji?

Prikupljanje profila za AOT-kompajliranu aplikaciju je složenije i zahteva dodatne korake od strane programera, ali ne uvodi dodatne troškove u konačnu aplikaciju. Ovde se profili moraju prikupljati posmatranjem aplikacije dok je u radu. Ovo se obično radi kompajliranjem aplikacije u posebnom režimu koji ubacuje instrumentacioni kod u samu aplikaciju. Instrumentacioni kod povećava brojače za događaje koji su od interesa za profil. Ovaj program nazivamo *instrumentalizovanom* aplikacijom, a proces dodavanja potrebnih brojača naziva se *instrumentacija*.

Kako nabaviti profile u AOT kompilaciji?

Prirodno, instrumentalizovana aplikacija neće biti tako performantana kao podrazumevana verzija zbog troškova instrumentacionog koda, pa se zato ona i ne pokreće redovno u produkciji. Međutim, izvršavanje sintetičkih reprezentativnih opterećenja na instrumentalizovanoj verziji omogućava nam da prikupimo reprezentativni profil aplikacije (baš kao što bi to okruženje za vreme izvršavanja uradilo za JIT kompajler). Profil aplikacije se skladišti kao eksterna datoteka sa informacijama koja je onda na raspolaganju AOT kompajleru. Kada se gradi optimizovani binarni fajl, AOT kompajler ima i statički prikaz i dinamički

profil aplikacije. Binarni fajl optimizovan korišćenjem profila ima značajno bolje performanse AOT-kompajliranog binarnog fajla koji je prevedn bez profila.

Optimizacije vođene profilima

- Informacije koje se dobijaju iz kvalitetnih profila značajno utiču na efikasnost i veličinu izvršive datoteke
- S druge strane proces dobijanja profila je zahtevan. Dodatno, pre samog prikupljanja profila potrebno je i definisati ulaze za koje će se profili prikupljati kako bi profili oslikavali realnu upotrebu programa. Pogrešni ulazi daju loše profile koje onda ne utiču povoljno na performanse.

Optimizacije vođene profilima

- Umesto prikupljanja profila može se koristiti i zaključivanje profila heuristikama ili metodama mašinskog učenja

- Pogledajte naredni tekst o statičkom i dinamičkom prikupljanju profila

Zaključak

- Međuproceduralne optimizacije su najkompleksnije optimizacije koje zahtevaju netrivijalnu analizu koda programa ili sakupljene informacije o ponašanju programa
- Najbolji rezultati se postižu ukoliko se prilikom optimizacija koriste profili izvršavanja programa
- Prikupljanje profila nije jednostavno tako da se ne koristi uvek
- Međuproceduralne optimizacije su veoma aktivna oblast istraživanja, pogledajte, na primer: GraalSP: Polyglot, efficient, and robust machine learning-based static profiler

3 Literatura

Literatura

(svaki naslov je link)

- Boris Spasojević, Profile Guided Optimizations for Native Image
- Vladimir Vukasović, Master rad

- Milan Čugurović, Dinamičko prikupljanje i statičko predviđanje profila izvršavanja u kontekstu kompajlerskih optimizacija
- Milan Čugurović, Milena Vujošević Janičić, Vojin Jovanović, Thomas Würthinger, GraalSP: Polyglot, efficient, and robust machine learning-based static profiler, Journal of Systems and Software, Volume 213, 2024.