Optimizacije u okviru kompajlera GCC/LLVM

Seminarski rad u okviru kursa
Metodologija stručnog i naučnog rada
Matematički fakultet

Tamara Stojković, Emilija Stošić, Teodora Isailović
tamara.stojkovic.1998@gmail.com, emilijazstosic@gmail.com,
teodora.isailovic@gmail.com

12. novembar 2022.

Sažetak

Ovo se piše na kraju.

Sadržaj

1 Uvod

Optimizacija predstavlja tehniku transformacije dela programa, tako da kod bude što je moguće efikasniji. Za cilj ima poboljšanje performansi koda, a ne savršen rezultat (uglavnom se ne može se reći da predstavlja pronalazak „optimalnog rešenja“). Optimizacija je oblast u kojoj se danas vrši većina istraživanja kompajlera. U okviru savremenih kompajlera dostupan je veliki broj optimizacija koje sve međusobno deluju na razne načine i imaju uticaj na kvalitet koda, veličinu koda, vreme kompilacije, potrošnju energije itd. Kompajleri obično obezbeđuju ograničen broj standardnih optimizacionih nivoa, kao što su O1, -O2,-O3 i -Os. Na ovim nivoima ostvaruju se kompromisi između različitih mera kao što su kvalitet koda, veličina koda i vreme kompilacije. U ovom radu ćemo upravo predstaviti vrste optimizacija, sa akcentom na one koje su dostupne u okviru kompajlera GCC i LLVM.

2 Osnovna podela optimizacija

Postoje razne tehnike u primeni procesa optimizacije. Razlikujemo optimizacije koje se primenjuju na međukod, kao deo generisanja ciljnog koda, ali i nakon generisanja ciljnog koda. **Optimizacija međukoda** primenjuje se da se kod pojednostavi, preuredi ili sažme. Predstavlja optimizaciju koja ne uzima u obzir specifičnost ciljne arhitekture. Neke od stavki koje **optimizacija kao deo generisanja ciljnog koda** omogućava su biranje instrukcija i određivanje načina alociranja objekata. Na kraju, može se izvršiti i **optimizacija ciljnog koda**, gde se pokušava prerada samog asemblerskog koda u nešto efikasnije. U ovom slučaju zahteva se detaljno poznavanje ciljne arhitekture, kao i asemblerskog i mašinskog jezika ciljnog programa.

2.1 Optimizacije međukoda

U okviru optimizacije međukoda razlikujemo :**lokalne**, **globalne** i **međuproceduralne optimizacije**.

2.1.1 Lokalne optimizacije

Lokalne optimizacije služe za ubrzavanje malih delova neke funkcije i rade sa konketnim naredbama unutar osnovnog bloka(eng. basic block) One su obično najlakše za izvođenje jer nije neophodno raditi analizu kontrole toka(eng. *Control - flow*). U nastavku navodimo neke od tehnika lokalne optimizacije koje se vrše nad međukodom.

- **Eliminacija čestih podizraza (eng. *Common subexpression elimination*)**
  Podrazumeva izbegavanje izvračunavanja čestog izraza više puta(ako se javlja više od jednom). U slučaju da dve operacije daju isti rezultat, kažemo da su česte, pa je bolje izračunati izraz jednom i pozvati ga naredni put na mestu gde je neophodno.
- **Slaganje konstanti (eng. *Constant folding*)**
  Podrazumeva da se konstantni izrazi mogu evaluirati u vreme kompilacije. To uključuje utvrđivanje da svi operandi u izrazu imaju konstantnu vrednost, a zatim menjanje izraza njegovom vrednošću. Često je i lepše za pisanje i čitanje.

- **Propagacija kopija (eng. *Copy propagation*)**
  Podrazumeva izbegavanje uvođenja promenljivih, koje samo čuvaju vrednosti nekih promenljivih koje već postoje. Posebno je značajna jer je u stanju da eliminiše veliki broj instrukcija, koje služe samo za kopiranje vrednosti iz jedne promenljive u drugu.
- **Smanjenje snage operatora (eng. *Operator strength reduction*)**
  Podrazumeva zamenu operatora „jeftinijim operatorom". Od same arhitekture zavisi koje operacije se smatraju najjeftinijim, a koje najskupljim, tako da je za dobru optimizaciju neophodno poznavati ciljnu mašinu.
- **Eliminacija mrtvog koda (eng. *Dead code elimination*)**
  Podrazumeva da ako se rezultat neke instrukcije nadalje ne koristi, instrukcija se smatra „mrtvom" i može biti uklonjena. Ovo omogućava uprošćavanje koda, izbacivanjem nepotrebnih izračunavanja.
- **Algebarsko pojednostavljenje i reasocijacija (eng. *Algebaric simplification and reassociation*)**
  Podrazumeva pojednostavljenje izraza korišćenjem algebarskih svojstava(zakona algebre). Reasocijacija podrazumeva korišćenje svojstava kao što su asocijativnost, komutativnost i distributivnost kako bi se nakon toga omogućila lakša primena drugih optimizacija.
- **Kompozicija lokalnih transformacija**
  Različite optimizacije koje smo do sada videli, brinu o malom delu koda. Za maksimalan efekat, možda će morati neke optimizacije da se primene više puta.

2.1.2 Globalne optimizacije

Predstavljaju optimizacije koje su slične lokalnim i koje možemo primeniti sa nekim dodatnim analizama. „Globalno" u ovom slučaju ne znači u celom programu, već se optimizacije primenjuju na jednu po jednu funkciju. Ova analiza je znatno moćnija od lokalne, ali dosta komplikovanija. Mnoge lokalne optimizacije se mogu se primenljivati i globalno, dok postoje i one koje se mogu primeniti samo na globalnom nivou. Neke globalne optimizacije date su u nastavku.

- **Globalna eliminacija mrtvog koda**
  U ovom slučaju jedina razlika je što se informacije o tome gde je neka promenljiva živa, moraju dobiti globalnom analizom, što je značajno komplikuje u odnosu na lokalnu eliminaciju mrtvog koda.
- **Globalno propagiranje konstante**
  Ovo je optimizacija koja svaku promenljivu za koju se zna da je konstantna menja sa tom konstantom. Potrebno je pratiti vrednosti koje mogu biti dodeljene promenljivoj u svakoj tački programa.
- **Globalna eliminacija čestih podizraza**
  Česti podizrazi mogu se eliminisati i na globalnom nivou, poznavanjem skupa dostupnih izraza. Izrazi su dostupni u jednom trenutku ako su živi pri ulasku u blok, što se utvrđuje ospežnom analizom.
- **Optimizacija kretanje koda (eng. *Code motion*)**
  Postoje optimizacije koje se mogu primeniti samo globalno i jedna takva je ova i ona objedinjuje sekvence koda zajedničke za jedan ili više blokova. Cilj je smanjiti veličinu koda i eliminisati eventualna

skupa ponovna izračunavanja. Dva oblika optimizacije izdizanjem koda koje ćemo izdvojiti su :

- **Pomeranje invarijantnog koda(eng. *loop-invariant*)**
  Podrazumeva da se računanje vrednosti neke promenljive može izdvojiti izvan petlje, ako je nezavisno od brojača petlje.
- **Parcijalna eliminacija suvišnosti (eng. *Partial redundancy elimination*)**
  Za neki račun u programu kaže se da je suvišan, ako računa već poznatu vrednost. Parcijalno redundantno izračunavanje je ono čija je vrednost poznata samo u nekim delovima.

2.1.3 Međuproceduralne optimizacije

U okviru ove optimizacije jedna od najznacajnijih tehnika je **uvlačenje definicija funkcija(eng. *Inlining*)**. Uglavnom se jednostavnije funkcije uvlače, a složenije ne. Premalo uvlačenja dovodi do troškova vezanih za pozive funkcija, a previše uveća veličinu koda i dovodi do neefikasnosti, pa treba uspostaviti ravnotežu.

2.2 Optimizacije koda

Optimizovani međukod se u fazi generisanja koda prevodi u asemblerski tj. mašinski kod. U finalnom generisanju koda, bitno je donošenje pametnih odluka kako bi generisani ciljni kod bio što efikasniji. U ovom slučaju bitne su specifične karakteristike mašine za prozivodnju optimizovanog koda za tu konkretnu arhitekturu. Ralikujemo sledeće tehnike optimizacija koda:

- **Optimizacija redosleda instrukcija**
  Razlikujemo fazu odabira instrukcija, fazu alokacije registara i fazu raspoređivanja instrukcija.
- **Optimizacija upotrebom keša**
  Upotreba keša se zasniva na prostornoj i vremenskoj lokalnosti, a cilj je da postane što bolja. Ako je nekoj memoriji skoro pristupano, vremenska podrazumeva da ce biti uskoro opet,a prostorna da ce i njeni susedni objekti biti uskoro korišćeni.

Postoje razne optimizacije koje se mogu sprovesti i na nivou izgenerisanog ciljnog koda.

3 Prvi naslov

Ovde pišem tekst.

3.1 Prvi podnaslov

Ovde pišem tekst.

4 Različite optimizacije u okviru kompajlera
GCC/LLVM

U odnosu na namenu svakog od ovih kompajlera i opseg programa za koje su specifikovani, određuje se i koji je bolji za optimizaciju u nekim konkretnim slučajevima. Karakteristike kompajlera koje određuju njihovu oblast upotrebe pa samim tim i optimizacije su:

- **GCC** je oficijalni kompajler za GNU i Linux sisteme i glavni kompajler za kompajliranje i kreiranje drugih operativnih sistema. Podržava tradicionaldne jezike kao što su Ada, Fortran i GO. GCC podržava manje popularne arhitekture, kao i RISC-V ranije u odnosu na LLVM. Trenutno je brži od LLVM-a u kompajliranju LINUX kernela. U slučaju korišćenja LLVM-a , kernel se ne može kompajlirati bez modifikacije izvornog koda i parametara kompilacije.
- Novi jezici kao što su Swift, Rust, Julia i Ruby, koriste **LLVM**. LLVM je usklađen strožije sa standardima C-a i C++-a nego GCC, pa se problemi manje javljaju. On takođe podržava neke ekstenzije, kao što su atributi za proveru bezbednosti niti. Ovaj kompilator omogućava tačnije i preciznije dijagnostičke informacije i poruke o greškama. U GCC-u su počele da se poboljšavaju od GCC-a 8. Clang obezbeđuje dodatne korisne alate za statičku analizu i skeniranje koda(scan-build, clang static analyzer, clang-format, clang-tidy). Clang je kompajler za C, C++, Objective-C, ili Objective-C++.

GCC se smatrao superiornijim, ali LLVM napreduje. Sada neki smatraju da se LLVM uglavnom koristi da obezbedi performanse superiornije od GCC-a.
Optimizacije su do sada postale jedna od osnovnih komponenti kompajlera. U kompajlerima je implementirano stotine optimizacija. Na primer, postoji više od 200 optimizacija za GCC i više od 100 za LLVM. Rezultati optimizacije u okviru ovih kompajlera najčešće se moraju posmatrati na konkretnim slučajevima, pa su neki primeri dati u nastavku.

4.1 Prvi primer testiranja razlika

Većina radnih opterećenja u oblaku (eng. *cloud workloads*) mogu da rade u različitim klasterima i nije potrebno navoditi parametre koji se odnose na mašinu. Samo radno opterećenje ima nizak nivo kompilacije i optimizacije (-O2 ili ispod). Posmatranjem razlika za GCC i LLVM na nivoima -O2 i -O3 za INT Speed programe (za testiranje brzine) utvrđeno je da GCC ima prednost od 1% do 4% u odnosu na LLVM u slučaju većine programa na nivoima -O2 i -O3. Stavke koje su uzete za test nemaju specifične žarišne tačke, pa se mogu smatrati sveobuhvatnim.

Iz ovog testiranja se mogu izvesti sledeća zapažanja:

-Rezultati testiranja pokazuju da je GCC uvek povoljan u pogledu optimizacije performansi. Međutim u 2 specifična programa u vezi sa veštačkom inteligencijom LLVM poboljšava performanse za više od 3% u odnosu na GCC.
-LLVM optimizuje vektore na nivou -O2, a GCC na -O3 nivou.
-Osim vektorizovanih programa, GCC ne poboljšava performanse na nivou -O3, u poređenju sa onim na O2, dok LLVM značajno poboljšava

performanse nekih programa na nivou O3.
-U nekim slučajevima(na nivoima -O3 i više) GCC može poboljšati performanse , dok Clang ima problem sa tendencijom da previše odmotava petlje, što može dovesti do problema (ovde je rešenje koristiti optimizovanje za veličinu(-Os) da biste ograničili odmotavanje petlji).

LLVM međukod je koncizniji i zauzima manje memorije tokom kompilacije od GCC međukoda Na osnovu urađenih testova, zaključeno je da Clang nudi više prednosti u procesu izgradnje velikih projekata od GCC-a. GCC je skoro uvek povoljan u pogledu performansi, ali ipak zavisi od specifične aplikacije.

4.2 Drugi primer testiranja razlika

U većini nabrojanih vrsta optimizacija u sekciji 2 kao što su uvlačenje definicija funkcija, slaganje konstanti, propagacija konstanti, na testiranom konkretnom primeru oba kompajlera pokazala su se dobro.

Neke razlike koje su primećene su:
Kod odmotavanja petlji niza(eng. *loop unroling, array loops*) LLVM ne funkcioniše ispravno i komplikuje, dok GCC dobro radi.
U slučaju indukcionih promenljivih sa float izrazima ili sa integer izrazima (koji nisu nizovi elemenata) GCC ne može da radi, a LLVM radi. Takođe važi i u slučaju više akumulatora (integer ili float).
U ovom slučaju GCC nije radio ni za algebarsku redukciju primenom asocijativnosti, dok LLVM jeste.

5 n-ti naslov

Ovde pišem tekst.

5.1 ... podnaslov

Ovde pišem tekst.

6 Zaključak

Ovde pišem zaključak.

Literatura

A Dodatak

Ovde pišem dodatne stvari, ukoliko za time ima potrebe.