# Optimizacije kroz GCC, LLVM i Native Image

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**

Optimizacije su važan deo u procesu poboljšanja performansi programa. Kompajleri imaju različite nivoe optimizacije. Koji kompajleri i koji nivoi optimizacije će se koristiti zavisi od programa koji se kompajlira. Važno je razumeti sličnosti i razlike između optimizacija koje su dostupne u okviru savremenih kompajlera. U ovom radu ćemo predstaviti vrste optimizacija, sa akcentom na one koje su dostupne u okviru kompajlera GCC, LLVM i Native Image.

## Sadržaj

**1 Uvod** **2**

**2 Osnovna podela optimizacija** **2**
2.1 Optimizacije međukoda . . . . . 2
2.1.1 Lokalne optimizacije . . . . . 2
2.1.2 Globalne optimizacije . . . . . 3
2.1.3 Međuproceduralne optimizacije . . . . . 4
2.2 Optimizacije koda . . . . . 4

**3 Napredne optimizacije u okviru kompajlera GCC/LLVM** **5**
3.1 Optimizacije u okviru kompajlera GCC . . . . . 5
3.2 Optimizacije u okviru kompajlera LLVM . . . . . 6

**4 Različite optimizacije u okviru kompajlera GCC/LLVM** **8**
4.1 Prvi primer testiranja razlika . . . . . 8
4.2 Drugi primer testiranja razlika . . . . . 9
4.3 Razlike u nivoima optimizacije . . . . . 9

**5 Native-Image kompajler** **10**
5.1 Problemi i optimizacije . . . . . 10
5.1.1 Dinamičko učitavanje i refleksija . . . . . 10
5.1.2 Java Native Interface . . . . . 11

**6 Zaključak** **11**

**Literatura** **11**

# 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 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. [1]

# 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. [14]

## 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. 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 izračunavanja 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. *Algebraic 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. [10]

### 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 mogu se primenjivati 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 se 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 opsežnom 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. [4]

### 2.1.3 Međuproceduralne optimizacije

Međuproceduralna optimizacija radi na celokupnom grafu kontrole toka (eng. *Control Flow Graph, CFG*) [1] koji prati pozive funkcija i njihovu međusobnu interakciju. [11] Ova optimizacija se vrši na nivou celog programa tj. više funkcija. U okviru ove optimizacije jedna od najznačajnijih 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. [12]

## 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. Razlikujemo sledeće tehnike optimizacija koda:

- **Optimizacija redosleda instrukcija**
Razlikujemo fazu odabira instrukcija, fazu alokacije registara i fazu raspoređivanja instrukcija. Ove 3 faze su veoma isprepletane, ali obično se najpre radi odabir instrukcija, dok se treća faza nekada vrši pre, a nekada posle druge faze.
Sekvenca instrukcija se obično može na više načina prevesti u asemblerski kod i instrukcija se može predstaviti drvetom. Nakon toga se svakoj asemblerskoj instrukciji pridružuju oblici drveta koji se mogu prekriti sa njom. **Odabir instrukcija** se vrši tako da se na osnovu tih šablona pokušava prekriti celo drvo. **Alokacija registara** je proces dodeljivanja promenljivih registrima, kao i upravljanje prenosom podataka u registrima i van njih. Cilj je da što više promenljivih bude smešteno u registre procesora. Jedna česta tehnika za alokaciju registara je bojenje grafa. **Raspoređivanje instrukcija** podrazumeva pravljenje rasopreda instrukcija, čiji je cilj poboljšanje performansi.
- **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 će biti uskoro opet, a prostorna da će i njeni susedni objekti biti uskoro korišćeni.

Postoje razne druge optimizacije koje se mogu sprovesti i na nivou izgenerisanog ciljnog koda. [11]

[1] Graf kontrole toka (eng. *Control Flow Graph, CFG*) je graf koji sadrži osnovne blokove funkcije.

# 3 Napredne optimizacije u okviru kompajlera GCC/LLVM

## 3.1 Optimizacije u okviru kompajlera GCC

GCC je kompajler programskog jezika koji je razvio GNU, to je zvanični kompajler za GNU i Linux sisteme. Ričard Stolman (eng. *Richard Stallman*) je 1984. godine pokrenuo GNU projekat čiji je cilj bio izgradnja softverskog sistema otvorenog koda sličnog UNIKS-u. GNU operativni sistem se nije razvijao kako se očekivalo. Međutim, razvili su se mnogi odlični i korisni softverski alati otvorenog koda, kao što su Make, Emacs, GDB i GCC. [9]

GCC prevodi program, ukoliko nema sintaksnih grešaka, kreiraće se izvršni fajl, u suprotnom će prevodilac ispisati poruke o greškama i broj linija u kodu gde se te greške nalaze. Takođe, GCC ima veliki broj opcija kojima se mogu podesiti parametri prevođenja. Opcijom -o se zadaje ime izvršnog programa, opcije -Wall, -Wextra se koriste za generisanje poruka o upozorenjima, kada postoje konstrukcije koje su sintaksno ispravne, ali mogu da budu semantički neispravne. Opcije koje su vrlo značajne u procesu kompilacije, su opcije za optimizaciju. Kao i kod opcija za upozorenja i kod opcija za optimizaciju moguće je precizno definisati koje se sve optimizacije vrše ili se može zadati neka od grupnih opcija koje podržavaju veći broj optimizacija. Nivo optimizacije koju kompajler vrši se kontroliše opcijom -On, gde je n nivo zahtevane optimizacije. Promena ove vrednosti će zahtevati više vremena za kompilaciju koda i koristiće mnogo više memorije, posebno kako se povećava nivo optimizacije. [13] Neke optimizacije smanjuju veličinu rezultujućeg koda, dok druge pokušavaju da kreiraju brži kod, potencijalno povećavajući njegovu veličinu. Postoji osam -O podesavanja: **-O0, -O1, -O2, -O3, -Os, -Oz, -Og, -Ofast**.

- **O0 nivo optimizacije:** Na ovom nivou optimizacija je potpuno isključena. Ako se ne navede -O opcija, ovo je podrazumevani nivo optimizacije. Značajno smanjuje vreme kompilacije i poboljšavaju se informacije o otklanjanju grešaka. Neki programi ne rade ispravno bez nekog nivoa optimizacije, pa se ovaj nivo ne preporučuje.
- **O1 nivo optimizacije:** Optimizacije koje se vrše imaju dva cilja, koji nekad mogu da budu suprotstavljeni, smanjenje veličine koda koji se kompajlira, uz povećanje njegovih performansi. Optimizacije obično ne zahtevaju značajnu količinu vremena da bi se izvršile.
- **O2 nivo optimizacije:** Preporučeni nivo optimizacije. Kompajler pokušava da poveća performanse koda bez ugrožavanja veličine samog koda i bez uzimanja previše vremena za kompilaciju.
- **O3 nivo optimizacije:** Najviši nivo optimizacije. Naglasak je na brzini kompajliranja, a ne na veličini generisanog koda. Omogućena je i optimizacija inline-funkcija, što povećava performanse, ali može da poveća veličinu samog koda, što zavisi od veličine funkcija koje su umetnute. Kompajliranje na ovom nivou ne garantuje povećanje performansi, zapravo, često sistem može da se uspori zbog većih binarnih datoteka i povećane upotrebe memorije. Nije preporučeno koristiti ovaj nivo optimizacije
- **Os nivo optimizacije:** Vrše se sve optimizacije kao i na nivou -O2, koje ne povećavaju veličinu koda, kao što su optimizacije poravnanja. Naglasak je stavljen na veličinu, a ne na brzinu generisanog koda.

- **Og nivo optimizacije:** Nivo optimizacije je uveden u 4.8 standardu. Ciljevi optimizacije su brzina kompajliranja i otklanjanje grešaka, dok se istovremeno pruža određeni nivo performansi.
- **Ofast nivo optimizacije:** Nivo je uveden u 4.7 standardu. Vrše se sve optimizacije kao i na nivou -O3 plus **-ffast-math, -fno-protect-parens i -fstack-arrais**. [7]

Program 1 se koristi za demonstraciju efekata različitih nivoa optimizacije:

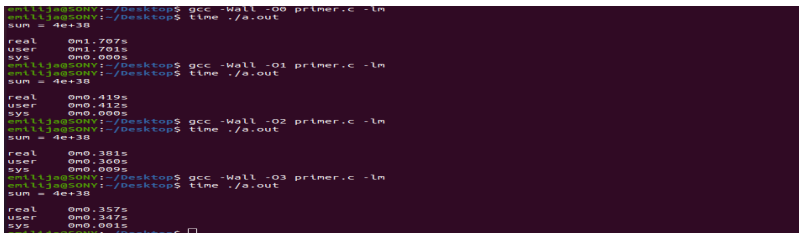
```c
#include <stdio.h>

double powern (double d, unsigned n)
{
  double x = 1.0;
  unsigned j;
  for (j = 1; j <= n; j++)
    x *= d;
  return x;
}

int main (void)
{
  double sum = 0.0;
  unsigned i;
  for (i = 1; i <= 100000000; i++) {
      sum += powern (i, i % 5);
  }
  printf ("sum = %g\n", sum);
  return 0;
}
```

Listing 1: Program za izračunavanje n-tog stepena brojeva

Glavni program sadrži petlju koja poziva funkciju powern, koja računa n-ti stepen broja. Vreme izvršavanja programa se može meriti naredbom time. Sa slike 1 se jasno vidi da se vreme izvršavanja programa smanjuje, kako se povećava nivo optimizacije.




Slika 1: Rezultati optimizacije kod GCC kompajlera

## 3.2 Optimizacije u okviru kompajlera LLVM

LLVM je projakat otvorenog koda koji je razvio Kris Latner (eng. *Chris Lattner*), kao istraživački projekat na Univerzitetu Ilinois. LLVM je zvanični naziv koji je primenljiv na sve projekte pod LLVM, koji zajedno čine potpun kompajler: prednji, srednji i zadnji deo, optimizatore, asemblere, linkere, libc++ i druge komponente. Pomaže u izgradnji novih računarskih jezika i poboljšanju postojećih. Automatizuje teške i neprijatne zadatke koji su uključeni u kreiranje jezika, kao što je prenos izlaznog koda na više platformi i arhitektura, LLVM je zapravo okvir za generisanje objektnog

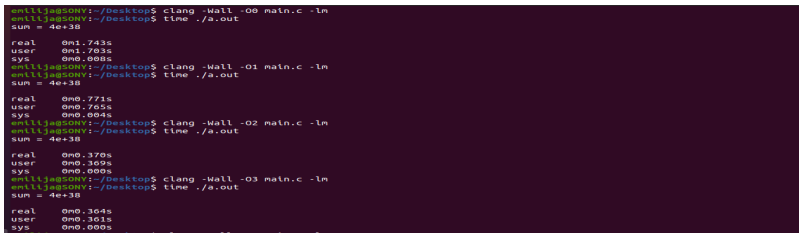
koda iz bilo koje vrste izvornog koda. LLVM ne vrši samo kompajliranje IR u izvorni mašinski kod, takođe može da vrši optimizaciju koda. Optimizacije mogu biti prilično agresivne, uključujući stvari kao što su umetanje funkcija, eliminisanje mrtvog koda i odmotavanje petlji. [15] [9]

Alat opt vrši analizu i optimizaciju koda. Uzima ulazne LLVM datoteke kao ulaz, pokreće optimizacije ili analize i na kraju vraća optimizovanu datoteku ili rezultate analize. Ukoliko je zadata opcija -analyze, opt pokreće različite analize nad ulazom. Ukoliko opcija -analyze nije data, opt vrši optimizacije i pokušava da proizvede optimizovanu izlaznu datoteku. Optimizacije su zadate u vidu prolaza, koji prelaze neki deo programa da bi prikupili informacije ili transformisali program. Alat opt može da pokrene samo odabrane optimizacije, u određenom redosledu, tako što se pri pokretanju alata zadaju imena tih prolaza u željenom redosledu. Svi LLVM prolazi su podklase klase Pass, funkcionalnosti prolaza su implementirane tako što prevazilaze virtuelne metode nasleđene od klase Pass. Analysis Passes se koriste za prikupljanje informacija koje mogu da služe za otklanjanje grešaka ili vizuelizaciju programa. Transform Passes se koriste za optimizaciju koda. Utility Passes služe za dobijanje raznih informacija, koje su bitne za razvoj drugih prolaza. Nakon svakog prolaza IR mora da bude u validnom stanju.

Postoji sedam -O nivoa optimizacije:

- **O0 nivo optimizacije:** Na ovom nivou, optimizacija je potpuno isključena. Ovo je podrazumevani nivo optimizacije i kod se najbrže kompajlira.
- **O1 nivo optimizacije:** Najosnovniji nivo optimizacije.
- **O2 nivo optimizacije:** Omogućena je većina optimizacija i ovo je preporučeni nivo optimizacije.
- **O3 nivo optimizacije:** Dozvoljene sve optimizacije kao i na nivou -O2, sa dodatnim optimizacijama koje mogu generisati veći kod u cilju da se program brže pokrene.
- **O4 nivo optimizacije:** Najviši nivo optimizacije. Optimizacija celog programa se vrši u vreme povezivanja.
- **Os nivo optimizacije:** Podržava sve optimizacije kao i nivo -O2, sa dodatnim optimizacijama za smanjenje veličine koda.
- **Oz nivo optimizacije:** Dozvoljene sve optimizacije kao i na nivou -Os, a samim tim i -O2, sa dodatnim smanjenjem veličine koda. [6] [8] [5]

Program 1 se može iskoristi i za demonstraciju efekta različitih nivoa optimizacije kod LLVM kompajlera. Sa slike 2 se jasno vidi da se vreme izvršavanja programa smanjuje, kako se povećava nivo optimizacije.




Slika 2: Rezultati optimizacije kod LLVM kompajlera

# 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** podržava tradicionalne 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 tendenciju 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 nego GCC. GCC je skoro uvek povoljan u pogledu performansi, ali izbor ipak zavisi od specifične aplikacije. [9]

## 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. [10]

## 4.3 Razlike u nivoima optimizacije

Kompajleri GCC i LLVM imaju po sedam nivoa optimizacije, sa manjim ili većim razlikama. Optimizacije koje se vrše na nivoima -O0 i -O1 su identične. Kod GCC kompajlera nivo -O2 je preporučeni nivo optimizacije, dok je kod LLVM kompajlera ovo umereni nivo optimizacije. -Os nivo optimizacije kod GCC kompajlera aktivira sve -O2 optimizacije koje ne povećavaju veličinu generisanog koda, dok kod LLVM kompajlera ovaj nivo aktivira sve -O2 optimizacije koje smanjuju veličinu koda. -Oz i -O4 nivoi optimizacije postoje samo kod LLVM kompajlera. -O4 nivo je najviši nivo optimizacije kod LLVM kompajlera, dok je kod GCC kompajlera to -O3 nivo. Ovo nisu preporučeni nivoi optimizacije. Nivoi -Og i -Ofast se javljaju samo kod GCC kompajlera.

| | GCC | LLVM |
|---|---|---|
| **Modularnost koda** | Monolitičan | Modularan |
| **Generisani kod** | Efikasan, veliki broj opcija | Efikasan, zbog korišćenja SSA forme na LLVM backend-u |
| **Brzina kompilacije** | Sporiji zbog oslanjanja na nezavisne kompajlere. Nakon IR, asemblerski kod, pa iz njega objektni fajl | Brži zbog integrisanja sopstvenih kompajlera. Objektni fajl direktno iz IR |
| **Brzina izvršavanja dobijenog koda** | GCC u proseku brži 1-4% za većinu programa na -02 i -03 nivou | |

# 5 Native-Image kompajler

Neke od glavnih prednosti softvera pisanog u jezicima koji se prevode na specifičnoj platformi je brzina pokretanja i manja količina neophodnih resursa. Takav softver se izvršava direktno na procesoru, a ne na virtualnoj mašini (apstrakciji procesora).

Native-Image kompajleri predstavljaju tehniku ahead-of-time kompilacije od izvornog koda, preko bytecode-a, do izvršivog koda određene platforme i arhitekture koji se naziva **native image**. [2] Najpoznatiji primeri su *Native Image* iz GraalVM za jezike koji se izvršavaju na Java virtuelnoj mašini (JVM) i *Ngen* za .NET Framework, odnosno *CrossGen* za .NET Core. U nastavku opisan je alat iz GraalVM.

Proizvod je statički povezan kod koji se ne izvršava na VM, ali u sebi sadrži njene neophodne komponente poput garbage collector-a. Takođe, sadrži i klase i pakete, runtime klase biblioteka i statički povezan mašinski kod. [3]

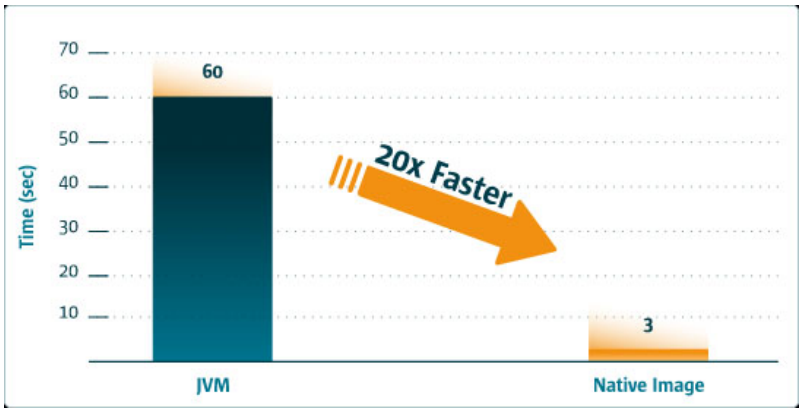
Originalni, čisti izvorni kod se prevodi do bytecode-a na poznat način. Native Image builder je zadužen za određivanje zavisnosti. Vrši se statička analiza dosežnosti klasa i metoda. Sav dosežan kod se zatim kompajlira ahead-of-time za ciljani operativni sistem i arhitekturu. Rezultujući kod se brže izvršava i zauzima manje memorije u odnosu na JVM.

## 5.1 Problemi i optimizacije

Problem na koji se nailazi u nekim prilikama je nemogućnost primene ahead-of-time kompajliranja. U tom slučaju se pokreće agresivnije analiziranje korišćenjem pretpostavki (eng. *closed-world optimization*). Na taj način sav dostižan kod biće poznat tokom prevođenja. Ukoliko se kod ne može optimizovati, generiše se fallback slika koja se pokreće na Java HotSpot VM. Da bi se izbegle ovakve situacije, moguće je napraviti konfiguracioni fajl sa razrešenjem.

### 5.1.1 Dinamičko učitavanje i refleksija

Svaka klasa kojoj se pristupa preko imena u runtime-u mora biti označena u build time-u. Da bi se razrešili svi elementi kojima se pristupa reflektivno, Native Image pokušava statičkom analizom da detektuje pozive iz Reflection Application Programming Interface (API).




Slika 3: Prikaz ubrzanja pri korišćenju Native Image kompajlera u odnosu na JVM

Dynamic Proxy su podržani ahead-of-time. Statička analiza jednostavno presreće pozive metoda newProxyInstance i getProxyClass i pokušava da automatski odredi listu interfejsa.

### 5.1.2 Java Native Interface

JNI omogućava native kodu da pristupa Java objektima, klasama, metodama i poljima preko imena. Gledajući da je princip sličan kao i korišćenje reflection API-a u Javi, neophodno je specificirati generisanje slike u konfiguracionom fajlu. Native image nudi svoj interfejs kao dopunu JNI. On omogućava jeftiniju i jednostavniju alternativu JNI. Takođe omogućava pozive između Java i C koda, kao i pristup C strukturama podataka iz koda pisanog u Javi.

# 6 Zaključak

Mnogi optimizacioni problemi su NP-kompletni (Np-teški ili neodlučivi), pa se samim tim većina algoritama za optimizaciju oslanja na aproksimaciju i heuristiku. I pored toga, algoritmi rade uglavnom prilično dobro. Programer ima jako veliku ulogu u dobijanju efikasnog koda, ali ako je algoritam koji se koristi loš, optimizacija ga ne može učiniti bržim. Bitno je napomenuti i da optimizacija ne bi trebalo da menja ispravnost koda. Očekuje se da program koji je optimizovan, daje iste izlaze, za iste ulaze, kao i početni neoptimizovani program. Optimizacije u navedenim kompajlerima su ključna stvar za dostizanje dobrih performansi koda. Jedni drugima su uglavnom konkurencija, pa će kao i dosadašnje napredne optimizacije u okviru njih, i dalja istraživanja u ovoj oblasti biti usmerena ka poboljšanjima, kako rešavanjem njihovih nedosatataka, tako i mana u poređenju sa konkurentima.

# Literatura

[1] Compiler Design - Code Optimization. on-line at: https://www.tutorialspoint.com/compiler_design/compiler_design_

code_optimization.htm.

[2] GraalVM Manual - Native Image. on-line at: https://www.graalvm.org/22.0/reference-manual/native-image/#:~:text=Native%20Image%20is%20a%20technology%20to%20ahead%2Dof%2Dtime%20compile%20Java%20code%20to%20a%20standalone%20executable%2C%20called%20a%20native%20image.

[3] GraalVM Manual - Native Image. on-line at: https://www.graalvm.org/22.0/reference-manual/native-image/#:~:text=This%20executable%20includes,thread%20scheduling%20etc.

[4] Kompajleri Stanford. on-line at: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/.

[5] LLVM's Analysis and Transform Passes. on-line at: https://llvm.org/docs/Passes.html#utility-passes.

[6] opt - LLVM optimizer. on-line at: https://releases.llvm.org/13.0.0/docs/CommandGuide/opt.html.

[7] Options That Control Optimization. on-line at: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[8] Writing an LLVM Pass. on-line at: https://llvm.org/docs/WritingAnLLVMPass.html.

[9] GCC vs. Clang/LLVM: An In-Depth Comparison of C/C++ Compilers, 2019. on-line at: https://alibabatech.medium.com/gcc-vs-clang-llvm-an-in-depth-comparison-of-c-c-compilers-899ede2be378.

[10] Agner Fog. *Optimizing software in C++*. 2004.

[11] Milena Vujošević Janičić. *Konstrukcija kompilatora*.

[12] Filip Marić. *Konstrukcija kompilatora*.

[13] Aleksandar Samardzić. *GNU programski alati*. 2002.

[14] Shekhar Kumar Singh. Code Optimization in Compiler Design. on-line at: https://www.codingninjas.com/codestudio/library/code-optimization-in-compiler-design.

[15] Serdar Yegulalp. What is LLVM? The power behind Swift, Rust, Clang, and more. *InfoWorld*, 2020.