Optimizacije dostupne u okviru kompajlera Native Image

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

Nikola Mićić Mirko Ilić Sara Kapetinić Anđela Ilić
nikolamicic065@gmail.com, ilicmirko07@gmail.com,
sara.kapetinic.sk@gmail.com, mladotmidji@gmail.com

Novembar 2022.

Sažetak

Programiranje u višim programskim jezicima nam omogućа da koristimo različite apstraktne konstrukcije. Međutim, moguća negativna strana korišćenja apstraktnih konstrukcija je smanjenje optimalnosti koda. Iz ovih razloga prilikom kompilacije, kompajleri samoinicijativno koriste optimizacije. U ovom radu osvrnućemo se na optmizacije u okviru *Native Image* kompajlera.

Sadržaj

1 Uvod

Optimizacija koda je proces transformacije izvornog koda kojim ga unapređujemo kako bi kod upotrebljavao što manje resursa (npr. memorije) i kako bi samo izvršavanje bilo brže.

Prilikom optimizacije koda u višim programskim jezicima veoma je važno da se tim promenama ne menja semantika programa, već samo da se unapređuje i poboljšava brzina izvršavanja koda i smanji upotreba resursa ukoliko je to moguće. Sam proces optimizacije bi trebalo da bude brz i da ne usporava celokupno prevođenje koda.

U prvom delu ovog seminarskog rada ćemo se baviti optimizacijama dostupnim u okviru kompajlera Native Image, dok će se drugi deo rada fokusirati na kompajlere LLVM i GCC, kao i na sličnosti i razlike između njih.

1.1 Optimizacione tehnike

Optimizacionih tehnika ima jako puno. Možemo ih podeliti na osnovu polja delovanja na:

- one koje deluju na ceo program
- one koje deluju na deo programa

Druga podela je na osnovu mašinske zavisnosti na:

- mašinski zavisne
- mašinski nezavisne

Mašinski zavisne optimizacije uzimaju u obzir arhitekturu mašine na kojoj se izvršavaju. Naime nakon generisanog ciljnog koda mogu da koriste apsolutne adrese CPU registra i na taj način unaprede memorijsku hijerahiju.
Mašinski nezavisne optimizacije vrše optimizaciju međukoda i optimizacione tehnike ove vrste ne uključuju CPU registre i memorijske lokacije. Takođe, ove promene se mogu izvršiti na svim tipovima procesora. Vid ovakve optimizacije se vrši na međureprezentaciji koda (IR - reprezentaciji) koja je nezavisna od same mašine na kojoj se izvršava program.

1.1.1 Mašinski nezavisne optimizacije

Kroz nekoliko sledećih primera čemo razmotriti mašinski nezavisne optimizacije koje vrše kompajleri.

Primer 1.

```
while(vrednost < 50){
    brojač=10;
    vrednost = vrednost + brojač;
}
```

Mašinski nezavisna optimizacija će izvući brojač van petlje da se ne bi vršila inicijalizacija svaki put u petlji. Ovaj vid optimizacije je nezavistan od specifične arhitekture. Naziv ove tehnike je pomeranje koda (engl. code motion).

```
brojač=10;
while(vrednost < 50){
    vrednost = vrednost + brojač;
}
```

Primer 2.

```
c = a * b
x = a
d = x*b++4
```

Optimizacija će zameniti pojavljivanje x sa a. Nakon ove optimizacije a * b će biti identifikovan kao običan podizraz. Ovaj vid optimizacije je poznat kao propagiranje promenljive (engl. variable propagation).

```
c = a * b
x = a
d = a*b++4
```

Primer 3.

```
c = a * b
x = b
d = a*b++4
```

Optimizacija će eliministati deo koda koji nema uticaj na izvršavanje programa (mrtav kod) i ukloniti naredbu dodele x = b. Ovakva tehnika optimizacije je poznata kao eliminacija mrtvog koda (engl. dead code elimination).

```
c = a * b
d = a*b++4
```

Primer 4.

```
a = b * 4
```

Operacija množenja će biti zamenjena operacijom bitovskog šiftovanja ulevo. Ova operacija je jeftinija u odnosu na operaciju množenja, a ovakav tip optimizacije je poznat kao engl. strength reduction

```
a = b << 2
```

1.1.2 Mašinski nezavisne optimizacije

Kada se izgeneriše ciljani kod za arhitekturu na kojoj će se kod izvršavati onda se pristupa mašinski nezavisnim optimizacijama. Mašinski nezavisne optimizacije se trude da u najvećoj meri iskoriste memorijsku hijerarhiju.

Osnovni blokovi (engl. Basic blok) su delovi koda koji se izvršavaju sekvencijalno. Ovi blokovi nemaju nikakve skokove što znači ako se izvrši prva insturkcija izvršiće se i sve ostale. Blokovi igraju bitnu ulogu u identifikaciji promenljivih koje se koriste u okviru jednog bloka. Ako se promenljivsa koristi u više blokova onda memorijski registar koji je alociran za tu promenljivu ne treba da bude ispražnjen nakon izvršavanja.

Graf kontrole toka je jedan naćin za prikazivanje osnovnih blokova. Pomoću njega možemo videti kako se menja tok programa. Graf nam može pomoći u optimizaciji u otkrivanju neželjenih petlji u programu.

Optimizacija petlji predstavlja bitan deo koda jer će se u njima određeni kod izvršavati više puta tako da će uštede u vremenu biti veće nego kada bi se taj deo koda izvršavao sekvencijalno.

Jedna od značajnijih primera je za optimizaciju petlji je izdvajanje invarijanti ispred petlji.

Raspoređivanje instrukcija predstavlja optimizaciju koda u kojoj pravimo različiti raspored insturkcija sa ciljem da poboljšamo performanse.

Zavisni podaci u mašinskom kodu predstavljaju insturkcije čije je ponašanje međuzavisno. Možemo da posmatramo kao skup insturkcija koje je nemoguće urediti na drugačiji način. Graf zavisnosti podataka je graf koji prikazuje zavisne podatke. To je direktan aciklični graf jer su nemoguće ciklične zavisnosti. Ideja u ovom vidu optimizacije je da se pomoću grafa napravi topološko sortiranje podataka i da se instrukcije urede u tom redosledu. Ovaj problem predstavlja NP-težak problem i za njegovo rešenje se uglavnom koriste heuristike.

1.1.3 Alternativni vidovi podele

Podela se može izvršiti i na osnovu programske zavisnosti na:

- programski zavisne
- programski nezavisne

Većina viših programskih jezika deli slične apstrakcije i programske konstrukte kao što su if, switch, for, itd. Neke optimizacione tehnike se mogu deliti između jezika, dok neke optimizacije mogu da se primene samo na određene više programske jezike. Takođe neke karakteristike čine neke optimizacije lakše.

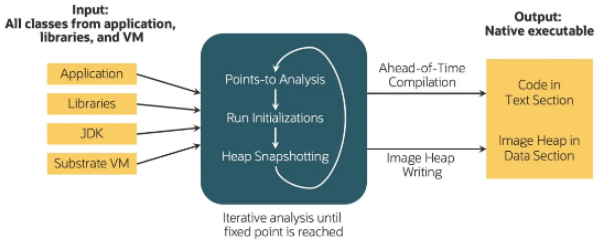
1.2 Native image

Native Image predstavlja drugačiji način za izvršavanje Java koda i tehnologiju koja se koristi za *ahead-of-time*[1] kompilaciju *Java* koda. Pored same AoT kompilacije, podrzumevani način kompajliranja Java koda je Just in Time(JIT).[2] Uz pomoć navedene tehnologije dobijamo samostalan izvršiv kod(*executable*), matičnu sliku(*native image*). U ovaj kod su ukljucene klase aplikacije, klase od kojih je aplikacija zavisna, *runtime* biblioteke i statički povezan izvorni kod *JDK*-a. Za izvršavanje ovog koda nije potrebna Java virtuelna mašina(*JVM*), ali uključuje različite neophodne komponente iz *Substrate VM*-a[3]. Neke od tih komponenti su upravljanje memorijom i raspoređivanje tredova(thread scheduling). *Native Image* podržava jezike koji se izvršavaju na *JVM*-u, kao što su Java, Scala, Clojure, Kotlin, a dobijena slika može da se izvršava u dinamičkim jezicima kao što su Python, R, Ruby, JavaScript.
U *Native Image*-u postoji jasna razlika između:

- *image build time*-a
- *image run time*-a

Tokom *image build tima*-a vrši se statička analiza koda i zahteva se pretpostavka zatvorenog sveta[4](*close-world assumption*). Pretpostavka zatvorenog sveta zahteva da sav *bytecode* i sve klase koje su dostupne u *runtime*-u moraju biti poznate u *build-time*-u. Traže se svi dostupni metodi od početne tačke aplikacije, i **samo** ovakvi metodi su deo *ahead-of-time* kompilacije koja generiše matičnu sliku(*native image*). Ovo dalje podrazumeva da tokom *image run time*-a je nemoguće obrađivanje novih podataka koji nisu bili dostupni u *image build time*-u. Na slici 1 prikazan je rad kompajlera Native image.

[1] Ahead-of-time(AoT) - podrzumeva generisanje objektnog koda pre samog izvršavanja programa.
[2] JIT- kompilacija tokom izvršavanja programa.
[3] Substate VM - ime za *runtime* komponente
[4] Pretpostavka zatvorenog sveta zapravo znači da postoji tačna izjava za koju se zna da je tačna, dok se za sve ostale za koje se ne zna istinitost smatra da su netačne.



Slika 1: Native image

2 Native Image - Napredne optimizacije

Native image[6] koristi optimizacije kako bi smanjio memorijsko iskorišćenje i vreme pokretanja aplikacije. Kako bi se ovo postiglo pretpostavlja se da važi pretpostavka o zatvorenom svetu. Ovo znači da mnogi programi nisu pogodni za Native Image optimizacije. U takvim situacijama se kreira takozvana rezervna slika(*fallback image*) koja koristi *Java HotSpot VM* i zahteva korišćenje JDK-a za izvršavanje programa.

Neke od optimizacija koje koristi *Native Image* jesu Optimizacija vođena profilom(*Profile-Guided Optimization*), Inicializacija klasa(*Class initialization*), Organizovanje memorije(*Memory management*).

2.1 Profile guided optimization - PGO

Optimizacija vođena profilisanjem[7] je još jedna od optimizacija koja se koristi kako bi se dobile još bolje performanse. Tačnije kod se izvršava veliki broj puta i na osnovu informacija o izvršavanju formiramo podatke profilisanja(*profiling data*). Ideja ovih podataka je da definišemo koji delovi koda se više, a koji manje koriste kako bi veću pažnju stavili na kod koji se više koristi. *Native Image* ove podatke koristi da bi se kod dodatno optimizovao.
PGO[8] uključuje naredna poboljšanja:

- *Inline* - Ako funkcija A često poziva funkciju B, a funkcija B je relativno mala, ova optimizacija će uvrstiti funkciju B u funkciju A
- *Virtual Call Speculation* - Ako virtuelni poziv ili neki drugi poziv preko pokazivača često cilja određenu funkciju, optimizacija vođena profilisanjem može umetnuti direktni poziv ka ciljanoj funkciji, umesto da se do te funkcije dolazi preko niza pokazivača.
- *Register Allocation* - Optimizacija bazirana na podacima dobijenim profilisanjem rezultira boljom alokacijom registara.
- *Size/Speed Optimization* - Funkcije gde program provodi najvise vremena izvršavajući ih, mogu biti optimizovane radi bržeg izvršavanja.
- *Function layout* - Na osnovu grafa poziva funkcija, funkcije koje imaju tendenciju da budu na istom putu izvršavanja su postavljene u istu sekciju.

```
1000 #!/bin/bash
native-image --pgo-instrument imePrograma
```

Listing 1: Pokretanje optimizacije vođene profilisanjem

Gradimo izvornu sliku (eng *Native image*), tako što dodajemo navedenu opciju(2.1), čije će izvršenje prikupiti profile učestalosti izvršavanja koda.

2.2 Inicijalizacija klasa

Semantika Jave zahteva da se klase inicializuju prvi put kada su pomenute tokom izvršavanja programa, što ima negativne posledice na *ahead of time* kompilaciju i znatno smanjuje performanse. Kako bi rešio ovaj problem *Native Image* omogućava inicializaciju klasa u *run time*-u. Neke od klasa mogu biti inicijalizovane u toku *image build* faze, što znači da inicijalizacija tokom *run time*-a nije neophodna. Sve statičke informacije o inicializaciji nalaze se u slici koja je formirana.

Međutim, semantika inicijalizacije klasa u Javi nameće određena ograničenja koja komplikuju ovaj proces. Kako bismo dobili što bolje rezultate prilikom korišćenja *Native Image*-a imamo na raspologanju nekoliko opcija:

- *build time* inicializacija
- automatska inicijalizacija bezbednih klasa
- eksplicitno označavanja klasne inicializacije

Build time inicializacijom se inicializuje se većina klasa tokom build time-a. Ovo uključuje garbage kolektore, važne JDK klase, deoptimizere itd. Za sve ove konstrukte Native image daje podršku kako bi semantika izvršavanja ostala ista.

Automatska inicializacija bezbednih klasa - za korisnički definisane klase *Native Image* pokušava da zaključi koje od klasa mogu bezbedno biti inicijalizovane u *run time*-u. Klasa se smatra sigurnom kada su svi njeni relevantni nadtipovi bezbedni i ukoliko inicijalizator klase ne poziva nijednu nebezbednu metodu niti inicijalizuje druge nebezbedne metode. *Metoda je* ***nebezbedna*** *ako*:

- Tranzitivno poziva izvorni kod(npr: *System.out.println*): Izvorni kod se ne analizira tako da *Native Image* ne može da zna da li su izvršene neke nelegalne radnje.
- Koristi virtualne metode: Dodavanjem ovih metoda u obradu bi znatno proširili prostor pretrage.
- Ukoliko je zamenjen *Native Image*-om. Pokretanje inicijalizacije na zamenjenim metodama dalo bi drugačije rezultate u *JVM*-u nego u generisanoj slici. Pa kao rezultat bezbednosna analiza bi smatrala neke metode bezbednim ali bi njihovo izvršavanje dovelo do nelegalnih radnji.

Za označavanje klase, u okviru *Eksplicitnog označavanje klasne inicalizacije*, dostupna su nam dva taga *–initalize-at-build-time* i *–initialize-at-run-time*. Pomoću ovih tagova možemo označiti ceo paket ili individualnu klasu.

Svaka klasa moze biti izgrađena u okviru *build time-a* ili *run time-a* . Za postavljanje ove opcije koriste se dve oznake (eng *Flag*) : **–initialize-at-build-time** and **–initialize-at-run-time**.

2.3 Iskorišćenje memorije

Native Image se izvršava na *runtime* sistemu koji je obezbeđen od strane *GraalVM*-a, koji sadrži neophodne konstrukte kao što je *upravljanje memorijom*. Objekti koji su generisani tokom izgradnje slike čuvaju se na takozvanom *"Java heap"*-u. Hip se kreira nakon pokretanja slike i može da varira u veličini tokom izvršavanja, kada se napuni, poziva se *garbage collector*. *Native image* koristi generacijsku *kolekciju otpada*. Generacijski *Garbage Collector*(GC) podrazumeva postojanje mlade i stare populacije, i u zavisnosti od implementirane logike imamo različite generacijske GC.

Native Image podržava različite *garbage collector*-e(GC):

- Serijski GC je podrazumevani GC i pogodan je za mali memorijski prostor i mali Java hip.
- G1 GC je višenitni GC koji je optimizovan da smanji *stop-the-word* pauze i samim tim poboljša latenciju dok istovremeno postiže visoku propusnost. Trenutno se koristi samo za Linux.
- Epsilon GC je sakupljač otpada bez operacija(*no-op garbage collector*) koji se nikada ne poziva i samim tim nikada ne briše alociranu memoriju. On se danas primarno koristi za male programe koji alociraju malu količinu memorije na hipu.

Metrike za merenje performansi GC jesu:

- *Throughout* odnosno propusnost - procenat ukupnog vremena koje nije utoršeno na čišćenje memorije u toku dužeg vremenskog perioda.
- *Latency* odnosno latencija ili kašnjenje - odziv aplikacije. Pauze prouzrokovane čišćenjem memorije negativno utiču na odziv aplikacije.
- *Footprint* odnosno otisak - skup procesa meren u stranama i linijama keša.

Određivanje kako će se hip podesiti zapravo uvek predstavlja kompromis između ove tri metrike. Podešavanje koja *Native Image* zapravo postavlja jesu maksimalna veličina hipa (gornja granica *Java heap*-a), minimalna veličina hipa (minimalna količina memorije koju GC može da pretpostavi da može biti alocirano) i veličina mlade generacije (količina memorije koja se može alocirati bez prethodnog čišćenja memorije).

```
1000 #!/bin/bash
     native-image --gc=serial imePrograma
```

Listing 2: Serial Garbage Collector-a.

```
1000 #!/bin/bash
     native-image --gc=G1 imePrograma
```

Listing 3: G1 Garbage Collector-a.

Na slikama 2.3 i 2.3, prikazano je kako možemo izgraditi izvornu sliku (eng Native image), koja koristi serijski GC ili G1 GC, sa podrazumevanim opcijama.

3 O LLVM-u i LLVM/GCC kompajleru

Jedan od poznatijih kompajlera otvorenog koda jeste u okviru LLVM projekta *Clang*. Česta zabuna je da je ceo LLVM projekat samo *Clang* kompajler za C++. Osvrnućemo se na LLVM projekat kao i na optimizacije koje su u njemu dostupne kako bismo ih kasnije lakše uporedili.

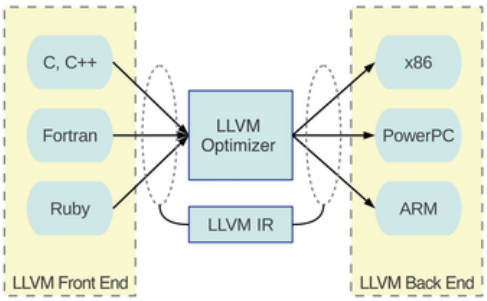
3.1 LLVM

LLVM[1] je skup različitih biblioteka i alata koji čine jednu kompajlersku strukturu. Započeo je kao istraživački projekat na Univerzitetu Ilinois. Imao je za cilj da obezbedi modernu strategiju kompilacije zasnovanu na SSA[5] koja moze da podrži i statičku i dinamičku kompilaciju proizvoljnih programskih jezika.

LLVM[4] projekat se sastoji od vise komponenti. Sama srž projekta se naziva LLVM koja sadrži sve alate, biblioteke i datoteke zaglavlja potrebnih za obradu međureprezentacija i pretvaranje u objektne datoteke. Alati uključuju asembler, disasembler[5], analizator bitkodova i optimizator bitkodova. Takođe sadrži i osnovne regresione testove.

C-oliki jezici koriste prednji deo Clanga-a. Ova komponenta kompilira C, C++, Objective C i Objective C++ kod u LLVM bitkod, a zatim u objektne datoteke koristeći LLVM.

LLVM[3] može da obezbedi srednje slojeve kompletnog sistema kompajlera, uzimanje koda srednje zastupljenosti (IR[6]) od kompajlera i emituje optimizovani IR. Ova nova IR se zatim može konvertovati i povezati u mašinsko zavisni asemblerski kod za ciljnu platformu. LLVM može da prihvati IR iz GNU Compiler Collection (GCC) alata, omogućavajući mu da se koristi sa širokim nizom postojećih *front-ends* kompajlera napisanih za taj projekat. Na slici 2 je prikazana infrastruktura LLVM kompajlera, odnosno prednji, središnji i zadnji deo kompajlera.



Slika 2: Infrastruktura LLVM kompajlera

[5]Disasembler - kompjuterski program koji prevodi mašinski jezik u asemblerski
[6]IR - Intermediate Representation

3.2 LLVM/GCC

Komanda llvm-gcc je prednji kraj LLVM-a. Predstavlja modifikovanu verziju gcc kompajlera za kompajliranje C/ObjC u izvorne objekte, LLVM bitkod ili LLVM asmeblerski jezik u zavisnosti od opcija. Podrazumevano llvm-gcc kompilira u izvorne objekte isto kao i gcc. Prilikom prevođenja llvm-u se mogu dodati dodatne opcije koje omogućavaju različite načine prevođenja. Na primer[2] ukoliko bismo dodali opciju -llvm-emit -c biće generisani LLVM bitkod fajlovi. Ukoliko iskoristimo opciju -llvm-emit -S biće generisan LLVM asembler.

Pošto je izveden iz *GNU Compiler Collection*-je llvm-gcc ima mnoge gcc-ove karakteristike i podržava većinu gcc-ovih opcija. Rukuje sa mnogo gcc-ovih ekstenzija za programski jezik C.

4 Sličnosti i razlike LLVM/GCC-a i Native Image-a

LLVM ne zavisi od izvornog koda i ciljane arhitekture. Native image je napravljen za jezike koji se izvršavaju na JVM-u. LLVM je open-source kod, dok Native image ima open source i enterprice verziju. LLVM podržava stastičku i dinamičku optimizaciju dok Native image samo statičku.

Kako se Clang često koristi kao sinonim za LLVM, možemo reci da oboje daju znatno bolje rezultate pri kompajliranju ciljnog jezika u odnosu na standardni kompilator. Clang nad GCC-om a Native image nad standardnim java interpreterom. Clang i Native Image rade na izvornom kodu. Oboje izvorni kod prevode do objektnih datoteka odnosno executabla.

Kao i što LLVM ima ideju da pravi nezavisne biblioteke koje se mogu ponovo koristiti, tako i Native image takođe pravi nezavisne executable koji se samostalno mogu izvršavati ali ne uvek i ponovo koristiti. Iako je Native image prvenstveno kompajler za jezike koji se pokrecu na JVM-u, on obezbeđuje API za C-olike jezike. C API je dostupan kada je Native Image izgrađen kao deljena biblioteka, a njene deklaracije su uključene u datoteku zaglavlja koja se generiše tokom izrade.

U tabeli 1 su prikazane sličnosti i razlike navedenih kompajlera.

LLVM/GCC	Native Image
Ne zavisi od koda i ciljane arhitekture	Napravljen za jezike koji se izvršavaju na JVM-u
Open-source kod	Open-source kod i enterprice verzija
Statička i dinamička optimizacija	Stastička optimizacija
Radi na izvornom kodu	Radi na izvornom kodu
Prevodi izvorni kod do objektnih datoteka	Prevodi izvorni kod do objektnih datoteka
Pravi nezavisne biblioteke koje se mogu ponovo koristiti	Pravi nezavisne executable koji se mogu samostalno izvršavati

Tabela 1: Sličnosti i razlike LLVM/GCC-a i Native Image-a

5 Zaključak

Optimizacija koda je bitna jer poboljšava prenosivost kompajlera ciljnom procesoru i omogućava potrošnju manjeg broja resursa. Optimizovani kod se brže izvršava, efikasno koristi memoriju, daje bolje performanse i često promoviše ponovnu upotrebljivost.

Dobro je poznato da vam *GraalVM Native image* daje brzo pokretanje i troši manje memorije. GraalVM 19.2.0 Enterprise vam donosi pojednostavljen način korišćenja optimizacija vođenih profilima (PGO) - uz njegovu pomoć moguće je obučiti aplikaciju za određena radna opterećenja i značajno poboljšati vrhunske performanse. Native image uključuje alternativni backend koji koristi LLVM središnju reprezentaciju i LLVM kompajler za proizvodnju izvornih izvršnih datoteka.

Literatura

[1] Llvm. https://www.inf.ed.ac.uk/teaching/courses/ct/slides-16-17/llvm/2-2008-10-04-ACAT-LLVM-Intro.pdf.

[2] Llvm command. https://releases.llvm.org/2.8/docs/CommandGuide/html/llvmgcc.html.

[3] Llvm explanation. https://www.heavy.ai/technical-glossary/llvm/.

[4] Llvm offical page. https://llvm.org/.

[5] Llvm ssa. https://wiki.aalto.fi/display/t1065450/LLVM+SSA.

[6] Native image. https://www.graalvm.org/22.0/reference-manual/native-image/.

[7] Pgo. https://docs.oracle.com/en/graalvm/enterprise/20/docs/reference-manual/native-image/PGO/.

[8] Pgo microsoft. https://learn.microsoft.com/en-us/cpp/build/profile-guided-optimizations?view=msvc-170.