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 omoguća da koristimo različite apstraktne kostrukcije, uslove, petlje itd. Međutim, moguća negativna strana korišćenja ovih apstraktnih konstrukcija je smanjenje optimalnosti koda. Iz ovih razloga prilikom kompilacije, kompajleri samoinicijativno koriste optimizacije. Koriste različite transformacije na petljama uslovnim konstrukcijama, rekurzivnim pozivima koristeći set instrukcija *ISA*. Zato se treba fokusirati na korišćenje što bržih algoritama a ne na manuelanu optimizaciju koda.

Sadržaj

1 Kako se optimizacije dele i sta sve obuhvataju?

Optimizacija koda je proces transformacije koda koji pokusava da unapredi kod da bi upotrebljavao manje resursa (npr. memorije...) i da bi samo izvrsavanje koda bilo brze.

U optimizaciji viših programskih jezika proces optimizacije mora da prati tri pravila:

- Optimizacija ne treba da menja semantiku programa
- Optimizacija treba da poboljša brzinu izvršavanja koda, i smanji upotrebu resursa ako je to moguće
- Sam proces optimizacije bi trebalo da bude brz i da ne usporava celokupno prevodjenje

Optimizacionih tehnika ima jako puno. Mozemo 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 medjukoda i optimizacione tehnike ove vrste ne uključuju CPU registre i memorijske lokacije. Takodje, ove promene se mogu izvršiti na svim tipovima procesora.

```
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.

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

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 izmedju jezika, dok neke optimizacije mogu da se primene samo na odredjene više programske jezike. Takdoje neke karakteristike čine neke optimizacije lakše.

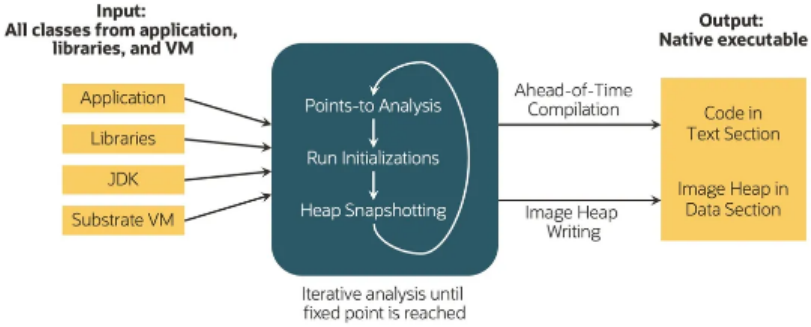
2 Koje napredne optimizacije su prisutne u okviru kompajlera Native Image?

2.1 Native Image

Native Image predstavlja drugačiji način za izvršavanje Java koda. *Native Image* je tehnologija koja se koristi za *ahead-of-time*(pre samog izvršavanja) kompilaciju *Java* koda koja nam daje samostalan izvršiv kod(*executable*) matičnu sliku(*native image*). Ovaj kod sadrži 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 *Šubstrate VM"*-a, kao što je upravljanje memorijom i sl. *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[1](*close-world assumption*) što znači da sav *byte-code* i sve klase koje su dostupne u *run-time*-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.



Slika 1: Native image

2.2 Optimizacije u okviru *Native Image*-a

Native image[2] 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

[1]Pretpostavka zatvorenog sveta zapraavo znači da postzoji tačna izjava za koju se zna da je tačna, dok se za sve ostale za koje se ne zna istinitost smatra da su netačne.

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*).

2.3 *Profile guided optimization* PGO

Optimizacija vođena profilisanjem[3] - kako bi dobili još bolje performanse, koristi se i ova vrsta optimizacija. 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 i u to ime bacimo akcent na delove koda koji se češće izvršavaju. Kada ove podatke ubacimo u *Native Image* on ih koristi da bi kod dodatno optimizovao.
PGO[4] ukljucuje naredna poboljsanja:

- *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 neku funkciju, optimizacija vođena profilisanjem može umetnuti uslovno izvršeni direktni poziv ciljanoj funkciji, a direktni poziv može biti umetnut.
- *Register Allocation* - Optimizacija bazirana na profilnim podacima rezultira boljom alokacijom registara.
- *Size/Speed Optimization* - Funkcije gde program provodi najvise vremena izvrsavajući 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.

2.4 Inicijalizacija klasa - *Class initialization*

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 se adresovao 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 kao ni druge razne provere. Sve statičke informacije o inicializaciji nalaze se u slici.

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 oynačavanja klasne inicializacije

Build time inicializacija - 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 ni jednu 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.

Eksplicitno označavanje klasne inicalizacije - za označavanje klase dostupna su nam dva taga *–initalize-at-build-time* i *–initialize-at-run-time*. Pomoću ovih tagova možemo označiti ceo paket ili individualnu kalsu.

2.5 Iskorišćenje memorije - *Memory management*

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 keira nakom pokretanja slike i može da varira u veličini tokom izvršavanja, kada se napuni, poziva se *garbage collector*. *Native image* koristi generacijsku *kolekciju otpada*. Generacijski 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. Ovo je zapravo i podrazumevani GC.
- 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 metirke. Podešavanje koja *Native Image* zapravo postavlja jesu:

- Maksimalna veličina hipa - gornja granica *Java heap*-a, ali ne gornja granica memorije.
- Minimalna veličina hipa - minimalna količina memorije koju GC može da pretpostavi da može biti alocirano.
- Veličina mlade generacije - količina memorije koja se može alocirati bez prethodnog čišćenja memorije.

3 Uopsteno o LLVM-u i LLVM/GCC kompajleru

3.1 LLVM

LLVMM[1] je skup različitih biblioteka i alata koji čine jednu kompajlersku strukturu. Započeo je kao istraživački projekata na Univerzitetu Ilinois sa ciljem da obezbedi modernu strategiju kompilacije zasnovanu na SSA koji moze da podrži i statičku i dinamičku kompilaciju proizvoljnih programskih jezika.

LLVM projekat se sastoji od vise komponeneti. Sama srž projekta se naziva "LLVM"koja sadrži sve alate, biblioteke i dadoteke zaglavlja potrebnih za obradu međureprezentacija i pretvaranje u objektne dadoteke. Alati uključuju asembler, disasembler[2], 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 dadoteke koristeći LLVM.

LLVM može da obezbedi srednje slojeve kompletnog sistema kompajlera, uzimanje koda srednje zastupljenosti (IR[3]) 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.

3.2 GCC/LLVM

Komanda llvm-gcc je prednji kraj LLVM C-a. Prdstavlja 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. Prevođenjem korišćenjem llvm mogu se dodati dodatne opcije koje omogućavaju različite načine prevođenja. Na primer ukoliko bismo dodali opciju -llvm-emit -c biće generisani LLVM bitksod 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.

[2]Disasembler - kompjuterski program koji prevodi mašinski jezik u asemblerski
[3]IR - Intermediate Representation

GCC/LLVM	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 optimimzacija
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, ali se ne mogu uvek i koristiti

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

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

LLVM ne zavisi od izvornog koda i ciljane arhitekture. NI je napravljen za jezike koji se izvršavaju na JVM-u.

LLVM je open-source kod, dok NI ima open source i enterprice verziju.

LLVM stastička i dinamička optimizacija dok je NI smao statička.

Kako se Clang često koristi kao sinonim za LLVM, možemo reci da oboje daju znatno bolje rezultate pri kompajliranju targetovanog jezika u odnosu na standardni kompilator. Clang nad GCC-om a NI nad standardnim java interpreterom.

Clang i NI rade na izvornom kodu

Oboje izvorni kod prevode do objektnih datoteka odnosno executabla.

Kao i sto LLVM ima ideju da pravi nezavisne biblioteke koje se mogu ponovo koristit. NI takođe pravi nezavisne executable koji se samostalno mogu izvršavati ali ne uvek i ponovo koristiti.

5 Zaključak

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.

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

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

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