GraalVM Native Image

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

David Nestorović, Momčilo Knežević
david.nestorovic@gmail.com, moma.knezevic7@gmail.com

22.novembar 2022

Sažetak

Kao jedan od najzastupljenijih jezika današnjice, izdvaja se Java programski jezik. Sa tim u vezi, problemima kao što su brzina prevođenja i izvršavanja programa, posvećuje se posebna pažnja. Kako se sve više svet programiranja usmerava ka Cloud tehnologijama, ovakvi problemi postaju još vidljiviji jer se najčešće ovi resursi naplaćuju. Ovaj rad objašnjava kako se ovi problemi mogu prevaziči korišćenjem GraalVM Native Image-a.

Sadržaj

1 Uvod **2**

2 GraalVM **2**

3 Native Image **3**
3.1 Pregled sistema 3
3.1.1 Analiza pokazivača 4
3.1.2 Izvršavanje inicijalizacija 4
3.1.3 Kreiranje slike hipa 5
3.2 Faze izvršavanja Native Image-a 5

4 Problemi **7**

5 Performanse Native Image-a **8**

6 Zaključak **11**

Literatura **12**

1 Uvod

Tradicionalni način prevođenja Java programa podrazumeva korišćenje JVM interpreter. Međutim bez obzira na visoke performanse ovog interpretera, on nije tako brz kao pokretanje kompajiliranog koda(eng. *compiled code*). Iz tog razloga, JVM takođe poseduje JIT kompajeler koji često korišćene delove koda kompajlira u mašinski kod(C1 JIT kompajler). Zatim ukoliko učestalost izvršavanja nekog dela koda pređe određeni prag taj deo se kompajlira C2 JIT kompajlerom. I ako na prvi pogled ovo deluje kao idealno rešenje, ipak ovaj pristup ima određene mane, kada JVM vrši operacije poput: verifikacije koda, učitavanja klasa, interpretiranje bajtkoda (eng. *bytecode*), prikupljanja profila... za to su mu neophodne kompleksne operacije a samim tim i veliko zauzeće procesora. Pored toga JVM zahteva i određenu memoriju za svoj rad, prvenstveno za smeštanje profila. [1]

Primer 1.1 *Ovaj nedostatak najviše se odražava na Cloud sisteme. U sitaucijama kada se opterećenost nekog servera poveća, Cloud platforma u pozadini servera pokreće nove radnike (eng.* workers*) da bi procesuirali pristigli posao. Prvi zahtev koji novi radnik obrađuje značajno je sporiji (jer radnik mora da obavi određene inicijalizacije pre početka procesuiranja samog zahteva) i naziva se hladni start(eng.* cold start*). Najbolji primer za hladni start daju nam JVM i .NET platforme, koje u hladnom startu moraju izvršiti veliki broj kompleksnih operacija koje su prethodno pomenute.*

Jedan način da se prevaziđe ovaj problem je da se koristi GraalVM Native Image umesto tradicionalog JVM pristupa, na način koji će biti opisan u nastavku.

2 GraalVM

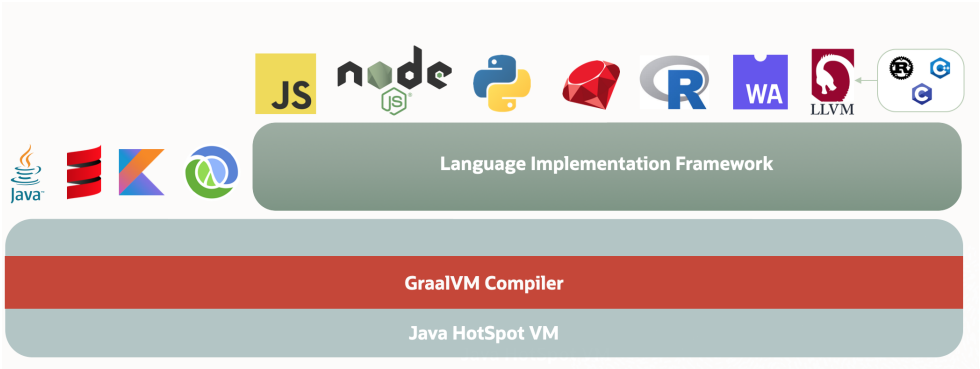
GraalVm je nastao pre deset godina kao istrački projekat Oracle Labs-a, koji i dan danas predstavlja ogranak kompanije Oracle i bavi se istraživanjima na poljima virtuelnih mašina, mašinskog učenja i raznih drugih oblasti. GraalVM je u osnovi kombinacija Java Virtuelne mašine i JDK-a(eng. *Java Development Kit*) i obuhvata:

- GraalVM kompajler(JIT kompajler za Javu)
- GraalVM Native Image(AOT kompajler za Javu)
- Truffle(Biblioteka za izgradnju alata i programskih jezika koja omogućuje pokretanje programskih jezika Python, Ruby, R i drugih na JVM-u)

Na slici 1 prikazana je arhitektura GraalVM-a. U samoj osnovi nalazi se Java Hotspot VM, na koji se naslanja GraalVM kompajler. Jezici iz Java familije prevode se upravo pomoću GraalVM kompajlera, dok je za prevođenje ostalih jezika podržanih od strane GraalVM projekta zadužen dodatni sloj koji u osnovi koristi Truffle [2].
Glavni ciljevi GraalVM projekta su:

- Poboljšanje performansi jezika koji se zasnivaju na Java virtuelnoj mašini kako bi brzina njihovog izvršavanja bila što približnija izvršavanjima programa koji svoj kod prevode u izvršne datoteke (eng. *native languages*) (ujedno i cilj Native Image-a).

- Smanjenje vremena pokretanja aplikacija koje se zasnivaju na Java virtuelnoj mašini, tehnikom kompajliranja unapred i korišćenjem GraalVM Native Image-a (ujedno i cilj Native Image-a).
- Omogućavanje da GraalVM integriše sa sledećim i njima sličnim tehnologijama: Oracle database, OpenJDK, Node.js, Android/iOS...
- Omogućavanje da se u jednom programu koristi više različitih programskih jezika(eng. *polyglot applications*).



Slika 1: Arhitektura GraalVM-a

3 Native Image

Native Image je tehnologija razvijena unutar GraalVM projekta koja omogućuje AOT kompajliranje Java koda u samostalnu izvršnu datotekuI(eng. *standalone executable*) pod nazivom native image. Native Image omogućuje smanjenje memorijskog zauzeća(eng. *memory footprint*) i vremena potrebnog za pokretanje aplikacije(eng. *startup time*). Kako bi ovo bilo moguće, neophodno je da bude ispunje uslov zatvorenog sveta(eng. *closed-world*) što podrazumeva da je sav kod poznat za vreme pravljenja slike(eng. *image build time*)(tj. ne postoji kod koji bi se učitao u vreme izvršavanja aplikacije).

3.1 Pregled sistema

Kao što je već pomenuto, pravljenje native image-a možemo podeliti u dve faze: izgradnja slike(eng. *build time*) i izvršavanje slike(eng. *run time*), pri čemu se gotovo ceo proces obavlja u fazi izgradnje slike. U fazi izvršavanja, obavlja se nekolicina inicijalizacija i to pre pokretanja glavne funkcije(eng. *main function*), npr. mapiranje memorije(eng. *memory-mapping*).

Kao ulaz za proces izgradnje native image-a koristi se Java bajtkod, dobijen kompajliranjem bilo kojeg jezika koji se može prevesti u bajtkod (Java, Skala, Kotlin). Pored toga na isti način se obrađuje aplikacija, biblioteke koje ona koristi, JDK i komponente virtuelne mašine neophodne za proces.

Proces počinje iterativnim ponavljanjem narednih tehnika analize koda sve dok nije dostignut unapred definisan prag zaustavljanja [3]:

- **Analiza pokazivača(eng. *points-to analysis*)**
- **Izvršavanje inicijalizacija(eng. *run initializations*)**
- **Kreiranje slike hipa(eng. *heap snapshotting*)**

Kao rezultzat analize dobija se lista dohvatljivih (eng. *heap reachable*) klasa, metoda i polja, kao i graf dohvatljivih objekata(eng. *object graph*). Potom se dohvatljive metode prevode u mašinski kod, a graf dohvatljivih objekata se upisuje u hip memoriju slike. Na kraju se, mašinski kod čuva u tekst sekciji slike, a hip slike u sekciji podataka. Ceo proces predstavljen je na slici 2.

3.1.1 Analiza pokazivača

Ova tehnika se, u opštem smislu, koristi za identifikovanje veza između pokazivačkih promenljivih i lokacija na koje pokazuju u vreme izvršavanja aplikacije [4].

Primer 3.1 *Rezultat analize pokazivača na narednom kodu, pokazao bi da je skup promenljivih na koje pokazuje pokazivač p zapravo x, y*

```
int x;
int y;
int* p = unknown() ? &x : &y;
```

Listing 1: Primer koda koji se analizira od strane tehnike analize pokaziva;a

Na primeru Native Image, koristi se za određivanje svih klasa, metoda i polja do kojih je moguće doći izvršavanjem koda. Proces počinje analizom svih ulaznih tačaka aplikacije, npr. glavne metode (eng. *main method*), a zatim se iterativnim postupkom utvrđuju sve metode koje su tranzitivno dohvatljive iz polaznih tačaka programa.

Ova tehnika, koristi kompajler iz GraalVM-a za parsiranje Java bajt-koda, kako bi se dobila određena međureprezentacija(eng. *intermediate representation*). U sledećem koraku ova međureprezentacija se prevodi u tzv. graf toka tipova(eng. *type-flow graph*). Svaki čvor u dobijenom grafu održava listu tipova koji mogu dostići ovaj čvor, dok su grane usmere od čvora koji poziva ka čvoru koji je pozvan. U slučaju da se stanje nekog čvora promeni, promena se propagira naviše, tj. svi čvorovi koji pozivaju čvor u kom se desila promena, ažuriraju svoju listu dohvatljivih tipova. Pored osnovnih informacija, Ovako dobijeni graf predstavlja reprezentaciju dohvatljivih tipova cele apliakcuje [4].

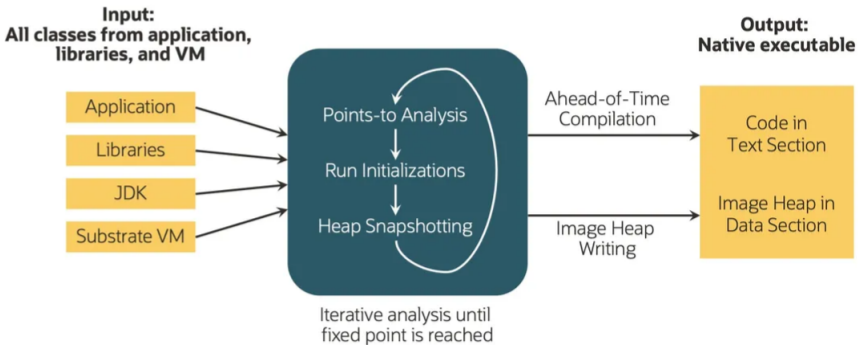
Iako jako teška za izvođenje u fazi prevođenja, uspešno izvođenje ove tehnike omogućuje pojednostavljenja u nekim drugim tehnikama analize koda (recimo u tehnici propagacije konstanti).

3.1.2 Izvršavanje inicijalizacija

Kada je analiza pokazivača dostigla određeni prag, prelazi se na izvršavanje inicijalizacija i to:

- **Inicijalizacija klasa** - svaka klasa u jeziku Java može da ima inicijalizatora klase koji se u ”class”fajlu naziva **clinit**.
- Aplikacija može eksplicitno da registruje funkcije koje se izvršavaju u fazi izgradnje slike i to korišćenjem unapred definisanih okidača(eng. *hooks*):
  - pre analize (eng. *before analysis*)

– tokom analize (eng. *during analysis*)
– nakon analize (eng. *after analysis*)



Slika 2: Pregled izvršavanja Native Image-a

3.1.3 Kreiranje slike hipa

Ova tehnika, kao ulazni parametar koristi neke od podataka dobijenih u analizi pokazivača, a kao finalni proizvod daje graf objekata. Graf objekata se upisuje u native image kao hip slike(eng. *image heap*). Obzirom da je hip slike inicijalizovan u fazi izgradnje slike, dostupan je odmah u fazi pokretanja aplikacije, te se samim tim sva inicijalizacija(koju bi Java radila u fazi izvršavanja) obavila ranije. Praktično, to znači da se vreme za pokretanje aplikacije drastično smanjilo, izmeštanjem posla u fazu izgradnje slike [4].

3.2 Faze izvršavanja Native Image-a

U cilju smanjivanja vremena pokretanja aplikacije(eng. *sturtup time*) i memorijskog zauzeća(eng. *memory footprint*) Native Image kombinuje tehnike kao što su: points-to analysis, inicijalizacija aplikacije u vreme izgradnje aplikacije, heap snapshotting, and AOT kompilaciju. Tokom svog izvršavanja, Native Image prolazi kroz sledeće faze [5]:

1. **Inicijalizacija (eng. *Initializing*)** - podrazumeva pripreme za izgradnju slike i inicijalizaciju posebno definisanog korisničkog koda.
2. **Analiza (eng. *Performing Analysis*)** - pokreće se analiza pokazivača i dobija se kompletna statistika dohvatljivih klasa, metoda i polja. Anomalije u ovim statistikama mogu ukazivati na loše struktuiran kod (velika razlika u broju dohvatljivih u odnosu na ukupan broj definisanih elemenata može sugerisati da je potrebno refaktorisati kod)
3. **Pravljenje univerzuma (eng. *Building Universe*)** - u ovoj fazi se: objedinjuju se rezultati analize, dobijeni elementi se organizuju, uključuju se mehanizmi za manipulaciju dobijenim elemntima i definiše se interfejs za njihovo korišćenje.
4. **Parsiranje metoda (eng. *Parsing Methods*)** - sve dohvatljive metode parsiraju se GraalVM kompajlerom.

5. **Inlajnovanje metoda (eng. *Inlining Methods*)** - parsirane metode iz prethodnog koraka podležu optimizacionoj tehnici *inlining-a*.
6. **Kompilacija metoda (eng. *Compiling Methods*)** - sve dohvatljive metode prevode se od strane GraalVM kompajlera.
7. **Pravljenje slike (eng. *Creating Image*)** - fizički se kreira izvršna datoteka i upisuje na sekundarnu memoriju.

Na slici 3 dat je prikaz ispisa na standardnom izlazu nakon pokretanja Native Image-a. Kao što se sa slike može primetiti, tokom izvršavanja svake od faza, korisnik ima uvid u progres ostvaren tokom određene faze, kao i koliko svaka od faza zahteva vremena i memorisjkog prostora.

```
================================================================================
GraalVM Native Image: Generating 'helloworld'...
================================================================================
[1/7] Initializing...                                           (2.5s @ 0.21GB)
 Version info: 'GraalVM dev Java 11 CE'
[2/7] Performing analysis...  [*******]                         (5.6s @ 0.46GB)
   2,565 (82.61%) of  3,105 classes reachable
   3,216 (60.42%) of  5,323 fields reachable
  11,652 (72.44%) of 16,086 methods reachable
      27 classes,     0 fields, and   135 methods registered for reflection
      57 classes,    59 fields, and    51 methods registered for JNI access
[3/7] Building universe...                                      (0.5s @ 0.61GB)
[4/7] Parsing methods...      [*]                               (0.5s @ 0.86GB)
[5/7] Inlining methods...     [****]                            (0.5s @ 0.73GB)
[6/7] Compiling methods...    [**]                              (3.7s @ 2.38GB)
[7/7] Creating image...                                         (2.1s @ 1.04GB)
   3.69MB (27.19%) for code area:    6,955 compilation units
   5.86MB (43.18%) for image heap:   1,545 classes and 80,528 objects
   3.05MB (22.46%) for debug info generated in 1.0s
 997.25KB ( 7.18%) for other data
  13.57MB in total
--------------------------------------------------------------------------------
Top 10 packages in code area:           Top 10 object types in image heap:
 606.23KB java.util                       1.64MB byte[] for general heap data
 282.34KB java.lang                      715.56KB java.lang.String
 222.47KB java.util.regex                549.46KB java.lang.Class
 219.55KB java.text                      451.79KB byte[] for java.lang.String
 193.17KB com.oracle.svm.jni             363.23KB java.util.HashMap$Node
 149.80KB java.util.concurrent           192.00KB java.util.HashMap$Node[]
 118.07KB java.math                      139.83KB java.lang.String[]
 103.60KB com.oracle.svm.core.reflect    139.04KB char[]
  97.83KB sun.text.normalizer            130.59KB j.u.c.ConcurrentHashMap$Node
  88.78KB c.oracle.svm.core.genscavenge  103.92KB s.u.l.LocaleObjec~e$CacheEntry
      ... 111 additional packages             ... 723 additional object types
                      (use GraalVM Dashboard to see all)
--------------------------------------------------------------------------------
    0.9s (5.6% of total time) in 17 GCs | Peak RSS: 3.22GB | CPU load: 10.87
--------------------------------------------------------------------------------
Produced artifacts:
 /home/janedoe/helloworld/helloworld (executable)
 /home/janedoe/helloworld/sources (debug_info)
 /home/janedoe/helloworld/helloworld (debug_info)
 /home/janedoe/helloworld/helloworld.build_artifacts.txt
================================================================================
Finished generating 'helloworld' in 16.2s.
```

Slika 3: Faze izvršavanja Native Image-a

4 Problemi

Nažalost, nisu sve aplikacije povoljne za ovakvu vrstu optimizacije. Radi ispunjenja uslova zatvorenog sveta, sav kod mora biti poznat u vreme izgradnje slike. Samim tim, da bi se ovi problemi rešili, neke dinamičke osobine jezika Java moraju biti rešene na drugačije načine nego što je to uobičajeno.

Primer 4.1 *Recimo da imamo klasu kojoj se može pristupiti u vreme izvršavanja, na primer* ***Class.forName("myClass")****. Ta klasa mora nekako biti poznata u fazi pravljenja slike. Ovaj problem prevaziđen je uvođenjem konfiguracionih fajlova(eng.* configuration files*) u kojima je potrebno navesti klase očekivane u vremenu izvršavanja aplikacije.*

Primer 4.2 *Slično važi i ukoliko je u kodu korišćena refleksija. Svi elementi koji se mogu očekivati kao rezultat refleksije moraju biti navedeni u konfiguracionim fajlovima.*

Ukoliko aplikaciju nije moguće optimizovati a pritom **ne** postoje konfiguracioni fajlovi, pristupa se pravljenju tzv. rezervne slike(eng. *fallback image*) koja u osnovi pokreće standardni JVM. Ukoliko postoje elemnti koji nisu registrovani u konfiguracionim fajlovima, a oni postoje, biće prijavljenja greška pri prevođenju.

Takođe, u cilju rešavanja raznih drugih situacija u kojima tradicionalni Java pristup(učitavanja koda u vremenu izvršavanja) narušava uslov zatvorenog sveta, osmišljen je princip zamene(eng. *substitution*). U osnovi, ova tehnika je jednostavna, i podrazumeva da svaki kritični deo Java koda treba zameniti implementacijom koja pogoduje Native Image pristupu. Teži deo ovog pristupa ogleda se u tome što nekada nije tako jednostavno zameniti dinamični deo koda, kodom koji bi se mogao u potpunosti prevesti u fazi izgradnje slike [6].

Primer 4.3 *Kako bi se Native Image-u naglasilo da neku klasu koristi kao zamensku klasu, ili da neki metod iz originalne klase treba obrisati ili zameniti, potrebno je koristiti predefinisane anotacije. Najpre je neophodno navesti koju klasu želimo da promenimo koristeći anotaciju* ***@TargetClass*** *koja kao argument prihavata klasu koja se menja. Zatim sledi telo zamenske klase u okviru kojeg je moguće:*

- *koristiti polje iz originalne klase, navođenjem* ***@Alias*** *anotacije*
- *obrisati polje iz originalne klase, navođenjem* ***@Delete*** *anotacije*
- *zameniti metodu originalne klase novom, navođenjem* ***@Substitute*** *anotacije*

```
@TargetClass(OriginalnaKlasa.class)
public final class Target_OriginalnaKlasa {

    @Alias  private static String poljeIzOriginalneKlase;
    @Delete private static String poljeOriginalneKlaseKojeBrisemo;

    @Substitute
    public static String metodaKojuZelimoDaZamenimo() {
        // nova logika
    }
}
```

Listing 2: Primer primene metode zamene

Obzirom da je veliki deo inicijalizacije izmešten u fazu izgradnje slike, a potencijalno se nešto od toga može koristiti i u fazi izvršavanja, neophodno je ostvariti komunikaciju između ove dve faze. Ovaj problem prevaziđen je korišćenjem skladišta ključ-vrednost konstanti. Konceptom nazvanim **image singleton** ostvaruje se konekcija između ove dve faze. Objekat je moguće dodati samo u fazi izgradnje slike korišćenjem metode **add** dok je pristup objektu moguć u bilo kojoj fazi korišćenjem metode **lookup** [3].

5 Performanse Native Image-a

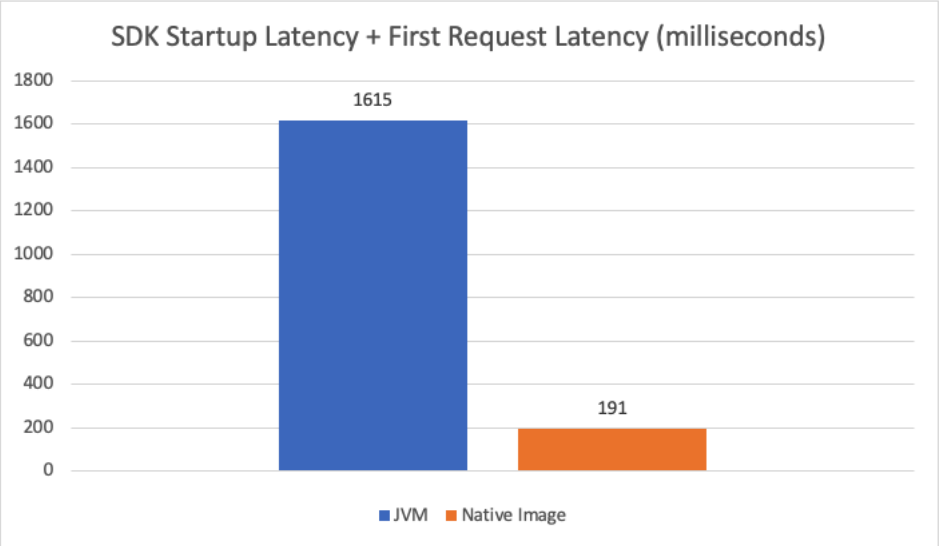
Kako bi se stekao adekvatan utisak da li je i koliko ovakav pristup prevođenja primenljiv u praksi, naveli smo nekoliko primera poređenja performansi izvršavanja programa, prevedenih Native Image-om, i nekim drugim relevantnim programskim prevodiocima.

Primer 5.1 *U tabeli 1 dato je poređenje: vremena izvršavanja programa, broja instrukcija i maksimalnog memorijskog zauzeća na primeru programa "Hello world". Prema rezultatima navedenim u tabeli, performanse Native Image-a daleko nadmašuju performanse JDK-a, dok su u poređenju sa programskim jezikom C, performanse neznatno lošije [3].*

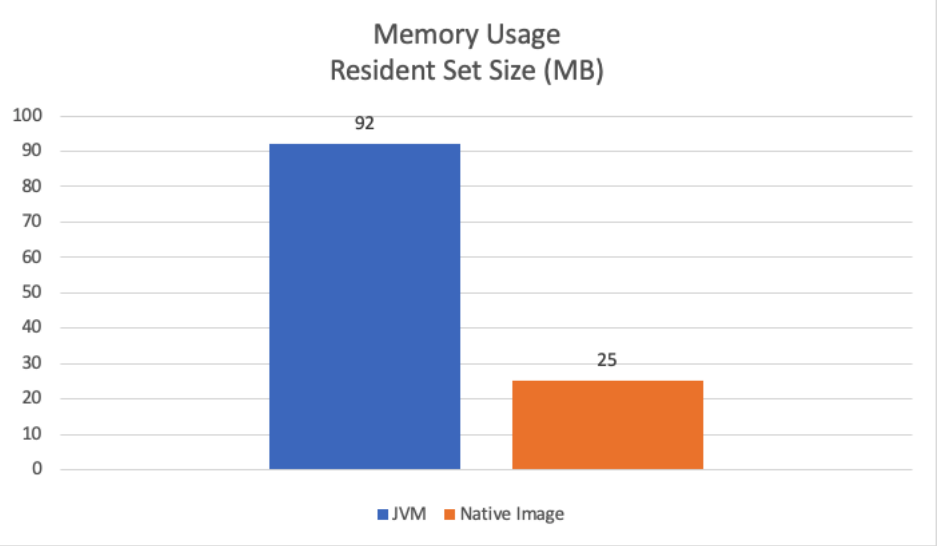
	Vreme izvršavanja [ms]	**Instrukcije korisničkog koda [milioni]**	**Instrukcije (ukupno) [milioni]**	**Maksimalno zauzeće memorije [MB]**
C	0.9	0.09	0.43	1.10
Go	1.9	0.47	1.23	7.29
JDK 8, Java HotSpot VM	103.9	152.45	171.41	37.50
JDK 12, Java HotSpot VM	74.8	119.13	144.09	52.26
JDK 12, Java HotSpot VM with AOT	89.1	119.60	147.82	69.21
GraalVM Native Image	1.9	0.53	1.98	3.37

Tabela 1: Poređenje statističkih podataka prilikom izvršavanja programa "Hello World".

Primer 5.2 *Jedna od vodećih svetskih kompanija u sferi Cloud-a, Amazon, testirala [7] je proizvoljnu aplikaciju na svojoj platformi, koristeći JVM i GraalVM Native Image. Dobijeni rezultati prikazani su na slikama 4 i 5. Kako grafici pokazuju GraalVM Native Image ima daleko manje vreme pokretanja i manje zauzeće memorije u odnosu na aplikaciju pokrenutu Java virtuelnom mašinom.*



Slika 4: Poređenje vremena pokretanja aplikacije izvršene JVM-om i GraalVM Native Image-om



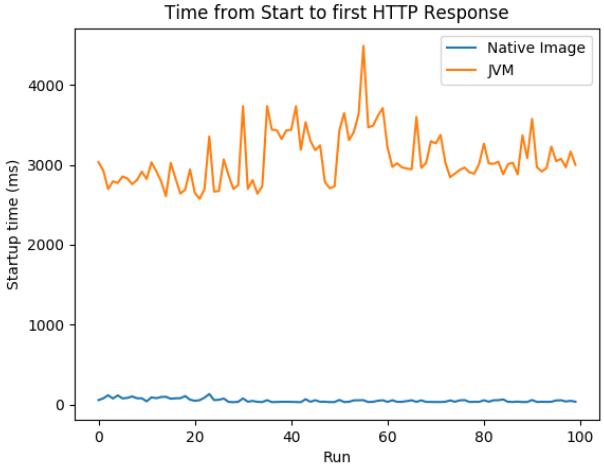
Slika 5: Poređenje memorijskog zauzeća aplikacije izvršene JVM-om i GraalVM Native Image-om

Primer 5.3 *Navešćemo i sledeći primer: Pokrenuta je serverska aplikacija pomoću klasičnog JVM-a i Native Image-a. Kako bi se performanse*

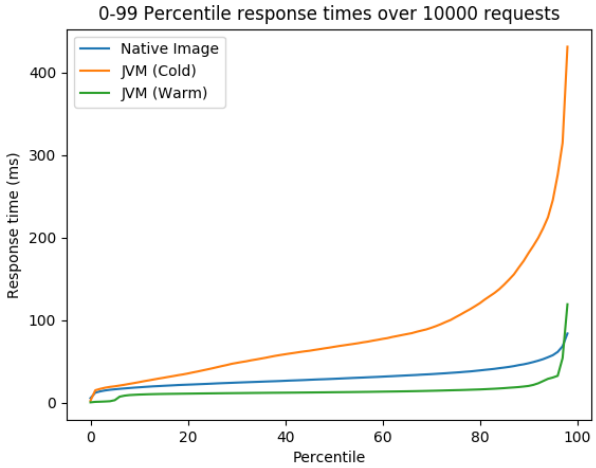
uporedile, mereni su sledeći parametri:

- *Vreme proteklo od pokretanja aplikacije do prvog odgovora servera na HTTP zahtev (slika 6)*
- *Vreme neophodno da server da odgovor na HTTP zahtev kada je pod velikim opterećenjem (slika 7)*

I na ovom primeru vidimo da Native Image daje daleko bolje rezultate od standardnog JVM pristupa.



Slika 6: Vreme proteklo od pokretanja aplikacije do prvog odgovora servera na HTTP zahtev



Slika 7: Vreme neophodno da server da odgovor na HTTP zahtev kada je pod velikim opterećenjem

6 Zaključak

Prateći trendove i pravac kojim se razvija moderno računarstvo, kao i ekonomske potrebe kompanija, Native Image je, krenuvši od istraživačkog projekta, došao do tačke gde se očekuje da njegova primena dotakne veliki broj krajnjih korisnika. Primarnu ulogu, imaće u svetu Cloud sistema, zbog značajno boljih performansi u odnosu na druge načine prevođenja aplikacija. Čist pokazatelj napretka u razvoju ove tehnologije govori i to da veliki broj pružalaca Cloud usluga nudi svojim korisnicima da koriste Native Image za izvršavanje svojig aplikacija.

Pored toga, Native Image u sklopu GraalVM-a dolazi u obliku Enterprise (plaćena verzija) i Community verzije. Upravo pristup da se određeni kod podeli sa zajednicom, omogućava Native Image-u da njegovom unapređenju doprinesu i njegovi krajnji korisnici a ne samo programeri zaposleni na ovom projektu.

Literatura

[1] https://www.infoq.com/articles/native-java-graalvm/

[2] https://docs.oracle.com/en/graalvm/enterprise/22/docs/overview/architecture/

[3] Christian Wimmer,Codrut Stancu, Peter Hofer, Vojin Jovanović, Paul Wogerer, Peter B.Kessler, Oleg Pliss, Thomas Wurthinger. *Initialize Once, Start Fast: Application Initialization at Build Time*, 2019.

[4] https://engineering.purdue.edu/Cetus/Documentation/manual/ch07s05.html

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

[6] https://build-native-java-apps.cc/developer-guide/substitution/

[7] https://aws.amazon.com/blogs/developer/graalvm-native-image-support-in-the-aws-sdk-for-java-2-x/

[8] https://www.inner-product.com/posts/benchmarking-graalvm-native-image/