Profajleri za programe napisane u programskom jeziku Java

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

Jelena Živović, Tomislav Janković, Jelena Jeremić, Milica Marić
jzivovic96@gmail.com, tomislavjankovic94@gmail.com, jjeremic597@yahoo.com,
nina.mari29@gmail.com

1. april 2020.

Sažetak

Java se u poslednjoj deceniji smatra jednim od najpopularnijih programskih jezika u svetu. Jedan od razloga tome je što njena virtuelna mašina oslobađa programera od direktnog upravljanja memorijom, što nije bio slučaj sa njenim prethodnicima poput jezika C i C++. Ovime se programeru omogućava da pažnju posveti bitnijim stvarima, poput pisanja razumnijeg koda[1]. Međutim, posledica ovog oslobađanja je da su performanse Java programa često slabije od programa pisanih na gore navedenim programskim jezicima i da se neke greške, koje nastaju pogotovo tokom izvršavanja, teže uočavaju. Jedna od tehnika pomoću koje možemo lakše uočiti i ispraviti ovakve previde u našem radu je **profajliranje**. U ovom radu smo se dotakli osnovnih karakteristika profajliranja, malo detaljnijeg opisa procesa i načina profajliranja programa pisanih u Javi. Na kraju smo naveli i koje sve vrste profajlera za programe pisane u Javi postoje i koji su njihovi najpoznatiji predstavnici.

Sadržaj

[1] Postoji izreka u svetu programiranja koja kaže da uvek treba osobu, koja bude posle nas održavala kod koji smo mi pisali, da zamišljamo kao psihopatu koja zna gde živimo.

1 Uvod

Analiza koda je veoma važna stavka u razvoju softvera koja predstavlja proces dobijanja informacija o programu na osnovu njegovog izvornog koda.[1] Deli se na statičku i dinamičku. Statička analiza predstavlja jedan skup tehnika otkrivanja mana izvornog koda programa bez potrebe da se program izvrši i nije zasnovana na optimizaciji vremenske/memorijske složenosti, već i na refaktorisanju. Dinamička analiza programa predstavlja metode prikupljanja podataka o programu tokom njegovog izvršavanja, kao i utvrđivanje ponašanja programa na osnovu dobijenih podataka.[9] Neki od predstavnika ovog vida analize su: debagovanje, profajliranje i testiranje. Jedna od bitnijih razlika između njih je što prve dve navedene tehnike su u obavezi da sprovode samo programeri, a testiranje nisu. Kako je tema ovog rada profajliranje i njegove tehnike, ohrabrujemo čitaoca da sam istražuje ostale vidove analize. **Profajliranje** je vid dinamičke analize programa čiji je cilj da pomogne programerima da pronađu kritične tačke u programu koje smanjuju njegove performanse, u cilju poboljšanja memorijske i vremenske efikasnosti programa. Često, u praksi, optimizacija malog dela koda ima pozitivan uticaj na poboljšanje performansi celog programa.[2][2].Alati koji sprovode ovaj vid dinamičke analize se zovu **profajleri**. Pored već gore navedenih razloga za korišćenje profajlera, bitno je napomenuti da ono pomaže programerima da bolje upoznaju način na koji se program izvršava, pošto nisu uvek u mogućnosti da spoznaju šta se dešava iza metoda koji se pozivaju i objekata koji se instanciraju, već im je samo bitno da taj metod i taj objekat rade ono što se od njih očekuje. Takav je slučaj sa programiranjem u Javi, gde je programer oslobođen od direktnog upravljanja memorijom zbog prisustva virtuelne mašine (JVM[3].). Dakle, dinamička analiza nam daje detaljan uvid u način izvršavanja programa, a samim tim i koliko se efikasno izvršava. Vremenom, kako je napredovao razvoj programiranja, pojavila se potreba za više vrsta tehnika, a samim tim i alata za profajliranje. Bitno je upoznati se sa svakom vrstom tehnike jer svaka ima svoj razlog i trenutak za korišćenje. U nastavku ovog teksta ćemo ukratko objasniti šta je to profajliranje, detaljnije opisati čime se ono bavi, koje vrste tehnika i alata postoje. Pojasnićemo na čemu su te tehnike i alati zasnovani u Javi, na koji način sprovode analize, koje su im prednosti i mane kao i koji su najznačajniji predstavnici tih vrsta.

[2] ,,Uglavnom 20% nekog programa radi 80% celokupnog posla.” je primenjena definicija Paretovog principa
[3] JVM (eng. *Java Virtual Machine*) je skraćenica za Java virtuelnu mašinu

2 Profajliranje

2.1 Osnovni pojmovi i faze

Rekli smo već na početku da je profajliranje namenjeno ispitivanju ponašanja efikasnosti programa, pri čemu se koriste informacije o programskom kodu koje su prikupljene tokom izvršavanja programa. Podaci dobijeni profajliranjem predstavljaju **profil programa**.[13] Jedna od najbitnijih uloga profajlera je da pruže podatke o tome koliko puta je koja funkcija (ili blok koda) pozvana tokom nekog konkretnog izvršavanja, koliko je potrošila vremena i memorije na hipu i slično.[8] Profajliranje se može podeiliti po vrstama na: samo softversko, samo hardversko i kao kombinacija prethodna dva. Hardverska podrška omogućava veću efikasnost samog profajliranja kao i veći opseg podataka dobijenih na osnovu profajliranja.[5] Profajliranje kao sam proces možemo podeliti na sledeće tri faze:

- instrumentalizacija (eng. *instrumentation*)
- prikupljanje podataka (eng. *data collecting*)
- obrada podataka (eng. *data managing*)[9]

Proces tokom koga profajler modifikuje analizirani program, ubacujući nove instrukcije u njega, naziva se **instrumentalizacija**. Instrukcije se obično dodaju u specifičnim tačkama analiziranog koda koje se nazivaju **merne tačke** (eng. *measure points*) i one predstavljaju kod za inicijalizaciju određenih dodatnih struktura i pravila za njihovo popunjavanje. Te strukture predstavljaju skladište za metapodatke (podaci o podacima) dok je za popunjavanje zadužen instrumentalizovani program. Instrumentalizacija se može vršiti manuelno, od strane programera (dodavanjem linija u kodu) ili automatski u različitim fazama. Programer može dodati jednostavne komande za ispis, ali i složene strukture za praćenje stanja: promenljivih, funkcija, grananja ili petlji. Automatsku instrumentalizaciju može da vrši prevodilac i/ili linker, u toku izvršavanja programa ili u toku prevođenja.[5]

2.2 Osobine i vrste profajliranja

Da bi rezultat profajliranja bio kvalitetan, profajler mora zadovoljiti sledeće osobine:

1. Da prikuplja samo potrebne podatke – zahtev za previše podataka usporava program kao i obradu tih podataka, s druge strane premalo informacija može biti beznačajno
2. Da ne utiče na funkcionalnost programa – ako instrumentalizacija utiče na funkcionalnost programa, prikupljeni podaci neće precizno oslikavati način njegovog rada
3. Da ne usporava previše rad programa – zavisi od tipa aplikacije i možemo ga kontrolisati u zavisnosti od delova programa koji se instrumentalizuju

Na osnovu mesta gde se ubacuje dodatni kod, profajliranje se može podeliti na:

- Profajliranje putanje (eng. *path profiling*)[4]
- Profajliranje ivica (eng. *edge profiling*)
- Profajliranje blokova (eng. *basic-block profiling*)[5]

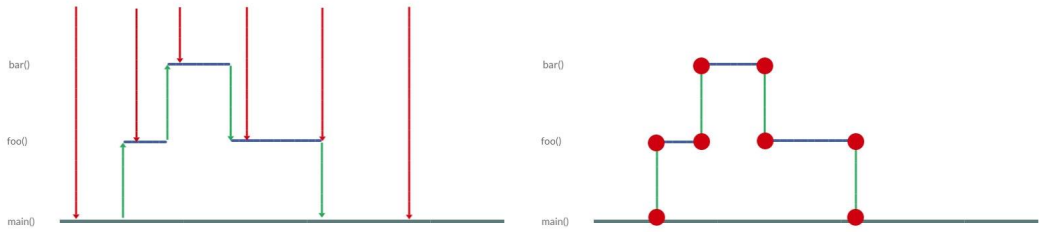
Profajliranjem blokova/ivica se broji ukupan broj njihovog izvršavanja, na osnovu brojača blokova/ivica.

[4]Služi za dobijanje informacija o najčešće korišćenim putanjama u programu. Ovakav profil sadrži informacije o profilima blokova i ivica, poboljšava performanse tokom izvršavanja programa, proverava pokrivenost koda testom, itd.

[5]Blok se odnosi na funkciju ili deo koda u kome se ne nalaze funkcije grananja ili skokova, dok ivica povezuje dva bloka i predstavlja instrukciju grananja ili skoka kojom se prebacuje tok izvršavanja programa iz jednog bloka u drugi.

2.3 Krajnji rezultat i osnovni problemi

U prvim koracima profajliranja dobijaju se kvalitetni rezultati, no kako optimizacija napreduje postaje sve teže poboljšavati program. **Uzimanje uzoraka (uzorkovanje)** je jedan od efikasnih načina kojim se može smanjiti usporavanje programa procesom instrumentalizacije. Posmatra se deo programa koji se izvršava i pravi profil programa od ***slika*** (eng. *snapshots*) uzetih u ručno zadatim vremenskim intervalima. Rezultat je dosta manje opterećenje programa, ali je posledica smanjena ažurnost dobijenih podataka. Ukoliko programer loše rasporedi vremenske intervale, može doći do potencijalnog propuštanja bitnog događaja koji nam ukazuje na neoptimizovanost našeg softvera.[13] Na slici 1 dat je primer ovog problema. Levo je prikazano profajliranje tehnikom uzorkovanja, gde strelice ukazuju na vremenske intervale kada su napravljene slike programa koji se izvršava. Desna slika nam prikazuje kako instrumentalizacija prati stanje programa za sve vreme njegovog izvršavanja. Možemo videti da se neki bitni događaji mogu propustiti uzorkovanjem jer ono prati stanje programa u samo određenim vremenskim intervalima. **Dakle, tehnika uzorkovanja ne određuje način instrumentalizacije već samo kako smanjiti njene troškove.**



Slika 1: Mesta u programu u kojima se prikupljaju informacije o njegovom izvršavanju kod uzorkovanja(levo) i instrumentalizacije(desno)

U nekim situacijama ne možemo lako koristiti metode za instrumentalizaciju. Primer koji možemo uzeti za to su sistemi u realnom vremenu koji imaju vremenska ograničenja koja se profajliranjem mogu prekršiti zbog prekomernih bespotrebnih troškova[6] koje alat pravi dodatno tokom analize. Instrumentalizacija može povećati broj linija mašinskog koda, te je moguće uvećati kod toliko da on prekorači memorijski opseg uređaja.[13] Prikupljanje podataka obuhvata mogućnost čitanja struktura sa metapodacima, njihovo predstavljanje u pogodnijem obliku i mogućnost eksternog skladištenja u datotekama. Pogodan oblik podrazumeva dobar odnos između obaveštajnosti (eng. *informativity*), čiji koeficijent mora biti što veći i veličine, koja mora biti što manja. Uklanjaju se oni podaci koji se mogu izvesti iz drugih. Proizvod prve dve faze su informacije o određenim karakteristikama programa sa konkretnim ulazima, koje se sastoje od sirovih podataka. Treća i poslednja faza je obrada sirovih podataka do korisnih informacija. Krajnji proizvod profajliranja jeste jedan ili više izveštaja (eng. *report*) koji su u formatu čitljivom prvenstveno za razvojni tim, a ne za računar. Oni se razlikuju u zavisnosti od karakteristika koje se mere i od potreba korisnika. Izveštaj može biti jedna rečenica, kolekcija fajlova pa i cela interaktivna aplikacija.[9] U tabeli 1 možemo videti primer krajnjeg rezultata procesa profajliranja tj. profil jednog programa.

[6] Troškovi se odnose na sve dodatne resurse koji su potrebni našem softveru da se nešto izvrši, ali nisu direktno vezani za izračunavanje rezultata našeg programa.

U slučajevima kada instrumentalizacija nije dobro rešenje, možemo izabrati profajliranje uzorkovanjem. Kod ove vrste profajliranja postepeno se izvršavaju originalni i instrumentalizovani kod. Intervali izvršavanja oba koda su veoma jasno definisani. Sam algoritam prepoznaje tri verzije koda programa:

- Originalni kod
- Proveravajući kod – veoma liči na originalni kod, ali ima dodate instrukcije kojima se proverava uslov prelaska u instrumentalizovani kod. Te instrukcije nemaju značajan uticaj na performanse programa. U ovaj kod se na određenim mestima dodaju sekcije (eng. *checks*) koji proveravaju uslove prelaska. Ako je uslov prelaska ispunjen, izvršava se instrumentalizovani kod
- Instrumentalizovani kod – kontrola se na određenim mestima, koja ne zadovoljavaju određena svojstva, vraća na stanje proveravajućeg koda[7]

[13]

Klasa	Metoda	Broj poziva	Vreme utrošeno po pozivu (ms)	% iskorišćenosti CPU-a
EventQueue	nextEvent	10	10.08	20
Timer	timedEvent	45	20.86	40
Item	vetoMove	4	8.9	5
Animate	vetoMove	3	7.5	6
Food	firstEvent	5	4.7	7
Cabbage	secondEvent	3	7.4	5
Player	jumpForward	5	6.3	4
Ogre	jumpBackward	6	7.4	6
Gameobject	vetoMove	4	6.4	4
Room	replaceMove	6	6.4	7

Tabela 1: Primer profila CPU intenzivne igre

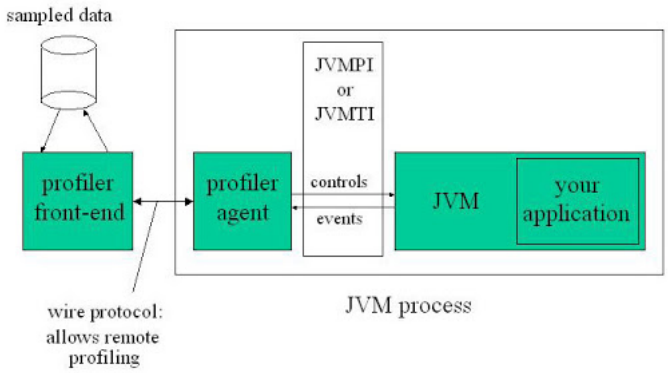
[7] U precizno određenim uzorcima program izvršava instrumentalizovane blokove i tako doprinosi formiranju profila, ali se najveći deo vremena ipak izvršava proveravajući kod. Neophodno je precizno odrediti odnos između vremena izvršavanja programa unutar proveravajućeg i instrumentalizovanog koda da bi smanjili usporenje programa, a pritom dobili dovoljno precizne podatke.

3 Specifičnosti Java profajliranja

Java profajler je alat koji posmatra segmente koda i operacije Java bajtkoda na nivou JVM[3]. U ove segmente koda i operacije spadaju: instanciranje objekta, izračunavanje iteracija, rekurzivni pozivi, pozivi metoda, izvršavanje niti i sakupljanje otpada[8]. Uglavnom se može instalirati zasebno ili kao dodatak (eng. *plug-in*) već poznatom razvojnom okruženju (npr. Intellij i Eclipse imaju svoje profajlere)[3][12].

3.1 Profajliranje Java programa

Arhitektura debagera Java platforme (eng. *Java Platform Debugger Architecture* - JPDA) predstavlja kolekciju aplikacionih programskih interfejsa (eng. *Aplication Programming Interface* - API) za debagovanje Java koda. Interfejs JVM alata (eng. *JVM Tools Interface* - JVMTI) i interfejs JVM profajlera (eng. *JVM Profiler Interface* - JVMPI) predstavljaju najniži nivo ove platforme[12]. JVMTI je predstavljen u okviru platforme Java SE 5.0 i u potpunosti je smenio do tad korišćen JVMPI, koji je kasnije i uklonjen u verziji Java SE6. Java profajleri ispituju stanje JVM-a na dva načina: pasivno, osluškujući događaje koje JVM generiše (eng. *listener*) i aktivno, ispitivanjem JVM-a o internom stanju tako što profajler otvori soket ili neki drugi međuprocesni komunikacioni kanal[10] [9] koji ga povezuje sa Java aplikacijom čije se performanse ispituju i onda aplikacija i profajler kroz taj kanal razmenjuju informacije. Primer osnovne arhitekture Java profajlera dat je na slici 2.



Slika 2: Osnovna arhitektura Java profajlera[7]

JVM može da generiše događaje koji se grubo mogu podeliti u dve grupe: **trenutni** (eng. *instant*) događaji i **trajni** (eng. *duration*). Trenutni događaji su oni koji sadrže vremenski žig i podatke o samom događaju, kao što su izuzeci, učitavanje klasa i instanciranje objekata. Daju informaciju o tome da se nešto desilo i profajleri mogu ili da reaguju na njih ili da ih samo posmatraju i analiziraju. Trajni događaji imaju početno i krajnje vreme izvršavanja, a time i mogućnost merenja vremena aktivnosti (npr. informacija o početku i kraju sakupljanja otpadaka itd). JVM ima i neke metode profajliranja koje vraćaju informacije o internom stanju, kao sto su *getThreadState* i *getAllThreads* ili *getStackTrace* i

[8]Definicija preuzeta sa https://www.baeldung.com/java-profilers i https://dzone.com/articles/top-9-free-java-process-monitoring-tools-amp-how-t.

[9]Međuprocesni komunikacioni kanali (eng. *Inter-process communication* - IPC) su određeni mehanizmi koji omogućavaju procesima da razmenjuju podatke između sebe. Primer su: soketi (eng. *sockets*), deljena memorija (eng. *shared memory*), pajp (eng. *pipe*) i red za poruke (eng. *message queue*).

getAllStackTraces[11][10].
Profajliranje se može sprovesti u različite svrhe i to:

- procesorsko (CPU) profajliranje
- profajliranje memorije
- profajliranje niti

CPU profajliranje se koristi u svrhu određivanja koliko aplikacija ima uticaj na procesor. Moguće je prepoznati metode i niti u programu koje troše najviše procesorskog vremena. **Profajliranje memorije** nam daje mogućnost da uvidimo kako naša aplikacija koristi hip memoriju, odnosno kako je objekti koriste i kako sakupljač otpadaka (eng. *garbage collector*) oslobađa memoriju. Na osnovu ovoga, možemo pronaći potencijalna curenja memorije, neefikasno korisćenje hip memorije, itd. **Profajliranje niti** omogućava da vidimo u kojim stanjima su trenutno niti i zašto. To je veoma korisno jer stalno imamo uvid u stanje paralelnosti, koliko vremena provode blokirane (samim tim i da otkrijemo potencijalno stanje deadlock-a ili livelock-a, koja su veoma pogubna), u stanju čekanja ili izvršavanja[11].

3.2 Performanse profajlera Java programa

Kao što smo gore naveli, profajliranje Java programa se vrši tako što profajleri posmatraju izvršavanje JVM-a na nivou bajtkoda i prikupljaju informacije o izvršavanju niti, upotrebi hip memorije, sakupljaču otpadaka, pozivu metoda, ispaljivanju izuzetaka, itd. Realizovani su kao **agenti** koji se, pomoću odgovarajućeg API-ja, povezuju na JVM. Zasnovani su na događajima, što znači da koriste brojače događaja hardvera kako bi dobili broj specifičnih događaja koji se dese tokom izvršavanja programa. Profajleri su napisani u C-u ili C++-u[12], a ne u jeziku same platforme, što može predstavljati poteškoću, pošto se od programera zahteva da poznaje dodatan programski jezik[12][3]. Interfejs nam nudi širok izbor alata sa kojima možemo ispitivati stanja programa ili kontrolisati njegovo izvršavanje. Dvosmeran je, tako da jedan smer koristi da agentu prosleđuje obaveštenja o događajima koji se dešavaju tokom izvršavanja programa (npr. o alokaciji hip memorije, o aktivnosti sakupljača otpadaka, ulasku u metodu, izlasku iz metode, itd.). Drugi smer, agent, koristeći funkcije interfejsa, šalje zahteve JVM-u o značajnim podacima. Zahtevi mogu biti u vidu kontrola (npr. zahtev za uključivanje ili isključivanje notifikacija o nekom događaju). Svi ovi zahtevi se formiraju u okviru prednjeg dela profajlera (eng. *front- end*)[14]. JVM i agent su pokrenuti u okviru istog procesa. Agent je zadužen da vrši komunikaciju sa prednjim delom profajlera. Prednji deo profajlera šalje zahteve koje će agent dalje proslediti JVM-u, ali i prima izveštaje koje mu agent šalje nazad kao odgovor. On se uglavnom pokreće u okviru posebnog procesa ili čak na nekoj drugoj mašini. Ovime sprečavamo da prednji deo profajlera utiče na performanse programa koji ispitujemo tj. obezbeđujemo da podaci o performansama posmatranog programa ne uključuju i performanse profajlera koji se izvršava uporedo sa njim.

Statičko uzorkovanje nema uticaj u velikoj meri na performanse programa, ali ne pruža celokupne i precizne informacije o izvršavanju. Sa druge strane, instrumentalizacija koda, iako ona može dosta uticati na performanse programa zbog troškova, dozvoljava da profajler dobije sve informacije o izvršavanju koje su mu potrebne (npr. statičkim uzorkovanjem možemo dobiti procenat vremena utrošenog na često pozivane metode, dok instrumentalizacijom možemo dobiti tačno koliko puta je svaka metoda pozvana)[15]. Alati koji vrše prikupljanje podataka uzorkovanjem se periodično aktiviraju i očitavaju potrebne parametre. Sa obzirom da alat nije stalno aktivan, smanjeno je opterećenje na praćeni sistem, ali zato postoji mogućnost da se propusti neki događaj između dva očitavanja. Prednost ovog

[10] Više o ovim metodama možete pogledati u zvaničnoj dokumentaciji Oracle-a

[11] Detaljnije možete pročitati ovde: https://docs.oracle.com/middleware/1213/jdev/user-guide/jdev-test-profile-java.htm#OJDUG6710.

[12] Ovi jezici su upravo i izabrani zbog niskog nivoa pristupa (programer je u mogućnosti da direktno upravlja korišćenjem radne memorije) i brzine

pristupa je što posmatrani program može da radi skoro punom brzinom, čime se dobija najmanja greška merenja. Nedostatak ovog pristupa je što je obično zasnovan na tehnikama koje su veoma zavisne od platforme na kojoj se izvršavaju jer se koriste prekidi, sistemski sat, raspoređivač (eng. *scheduler*) ili programski brojač na platformi. Iako instrumentalizacija može da izazove ozbiljne poremećaje u performansama, uz dobro praćenje i kontrolu metoda može obezbediti dobre rezultate. Greška merenja može i da se izmeri, pa da se isključi iz rezultata. Još jedan nedostatak pristupa je taj što se programski kod "prlja"kodom koji se koristi za praćenje, što otežava kasnije održavanje.

4 Profajleri Java programskog jezika

Kao što smo se gore pozabavili temom profajliranja i osnovama profajliranja Java programa, u nastavku ćemo objasniti koje sve vrste profajlera postoje, koja im je namena i koji su najbitniji predstavnici i njihove prednosti i mane. Postoje tri vrste profajlera[4]:

- **Standardni JVM profajleri**
- **Profajleri ,,laganog" tipa**[13]
- **APM alati**[14]

Svaka vrsta profajlera, od navedenih, ima svoje prednosti i mane. Cilj programera je da minimizuje mane svakog, da bi mogao sa lakoćom da pronađe što više propusta i optimizuje neefikasne delove u svojem kodu. Zato se programeri ohrabruju da prilikom testiranja efikasnosti svoje aplikacije koriste svaku od navedenih vrsta, da bi upotpunili svoje znanje. Na slici 2 dati su primeri poznatijih profajlera u Javi sa svojim specifikacijama. Prikazane su i korisničke ocene lakoće instalacije, upotrebljivosti i pouzdanosti svakog prikazanog profajlera.

4.1 Standardni JVM profajleri

Ova vrsta profajlera prati resurse koje JVM koristi, kao što su procesor, niti, radna memorija, sakupljač otpadaka, keš memoriju, itd. Ovi podaci jesu sveobuhvatni, ali većina, u zavisnosti od zadatka, nije toliko korisna. Ono što jeste stvarno korisno kod njih je to što prate sve pozive funkcija i pristup (i korišćenje) radne memorije, čime možemo sa lakoćom da utvrdimo curenje memorije, neefikasno zamenjivanje keš stranica sa memorijom, itd. Takođe, prateći zauzetost procesora, možemo utvrditi koji instancirani objekti ili metodi ,,izgladnjuju" procesor. Veliki problem nastaje u tome što oni moraju paziti na resurse koje JVM koristi, čime veoma usporavaju okruženje, a samim tim i aplikaciju koja se testira, zato što procesor te informacije profajlera mora da obradi i isporuči programeru. Pritom, neki programi za profajliranje koriste niti i isporučuju stanje memorije u mnogo manje učestalim trenucima. Najpoznatiji profajleri ove vrste su: *Visual VM i JDK Mission Control* (Oracle), *JProfiler* (GmbH), *Your Kit* (Your Kit)[4].

4.2 Java profajleri ,,laganog" tipa

Za razliku od opštih JVM profajlera, ova grupa meri izvršavanje aplikacije tako što *,,se ubacuje"* u kod. Postoje dva podtipa:

- Aspektni profajleri (eng. *Aspect profilers*)
- Profajleri Java agenta (eng. *Java Agent Profilers* - JAP)

Prvi podtip, zasnovan na **AOP paradigmi**[15] (eng. *Aspect Oriented Programming*), se ubacuje u naš kod na početku i na kraju izabranih metoda. Taj ubačeni kod pokreće štopericu i izračunava vreme izvršavanja metoda. Prednost kod ovoga je u tome što se veoma trivijalno podešava i nije toliko memorijski zahtevno kao prethodna grupa. Mana je, pak, što moramo da znamo šta želimo da ispitujemo i mogu se opteretiti analizom. Sledeći podtip koristi **API Instrumentalizacije** (eng. *Instrumetalisation API*) da se ubaci u naš kod. Sa ovim API-jem je omogućen bolji pristup našem softveru jer se kod upisuje na nivou bajtkoda[6]. Ovo znači da je bilo koji kod koji se izvršava, podložan instrumentalizaciji. Za razliku od aspektnih, dubina steka pretrage koju mogu postići Java agenti je mnogo veća, ali su mnogo komplikovaniji za pisanje. Primeri ovih tipova su: *XRebel*, *Prefix* (Stackify)[4].

[13] Kad se misli na pridev lagano (eng. *lightweight*) u softverskom smislu, uglavnom se misli aplikacije koje su napisane tako da mnogo manje opterećuju hardver od drugih verzija

[14] APM je skraćenica za upravljanje performansama aplikacije (eng. *Application performance management*)

[15] Aspektno-orijentisano prograiranje je paradigma koja omogućava ubacivanje dodatnog koda na već postojeći

4.3 APM alati

Kako je bitno, tokom razvoja softvera, profajlerima analizirati optimizovanost našeg koda, tako je potrebno to raditi i *„u produkciji"* i tu nam pomaže ova grupa alata. Puštanje softvera u produkciju je dijametralno drugačije od njegovog razvoja i potrebni su nam alati koji bi mogli da analiziraju softver, bez da narušavaju performanse softvera. Ovo se postiže sa APM alatima, koji su zasnovani na profajlerima Java agenata, ali sa drugačijim parametrima instrumentalizacije, tako da neće preterano zauzimati cikluse časovnika procesora.Primeri ove grupe su *New Relic*, *Retrace*, *Dynatrace*[4].

Profajler	Kompanija	Cena	Instalacija	Upotrebljivost	Pouzdanost	Hip	CPU	SQL	IO	Niti	Brojač poziva
HPROF	Oracle	Besplatan	5	2	2	Da	Da	Ne	Ne	Da	Da
NetBeans	Oracle	Besplatan	5	4	4	Da	Da	Da	Ne	Da	Da
VisualVM	Oracle	Besplatan	5	4	4	Da	Da	Ne	Ne	Da	Da
JMC&JFR	Oracle	Besplatan	5	5	4	Da	Da	Ne	Da	Da	Da
JMH	OpenJDK	Besplatan	4	3	5	Ne	Da	Ne	Ne	Ne	Ne
XRebel	ZeroTurnaround	$365/god	3	4	4	Da	Da	Da	Ne	Da	Da
JProfiler	EJ Technologies	$409-$599/god	5	4	4	Da	Da	Da	Ne	Da	Da
JMeter	Apache	Besplatan	5	4	5	Da	Da	Ne	Ne	Da	Da
YourKit	YourKit	$499/god	4	3	3	Da	Da	Da	Ne	Da	Da

Tabela 2: Primeri nekih od postojećih profajlera, njihove ocene i specifikacije

5 Zaključak

Cilj ovog rada je bio da objasni šta je profajliranje i profajleri, zašto je važno, kada se ono koristi, kakve sve vrste i tipovi postoje, koji su najčešći problemi sa kojima se možemo susresti prilikom istog i koji su najpoznatiji i najkorišćeniji predstavnici u programskom jeziku Java. Prilikom međusobnog poređenja, primetili smo da sve vrste profajlera imaju svoje prednosti i mane, koje zavise i od toga u kojem trenutku razvoja softvera se određene vrste koriste. Isto tako, veoma je bitno napomenuti da **dinamička analiza nije zamena za statičku**. Obe su podjednako važne, (možemo dosta da uštedimo na vremenu ako govorimo o malim, *,,okom odstranljivim"* problemima) pošto određeni tipovi profajlera (npr. Standardni JVM profajleri koji izračunavaju maltene sve interakcije našeg programa sa memorijom, procesorom, operativnim sistemom i IO jedinicama, pritom trošeći procesorsko vreme[4]) su veoma vremenski zahtevni. Primera radi, ako imamo softver koji računa algoritmom A* rastojanje najkraćeg puta za dostavu hrane od picerije do korisnika i ako smo koristili jednostruko povezanu listu umesto binarnog mini-hipa za otvorene čvorove, jednostavnije je primetiti taj propust okom i promeniti, nego gubiti vreme sa nekom vrstom profajlera da utvrdimo zašto nam softver neefikasno radi u realnom vremenu. Tako da, pored ogromne moći koju nam profajleri nude, ne bi trebalo da zaboravimo na *,,programersku pismenost"*[16] koja se stiče vežbom i učenjem.

Optimizacija softvera je od ogromnog značaja u svetu programiranja i može ozbiljno da ugrozi profit sa preteranim korišćenjem, prevremenim[17] korišćenjem ili pak čistim izbegavanjem. Zato je važno shvatiti profajliranje veoma ozbiljno. Bez pažljivog pristupa može doći do posledica katastrofalnih razmera, od kojih je najblaža da softver bude nečitljiv zbog ubačenog optimizovanog koda ili da prekine sa radom (prekoračenjem memorijskog opsega zbog preteranog ubacivanja optimizovanog koda ili samo zbog nekih promaknutih neoptimizovanih delova). To može dovesti do ogromnih novčanih gubitaka, ali i ljudskih života[18]. Upravo je i to razlog zašto ima toliko mnogo vrsta profajlera, u zavisnosti o kojoj razvojnoj grani razvoja softvera govorimo, kao i zašto su oni veoma skupi. Ta cena se dvostruko, pa možda i trostruko uvećava kad govorimo o profajlerima koji rade nad *,,softverima u produkciji"*. Primera radi, JProfiler košta 2500$ (oko 270 000 dinara) na godišnjem nivou, dok se Xrebel plaća 365$ na godišnjem nivou (oko 40 000 dinara). Postoje i besplatni APM profajleri kao što su Glowroot i Scouter, za koje je važno napomenuti da su takođe i otvorenog koda[4][6]. Ideja ovog rada je bila da vas upozna i sa određenim predstavnicima i da vam ponudi jeftinije alternative da bi se mogli upoznati sa radom sa kojim ćete se sigurno sretati i dalje u praksi.

[16] Misli se na upućenost u pisanje koda po određenim standardima navedenim u dokumentaciji, bilo programskog jezika, bilo u poslovnoj

[17] *,,Prevremena optimizacija je početak svakog zla"*, Donald Knut iz papira *Stuctured Programming With Goto Statements*

[18] Možete više o ovome pogledati ovde: https://www.computerworld.com/article/3412197/top-software-failures-in-recent-history.html.

Literatura

[1] David Binkley. Source Code Analysis: A Road Map. *Future of Software Engineering (FOSE '07)*, pages 2–5, 2007.

[2] Tannaz Alinaghi Camellia Ghoroghi. An introduction to profiling mechanisms and linux profilers.

[3] Zora Konjović Dušan Okanović, Milan Vidaković. Dokumentacija tehničkog rešenja - sistem za adaptivno praćenje performanski u distribuiranim softverskim sistemima, 2014.

[4] Darin Howard. Java profilers: Why you need these 3 different types. https://stackify.com/java-profilers-3-types/, 2019.

[5] Milena Vujošević Janičić. Verifikacija softvera. on-line at: http://www.verifikacijasoftvera.matf.bg.ac.rs/vs/predavanja/03_dinamicka_analiza/03_dinamicka_analiza_slajdovi.pdf.

[6] Eyal Katz. Top 9 free java process monitoring tools and how to choose one. https://dzone.com/articles/top-9-free-java-process-monitoring-tools-amp-how-t, 2019.

[7] Angelika Langer Klaus Kreft. Java performance: Profiling. 2005.

[8] Predrag Janičić Filip Marić. *Programiranje 2.* Matematički fakultet Univerziteta u Beogradu, 2020.

[9] Marina R. Nikolić. Prikupljanje i prikaz podataka o izvršavanju programa. Master's thesis, Univerzitet u Beogradu, Matematički fakultet, 2019.

[10] Scott Oaks. *Java Performance: The Definitive Guide*, chapter 3 A Java Performance Toolbox, page 51. O'Reilly, 2014.

[11] Tim Ojo. How java profilers work. https://dzone.com/articles/how-java-profilers-work, 2019.

[12] Dušan Okanović. *Model adaptivnog sistema za praćenje i predikciju rada distribuiranih aplikacija.* PhD thesis, Fakultet tehničkih nauka u Novom Sadu, 2012.

[13] Nikola B. Prica. Podrška za profajliranje softvera uređaja sa ugrađenim računarom. Master's thesis, Univerzitet u Beogradu, Matematički fakultet, 2018.

[14] Allen D. Malony Sameer Suresh Shende. Integration and application of the tau performance system in parallel java environments. 2001.

[15] Deepa Viswanathan Sheng Liang. Comprehensive profiling support in the java ^{tm} virtual machine. 1998.