UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Milan M. Čugurović

PREDVIĐANJE PROFILA IZVRŠAVANJA PROGRAMA TEHNIKAMA MAŠINSKOG UČENJA

doktorska disertacija

Beograd, 2025.

UNIVERSITY OF BELGRADE
FACULTY OF MATHEMATICS

Milan M. Čugurović

PREDICTION OF PROGRAM PROFILES USING MACHINE LEARNING TECHNIQUES

Doctoral Dissertation

Belgrade, 2025.

Mentor:

dr Milena Vujošević Janičić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Filip Marić, redovni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Mladen Nikolić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Saša Misailović, vanredni profesor
Univerzitet Ilinois u Urbana-Šempejnu
(*University of Illinois at Urbana-Champaign*)

Datum odbrane: _____

Zahvalnica

Zahvaljujem se svojoj mentorki, prof. dr Mileni Vujosević Janičić, na podršci i saradnji tokom doktorskih studija i izrade ovog rada. Takođe dugujem zahvalnost članovima komisije za pažljivo čitanje rukopisa i korisne sugestije. Veliku zahvalnost dugujem i kolegama iz kompanije Oracle Labs na saradnji tokom razvoja projekata koji su pratili izradu ove disertacije. Posebnu zahvalnost upućujem porodici i prijateljima.

Milan

Acknowledgments

I would like to thank my advisor, Prof. Milena Vujosević Janičić, for her support and collaboration throughout my doctoral studies and the preparation of this dissertation. I am also grateful to the members of the committee for their careful reading of the manuscript and valuable suggestions. I owe great thanks to my colleagues at Oracle Labs for their collaboration on the projects that accompanied the development of this dissertation. Finally, I extend my special gratitude to my family and friends.

Milan

U sećanje i sa zahvalnošću, đedu.

Naslov disertacije: Predviđanje profila izvršavanja programa tehnikama mašinskog učenja

Rezime: Kompilatori koriste profil izvršavanja programa kako bi izvodili optimizacije i kreirali efikasne programe. Iako dinamički profajleri prikupljaju profile visokog kvaliteta, oni takođe imaju i značajne nedostatke. Oni komplikuju proces izgradnje aplikacije zahtevajući dve faze kompilacije i dodatno prikupljanje profila. Dinamički profajleri takođe troše značajnu količinu vremena i memorije, i unose veliko opterećenje za programere, koji moraju da pripreme odgovarajuće test programe koji verno oslikavaju tipičan način korišćenja aplikacije, pokrivaju važne delove koda i obezbeđuju prikupljanje kvalitetnih profila.

Kako bi prevazišli nedostatke dinamičkih profajlra, moderni statički profajleri koriste tehnike mašinskog učenja za predviđanje profila izvršavanja programa. Ipak, najbolji statički profajleri zasnovani na mašinskom učenju često se oslanjaju na manuelno definisane atribute koji su specifični za platformu i koje je teško prilagoditi različitim arhitekturama i programskim jezicima. Takođe, oni često koriste računski zahtevne duboke neuronske mreže, što povećava vreme kompilacije aplikacija. Takođe, statički profajleri zasnovani na modelima mašinskog učenja mogu narušiti performanse optimizovanih programa zbog grešaka prilikom predviđanja profila.

GraalSP je statički profajler zasnovan na mašinskom učenju koji je portabilan, višejezičan, efikasan i robustan. *GraalSP* postiže portabilnost definisanjem atributa nad grafovskom međureprezentacijom visokog nivoa i delimičnom automatizacijom procesa izdvajanja atributa. Ovo takođe omogućava višejezičnost jer *GraalSP* može da predviđa profile za programe napisane u bilo kom jeziku koji se kompajlira u Java bajtkod, poput Jave, Skale ili Kotlina. *GraalSP* je efikasan zahvaljujući upotrebi efikasnog modela *XGBoost* zasnovanog na stablima odlučivanja, a robustan jer se oslanja na pažljivo dizajnirane heuristike koje koriguju predviđanja modela mašinskog učenja i obezbeđuju visoke performanse u programima optimizovanim na osnovu predviđenih profila.

Statički profajler *GraalSP* sastavni je deo kompilatora *Enterprise GraalVM Native Image*. Evaluacija na skupu od 28 programa iz skupova test programa *Renaissance*, *DaCapo* i *DaCapo con Scala* pokazuje da *GraalSP* postiže prosečno ubrzanje aplikacija od 7.46% geometrijske sredine. Ovo ubrzanje odnosi se na ubrzanje vremena izvršavanja programa u poređenju sa podrazumevanom konfiguracijom kompilatora, koja koristi uniformnu raspodelu za modelovanje profila izvršavanja programa. Korišćeni skupovi test programa predstavljaju savremenu i raznovrsnu kolekciju testova, sa brojnim modernim programima pisanim u različitim programskim paradigmama što potvrđuje kvalitet razvijenog statičkog profajlera. Disertacija prikazuje i detaljnu kvalitativna i kvantitativna analiza i pozicioniranje i upoređivanje predloženog rešenja sa do sada najboljim razvijenim statičkim profajlerima. Dodatno, kako bi se unapredila i proširila evaluacija, i kako bi se programerima omogućila bolja analiza predviđanja statičkog profajlera *GraalSP*, u okviru ove disertacije razvijen je i alat *GraalSP-PLog*. Ovaj alat omogućava programerima da pokrenu statički profajler *GraalSP* na proizvoljnim programima i generišu detaljne izveštaje o predviđanjima modela, olakšavajući identifikaciju pojedinačnih predviđanja i eventualnih grešaka modela.

S obzirom na to da *GraalSP* obezbeđuje značajna poboljšanja performansi, ima minimalan uticaj na veličinu izvršivih fajlova i vreme kompilacije programa, kao i da poseduje modernu, potpuno automatizovanu infrastrukturu za ponovno obučavanje modela, veoma je pogodan za komercijalnu primenu. Kao rezultat toga, *GraalSP* je podrazumevano omogućen u kompilatoru *Enterprise GraalVM Native Image* od juna 2023. godine i od tada konstantno unapređuje performanse programa pri svakoj kompilaciji.

Ključne reči: kompilatori, statički profajleri, mašinsko učenje, regresija, gradijentno pojačavanje, ansambli, *Enterprise GraalVM Native Image*

Naučna oblast: računarstvo

Uža naučna oblast: kompilatori, mašinsko učenje

UDK broj: 004.415.5(043.3)

Dissertation title: Prediction of program profiles using machine learning techniques

Abstract: Compilers use program profiles to perform profile-guided optimizations and produce efficient programs. Although dynamic profilers generate high-quality profiles, they have significant drawbacks. They complicate the application build pipeline by requiring two compilation steps and an additional profile collection run. Dynamic profilers also consume substantial time and memory and place a heavy burden on developers to create suitable workloads that accurately reflect typical application usage, cover important code paths, and generate well-distributed profiles.

In response to the shortcomings of dynamic profilers, modern static profilers employ machine learning (ML) techniques to predict program profiles. However, state-of-the-art ML-based static profilers often rely on handcrafted features that are platform-specific and difficult to adapt across different architectures and programming languages. They also tend to use computationally intensive deep neural networks, which increase application compilation time. Moreover, ML-based static profilers can degrade the performance of optimized programs due to inaccurate profile predictions.

This dissertation presents *GraalSP*, an ML-based static profiler that is portable, polyglot, efficient, and robust. *GraalSP* achieves portability by defining features on a high-level, graph-based intermediate representation and by partially automating the feature extraction process. This design makes *GraalSP* polyglot, allowing it to predict profiles for programs written in any language that compiles to Java bytecode, such as Java, Scala, or Kotlin. *GraalSP* is efficient due to its use of a lightweight *XGBoost* model based on decision trees, and robust because it relies on carefully designed heuristics that correct machine learning predictions and ensure high performance in programs optimized using the predicted profiles.

We integrate *GraalSP* into the *Enterprise GraalVM Native Image* compiler and evaluate it on 28 benchmarks from the *Renaissance*, *DaCapo*, and *DaCapo Scala* benchmark suites. These suites represent a modern and diverse collection of benchmarks, featuring numerous real-world workloads across a variety of programming paradigms. Our comprehensive evaluation shows that *GraalSP* achieves a geometric mean speedup of 7.46% in execution time compared to the default compiler configuration, which models program profiles using a uniform distribution. This dissertation also presents a detailed qualitative and quantitative analysis to position and compare the proposed solution against state-of-the-art static profilers. Additionally, to enhance and expand the evaluation and support developers in analyzing *GraalSP*'s predictions, this dissertation introduces the *GraalSP-PLog* tool. This tool allows developers to run the *GraalSP* static profiler on any program and generate detailed prediction reports, making it easier to inspect individual predictions and identify model mispredictions.

Since *GraalSP* provides substantial performance gains, has minimal impact on binary size and compile time, and includes a modern, fully automated model retraining pipeline, it is well-suited for commercial deployment. As a result, *GraalSP* has been the default static profiler for the *Enterprise GraalVM Native Image* compiler since June 2023, consistently improving performance with every build.

Keywords: compilers, static profiler, machine learning, regression, gradient boosting, ensemble models, Enterprise GraalVM Native Image

Research area: computer science

Research sub-area: compilers, machine learning

UDC number: 004.415.5(043.3)

Sadržaj

Glava 1

Uvod

Razvoj kompilatora odigrao je ključnu ulogu u napretku modernog računarstva. Prvi programabilni računari, poput računara ENIAC[1], programirani su „prespajanjem žica" odnosno direktnom rekonfiguracijom hardvera. Iako su ovi računari predstavljali revolucionaran tehnološki iskorak, ubrzo se pokazalo da ovakav pristup programiranju nije ni efikasan ni praktičan, naročito za kompleksne zadatke. Fon Nojmanova[2] arhitektura računara omogućila je programiranje u mašinskom ili asemblerskom jeziku niskog nivoa, što je predstavljalo napredak, ali je ovakav način programiranja i dalje bio nepraktičan, zamoran, sklon greškama i teško prenosiv između različitih platformi.

Već pedesetih godina dvadesetog veka pojavili su se prvi kompilatori koji su programerima omogućili da pišu kôd u apstraktnijim, ljudima razumljivijim programskim jezicima višeg nivoa. Ovi jezici omogućili su intuitivnije i izražajnije programiranje, dok je prevođenje tih programa u mašinski kôd postalo zadatak kompilatora. Kvalitet kompilatora se, između ostalog, ogledao u efikasnosti izvršavanja prevedenih programa. Kako performanse prevedenih programa direktno zavise od optimizacija koje kompilator implementira, ove optimizacije su još od samih početaka modernog računarstva u fokusu interesovanja istraživača i inženjera. Već više od sedamdeset godina oni nastoje da pomere granice i postignu sve veći stepen optimizacije, kako bi programi radili brže i efikasnije.

1.1 Optimizacije vođene profilom

Jedna od najvažnijih klasa optimizacija su optimizacije vođene profilom (engl. *profile-guided optimizations* ili *feedback-driven optimizations*). Optimizacije vođene profilom koriste profil izvršavanja programa da bi kreirale optimizovan program. Profil izvršavanja programa je skup informacija koje opisuju izvršavanje programa, kao što su brojevi izvršavanja grana naredbi grananja, broj poziva metoda, frekvencije izvršavanja blokova u grafu kontrole toka programa (engl. *control flow graph*), informacije o otključavanju ili zaključavanju monitora i slično.

Poznavanje profila izvršavanja programa omogućava agresivnu primenu optimizacija poput umetanja (engl. *inlining*) [82, 314], duplikacije repa [171, 176], optimizacija petlje [12], optimizacija keš memorije [261] i analize delimičnog izlaska (engl. *partial escape analysis*) [284]. Na taj način, optimizacije vođene profilom znatno poboljšavaju performanse prevedenih programa i prevazilaze domet statičkih optimizacija, koje nemaju uvid u ponašanje programa pri njegovom izvršavanju. Optimizacije vođene profilom uspešno poboljšavaju performanse razli-

[1] Univerzitet u Pensilvaniji, 1945.

[2] John von Neumann

čitih aplikacija poput kompilatora [317, 166], aplikacija koje se koriste u centrima podataka (engl. *warehouse-scale applications*) [46], veb pretraživača [203, 110] i procesora video igara (engl. *game engine*) [91]. One dovode do značajnog poboljšanja performansi izvršavanja programa, koje varira i može iznositi i preko 50% [228].

Da bi optimizacije vođene profilom značajno poboljšale performanse programa, neophodno je da profil izvršavanja programa bude kvalitetan [317, 47]. Kvalitetan profil odlikuje visoka pokrivenost koda[3] i visoka preciznost, odnosno on sadrži precizne informacije o izvršavanju najvećeg dela koda i na taj način verno oslikava ponašanje programa pri njegovom izvršavanju. Na primer, ako postoje precizni podaci o tome koje će se putanje u programu najčešće izvršavati, kompilator ih može optimizovati umetanjem poziva metoda sa tih putanja. Na taj način se smanjuju troškovi pozivanja tih metoda prilikom izvršavanja programa i poboljšavaju se performanse programa. Sa druge strane, nekvalitetan profil može navesti kompilator da umeće metode sa putanja koje će se retko izvršavati, trošeći resurse na pogrešan način i sprečavajući optimizaciju metoda koji će biti često izvršavani.

1.2 Dinamički i statički profajleri

Kako optimizacije vođene profilom mogu značajno poboljšati performanse prevedenih programa, u naučnim istraživanjima i praksi dosta pažnje posvećeno je prikupljanju kvalitetnih profila izvršavanja programa. Dinamičko profajliranje (engl. *dynamic profiling*) [196, 345, 260, 69] predstavlja vid dinamičke analize programa kojom se prikuplja profil izvršavanja programa. Ono omogućava prikupljanje najkvalitetnijih podataka o izvršavanju programa [124]. Međutim, dinamičko profajliranje značajno povećava složenost procesa izgradnje aplikacije. Umesto jednog procesa kompilacije, potrebno je obaviti dve kompilacije i prikupljanje profila, što zahteva značajno vreme i memorijske resurse [47, 124, 211, 52]. Osim toga, ovaj pristup otežava rad programerima jer nije jednostavno odabrati reprezentativne ulaze koji oponašaju uobičajene slučajeve korišćenja aplikacije, pokrivaju sve važne delove koda i obezbeđuju ravnomernu pokrivenost [47]. Dodatni izazov predstavlja i održavanje takvih ulaznih podataka za prikupljanje profila tokom vremena, kako aplikacija evoluira. Troškovi dinamičkog profajliranja su posebno značajni u kontekstu kontinuirane integracije, gde se oni vremenom sabiraju.

Statičko profajliranje (engl. *static profiling*) [326, 202] predstavlja alternativu dinamičkom profajliranju. Umesto da prikupljaju profile izvršavanja programa, statički profajleri predviđaju profile izvršavanja programa. Termin statički profajler se u kontekstu ove teze, kao i u naučnoj literaturi [326, 202, 68, 254, 242], koristi za alate koji predviđaju verovatnoće izvršavanja grana naredbi grananja, a ne druge tipove profila. Statički profajleri definišu atribute koji karakterišu naredbe grananja u internim reprezentacijama kompilatora ili u grafu kontrole toka programa, na osnovu kojih predviđaju verovatnoću izvršavanja grana. Oni funkcionišu u potpunosti statički, bez potrebe za dve kompilacije i potrebe za prikupljanjem podataka o izvršavanju prilikom same izgradnje aplikacije. Na taj način, statički profajleri pojednostavljuju proces izgradnje aplikacije, smanjuju vreme i potrošnju memorije i oslobađaju programere od potrage za reprezentativnim ulaznim podacima za prikupljanje profila.

Prvi statički profajleri oslanjali su se na heuristike [326, 13] za predviđanje verovatnoća izvršavanja grana. Takve heuristike su jednostavne i efikasne, zasnovane na intuiciji i praktičnom iskustvu programera. Svaka heuristika dodeljuje određenu verovatnoću izvršavanja grani, a konačna verovatnoća se zatim dobija ili primenom prve primenljive heuristike ili objedinja-

[3]Pokrivenost se u ovom kontekstu odnosi na pokrivenost koda ostvarenu prilikom prikupljanja profila.

vanjem predviđanja svih primenljivih heuristika — na primer, korišćenjem Dempster[4]–Šafer[5] teorije [111]. Iako relativno jednostavni, statički profajleri zasnovani na heuristikama i dalje se primenjuju u modernim kompilatorima, poput kompilatora *Clang* [202].

Tačnost predviđanja statičkih profajlera ima presudan značaj jer pogrešna predviđanja mogu da dovedu do donošenja loših optimizacionih odluka i pada performansi. Zbog toga se pri razvoju modernih statičkih profajlera sve češće koriste modeli mašinskog učenja, koji su znatno izražajniji od heuristika. Savremeni statički profajleri primenjuju modele mašinskog učenja za klasifikaciju [155, 225] i regresiju [39, 6]. Takvi modeli mogu da predvide znatno preciznije profile od heuristika [202]. Međutim, ovi modeli su osetljivi na odstupanja u podacima, koja se mogu javiti u bilo kojoj distribuciji i ne mogu se u potpunosti eliminisati [131, 161]. To znači da statički profajleri zasnovani isključivo na mašinskom učenju, iako kvalitetni i izražajni, ponekad mogu negativno da utiču na performanse prevedenih programa [254].

1.3 Doprinosi i organizacija disertacije

U ovoj disertaciji predstavljen je alat *GraalSP*, robustan statički profajler zasnovan na mašinskom učenju. *GraalSP* objedinjuje izražajnost i tačnost predviđanja modela mašinskog učenja sa statičkim heuristikama zasnovanim na intuiciji programera. Ove heuristike obogaćuju profajler dodatnim informacijama i osiguravaju dobre performanse prevedenih programa i u slučajevima odudarajućih podataka. U okviru rada na ovoj disertaciji razvijen je i alat *GraalSP-PLog* (*GraalSP Profiles Logger*) za analizu predviđanja statičkog profajlera *GraalSP*. Doprinosi ove disertacije obuhvataju više aspekata, koji su navedeni u nastavku.

- Definisan je izražajan i jezički nezavisan skup atributa za opisivanje naredbi grananja. Definisani skup atributa zasnovan je na grafovskoj internoj reprezentaciji kompilatora visokog nivoa, *Graal IR* [175] i grafu kontrole toka programa.
- Kreiran je novi skup podataka za obučavanje modela za predviđanje verovatnoća izvršavanja grana, zasnovan na modernim Java aplikacijama. Izvršena je analiza tog skupa podataka koja je pružila uvid u to kako treba dizajnirati i obučavati modele mašinskog učenja radi pouzdanijeg i kvalitetnijeg predviđanja verovatnoća izvršavanja grana.
- Dizajniran je i obučen regresioni model mašinskog učenja za predviđanje verovatnoća izvršavanja grana. Model gradijentnog pojačavanja *XGBoost* je odabran kao efikasan i precizan model.
- Razvijene su heuristike za predviđanje verovatnoća izvršavanja grana sa novom ulogom u kojoj one koriguju predviđanja modela mašinskog učenja. Na taj način, heuristike obezbeđuju visoke performanse optimizovanih programa čak i u slučajevima odudarajućih podataka.
- Razvijen je višejezičan, efikasan i robustan statički profajler *GraalSP*. On se može koristiti za predviđanje profila u svim aplikacijama napisanim u jezicima koji se prevode na Java bajtkod (kao što su Java, Skala ili Kotlin).
- Razvijen je alat *GraalSP-PLog* koji omogućava analizu predviđanja statičkog profajlera *GraalSP* i identifikaciju uspešnih, kao i eventualno pogrešnih predviđanja statičkog

[4]Arthur P. Dempster
[5]Glenn Shafer

profajlera. Alat omogućava detaljniju evaluaciju statičkog profajlera *GraalSP* i olakšava korisnicima razumevanje uticaja profajlera na optimizacije i performanse optimizovanih programa.

- Razvijeno je efikasno i sveobuhvatno softversko rešenje koje je integrisano u komercijalni kompilator *Enterprise GraalVM Native Image*[6] [320] počevši od verzije 23.0, objavljene u junu 2023. godine. Pri tome je statički profajler *GraalSP* podrazumevano uključen, pa se prilikom kompilacije programa automatski pokreće kako bi predvideo verovatnoće izvršavanja grana i usmeravao optimizacije vođene profilom.

- Postignuto je ubrzanje izvršavanja programa od 7.46%, geometrijske sredine[7], u poređenju sa osnovnom verzijom kompilatora *Enterprise GraalVM Native Image* koja koristi uniformnu raspodelu da modeluje verovatnoće izvršavanja grana. Evaluacija je izvršena na 28 test programa iz referentnih skupova test programa *Renaissance* [241], *DaCapo* [23] i *DaCapo con Scala* [267] koji se koriste za ocenu performansi kompilatora *Enterprise GraalVM Native Image*.

U okviru rada na ovoj disertaciji kao i bočnih istraživanja rađenih tokom razvoja ove disertacije, objavljeni su naučni radovi u međunarodnim i domaćim časopisima [68, 69], prijavljena su dva patenta u Sjedinjenim Američkim Državama [351, 140] i prezentovani su radovi [199, 350, 249] i apstrakti [67, 70, 279, 349] na međunarodnim i domaćim konferencijama. Disertacija je organizovana na sledeći način.

U poglavlju 2 (Osnove) prikazane su osnove mašinskog učenja, uz pregled nadgledanog učenja, nenadgledanog učenja i učenja potkrepljivanjem. Objašnjeni su atributi i načini njihovog kodiranja, kao i tehnike redukcije dimenzionalnosti. Pored toga, prikazana je platforma *Oracle GraalVM*, programski jezik Java i kompilator *Enterprise GraalVM Native Image*. Konačno, u ovom poglavlju prikazane su i tehnike profajliranja, dinamičko profajliranje i statičko profajliranje, kao i odnos između njih.

U poglavlju 3 (Statički profajleri) dat je pregled tehnika za statičko predviđanje izvršavanja grana i prikazane su tehnike za predviđanje izvršavanja grana koje koriste statičke heuristike i one koje koriste modele mašinskog učenja. Pored toga, u ovom poglavlju prikazani su i najsavremeniji statički profajleri: profajler Vua[8] i Larusa[9] [326], koji se zasniva na statičkim heuristikama, profajler Rotema[10] i Kuminsa[11] [254], koji koristi klasifikacioni model mašinskog učenja, kao i profajleri Ramana[12] i Lija[13] [242] i *VESPA* [202], koji se oslanjaju na duboke neuronske mreže za regresiju.

[6]Broj korisnika komercijalnog kompilatora *Enterprise GraalVM Native Image* nije javno dostupan. Nekomercijalnu verziju kompilatora, odnosno kompilator *GraalVM Native Image* koristi čak 58.16% korisnika platforme *GraalVM* [60, 339], četvrtina svih korisnika radnog okvira *Spring* (engl. *Spring framework*) [311, 200] i preko 3400 projekata otvorenog koda dostupnih na servisu *GitHub* [223]. Broj preuzimanja nekomercijalnih izdanja platforme *GraalVM* sa servisa *GitHub* premašuje devet miliona [297].

[7]Za agregaciju rezultata korišćena je geometrijska sredina jer se u literaturi smatra boljim izborom od aritmetičke kada je reč o agregiranju skaliranih vrednosti [96].

[8]Youfeng Wu

[9]James R. Larus

[10]Nadav Rotem

[11]Chris Cummins

[12]Easwaran Raman

[13]Xinliang David Li

U poglavlju 4 (Statički profajler *GraalSP*) predstavljen je statički profajler *GraalSP*, kao i skup atributa korišćenih za opisivanje naredbi grananja i proces kreiranja skupa podataka za obučavanje modela za predviđanje verovatnoća izvršavanja grana. Pored toga, u ovom poglavlju su prikazani i modeli mašinskog učenja korišćeni za predviđanje ovih verovatnoća, kao i statičke heuristike koje se primenjuju radi korekcije predviđanja modela. Predstavljen je i alat *GraalSP-PLog*, namenjen analizi i otkrivanju grešaka u predviđanjima statičkog profajlera *GraalSP*.

U poglavlju 5 (Implementacija) prikazana je implementacija statičkog profajlera *GraalSP*. Analizirani su skupovi podataka za obučavanje i validaciju modela mašinskog učenja za predviđanje verovatnoća izvršavanja grana, kao i načini implementacije samih modela, procedure obučavanja i softverska i hardverska podrška. Pored toga, predstavljena je implementacija statičkih heuristika za korekciju predviđanja modela, kao i implementacija alata *GraalSP-PLog*.

U poglavlju 6 (Evaluacija) prikazana je evaluacija statičkog profajlera *GraalSP*. Evaluacija statičkog profajlera fokusira se na evaluaciju obučenih modela za predviđanje verovatnoća izvršavanja grana i odabir najboljeg od njih. Evaluacija obučenih modela izvršena je u terminima metrika mašinskog učenja i uticaja odgovarajućih statičkih profajlera na vreme izvršavanja programa, veličinu generisanih izvršivih fajlova i vreme kompilacije programa. U ovom poglavlju prikazana su evaluacija i poređenje statičkog profajlera *GraalSP* sa relevantnim statičkim profajlerima. U tu svrhu, u kompilatoru *Enterprise GraalVM Native Image* implementiran je statički profajler Vua i Larusa [326]. Takođe, u ovom poglavlju prikazani su i rezultati ispitivanja uticaja veličine skupa atributa i veličine skupa podataka za obučavanje, kao i drugih aspekata treninga, poput težina instanci, na kvalitet modela. Konačno, analizirana su predviđanja statičkog profajlera *GraalSP* na nekoliko konkretnih primera programa.

U poglavlju 7 (Poređenje sa relevantnim statičkim profajlerima) prikazano je sveobuhvatno kvantitativno i kvalitativno poređenje statičkog profajlera *GraalSP* sa najboljim postojećim statičkim profajlerima: profajlerom Vua i Larusa [326], profajlerom Rotema i Kuminsa [254], profajlerom Ramana i Lija [242] i profajlerom *VESPA* [202].

U poglavlju 8 (Zaključci i pravi daljeg razvoja) sumirani su osnovni zaključci ove disertacije i predloženi mogući pravci daljeg istraživanja koji se mogu razvijati na osnovu predstavljenih rezultata. Pored toga, u ovom poglavlju dat je i kratak pregled radova objavljenih tokom izrade ove disertacije, kako onih koji se direktno odnose na temu, tako i onih nastalih u sklopu bočnih istraživanja.

U dodatku A (Implementacija heuristika Vua i Larusa) prikazana je implementacija statičkog profajlera Vua i Larusa zasnovanog na statičkim heuristikama u kompilatoru *Enterprise GraalVM Native Image*.

Glava 2

Osnove

U ovom poglavlju prikazaćemo tehnike nadgledanog mašinskog učenja poput tehnika preprocesiranja podataka [154, 162, 188], stabala odlučivanja [255, 32, 252], gradijentnog pojačavanja [18] i dubokih neuronskih mreža [169, 109]. U ovom poglavlju biće dat i pregled platforme *Oracle GraalVM* [60], sa posebnim akcentom na kompilator *GraalVM Native Image* [320]. Pored toga, biće opisani profajleri, osnovne i napredne tehnike profajliranja i optimizacije vođene profilom.

2.1 Mašinsko učenje

Skup podataka (engl. *dataset*) predstavlja kolekciju podataka koja se koristi za obučavanje modela mašinskog učenja. Svaki podatak, tj. instanca iz skupa podataka, opisuje se atributima (engl. *features*). Najčešći primeri atributa su numerički i tekstualni atributi. Na primer, telefonski imenik jeste skup informacija o pojedinačnim kontaktima, gde je svaki kontakt instanca opisana obično svojim imenom, prezimenom, brojem telefona i opciono slikom. Skup podataka sa označenim instancama (engl. *labels*) naziva se označen skup podataka (engl. *labeled dataset*).

Model mašinskog učenja predstavlja matematičku funkciju koja, na osnovu ulaznih podataka, može da daje predviđanja ili donosi odluke u cilju rešavanja konkretnog zadatka. Modeli mašinskog učenja uče zakonitosti koje važe u skupu podataka. Prema postavci problema učenja, problemi mašinskog učenja obično se klasifikuju u tri kategorije, u probleme nadgledanog mašinskog učenja (engl. *supervised learning*) [201, 271], probleme nenadgledanog mašinskog učenja (engl. *unsupervised learning*) [17, 104] i probleme učenja potkrepljivanjem (engl. *reinforcement learning*) [141].

Nadgledano učenje

U označenom skupu podataka svaka instanca opisana je skupom atributa i označena ciljnom promenljivom koju je potrebno naučiti. Modeli nadgledanog mašinskog učenja uče zakonitosti koje važe između atributa koji opisuju instance i njihovih oznaka. Termin nadgledano učenje motivisan je analogijom sa procesom učenja gde učitelj zadaje zadatke učeniku koji ih nakon toga rešava, nakon čega učitelj koriguje pogrešno rešene zadatke [213].

U zavisnosti od vrste oznaka, postoje dve osnovne vrste nadgledanog učenja: klasifikacija i regresija. Klasifikacija [4] podrazumeva učenje modela koji instancama dodeljuju diskretne oznake, tj. klase. Regresija [253, 193], s druge strane, podrazumeva predviđanje kontinualnih (neprekidnih) numeričkih vrednosti.

Primer zadatka nadgledanog mašinskog učenja i označenog skupa podataka jeste klasifikacija cveća u skupu podataka Iris [221]. Skup podataka Iris sadrži 150 instanci koje su opisane realnim brojevima odnosno merenjima koja opisuju svaki cvet, na primer dužinama i širinama listova. Svaka instanca označena je sa jednom od tri vrste cveća. Na skupu podataka Iris moguće je obučavati nadgledane modele mašinskog učenja za klasifikaciju [154, 155, 278].

Još jedan primer označenih skupova podataka za obučavanje modela nadgledanog mašinskog učenja jesu skup podataka za klasifikaciju rukom pisanih cifara *MNIST* [80] i skupovi podataka za klasifikaciju rukom pisanih slova *EMNIST* [59] i *NIST* [116]. Skup podataka *MNIST* sastoji se od 60000 slika rukom pisanih brojeva u formatu crno belih slika rezolucije 28×28. Svaka slika označena je brojem koji je na njoj napisan, odnosno cifrom od 0 do 9. Skupovi podataka *EMNIST* i *NIST* sadrže veći broj slika rukom pisanih karaktera, ne samo cifara, u većoj rezoluciji. Skupovi podataka rukom pisanih cifara primer su skupova u kojima su instance slike cifara, karakterisane položajem i vrednostima piksela. U zadacima nadgledanog mašinskog učenja, ovi podaci mogu se koristiti za obučavanje modela za klasifikaciju cifara od 0 do 9, na osnovu vizuelnih karakteristika svake slike, odnosno rasporeda piksela koji opisuju oblik napisane cifre.

Prethnodni primeri uključivali su označene skupove podataka gde su oznake uzimale neku od konačnog broja vrednosti odnosno jednu od konačnog broja klasa: tri klase u slučaju skupa podataka Iris i deset klasa u slučaju skupova podataka *MNIST* i *EMNIST*[1] i 62 klase u slučaju podataka NIST[2]. U slučaju skupa podataka o cenama nekretnina u gradu Bostonu [259, 219], nekretnine su opisane numeričkim atributima koji opisuju socijalne i ekonomske aspekte dela grada u kojem se nekretnina nalazi, kao i atributima koji opisuju samu nekretninu ali i demografske i infrastrukturne karakteristike dela grada u kome se nekretnina nalazi. Svaka nekretnina označena je cenom, koja je realna vrednost. Na ovom skupu podataka moguće je obučavati regresione modele mašinskog učenja za predviđanje cena nekretnina na osnovu njihovih karakteristika.

Nenadgledano učenje i učenje potkrepljivanjem

Nenadgledano mašinsko učenje odnosi se na modele koji se obučavaju na skupovima podataka koji nisu označeni. Metode nenadgledanog mašinskog učenja obično su specijalizovane za zadatke pronalaska strukture u podacima, na primer to su algoritmi klasterovanja (engl. *clustering*) [251, 137] ili redukcije dimenzionalnosti odnosno učenja reprezentacije podataka [305].

Učenje potkrepljivanjem odnosi se na scenarije učenja u kojima je potrebno rešiti neki problem izvođenjem niza akcija, čijim se zajedničkim delovanjem ostvaruje cilj odnosno rešava problem. U slučaju učenja potkrepljivanjem osnovna pretpostavka jeste postojanje subjekta koji se naziva agent, koji preduzima korake i izvodi radnju. Agent opaža stanje svog okruženja i na osnovu stanja preduzima neke od akcija za koje dobija nagrade (engl. *reward*) ili kazne. Rezultat učenja je optimalna politika (engl. *optimal policy*) koja se sastoji od odluka koje je potrebno doneti odnosno akcija koje je potrebno preduzeti u zavisnosti od stanja okruženja i stanja u kome se agent nalazi. Optimalna politika vodi do najboljeg rešenja.

[1] U skupu podataka EMNIST broj klasa zavisi od varijante skupa koja se koristi. Osnovna varijanta, *EMNIST MNIST*, sadrži 10 klasa i predstavlja cifre od 0 do 9, kao u originalnom *MNIST* skupu. Varijanta *EMNIST Letters* sadrži 26 klasa i obuhvata rukom pisana slova engleskog alfabeta, gde su velika i mala slova objedinjena u iste klase. *EMNIST Balanced* sadrži 47 klasa i uključuje cifre i izabrana slova, takođe sa objedinjavanjem velikih i malih slova. Najbogatija varijanta, *EMNIST ByClass*, sadrži 62 klase i razlikuje cifre, velika i mala slova kao posebne kategorije.

[2] U skupu podataka *NIST* nalaze se slike cifara, kao i velikih i malih slova engleskog alfabeta, ukupno 62 različita karaktera.

Ilustracija problema koji se rešava učenjem potkrepljivanjem jeste autonomna vožnja automobila. U primeru vožnje automobil je agent koji sprovodi akcije poput kretanja, stajanja, ubrzavanja, usporavanja i prelazi željeni put odnosno stiže na zadatu lokaciju. Stanje okruženja koje agent opaža je trenutno stanje na putu i stanje u saobraćaju koje ga okružuje. Pored toga, neki od primera učenja potkrepljivanjem jesu učenje agenta za igranje šaha ili prelazak nivoa u arkadnim igrama.

U nastavku ćemo se fokusirati na statistike za opis raspodela podataka i tehnike nadgledanog mašinskog učenja, prvenstveno na one koje koristimo za razvoj statičkog profajlera.

Statistike za opis raspodele podataka

Za numeričku karakterizaciju raspodele podataka, najčešće korišćene statistike su srednja vrednost (engl. *mean*), varijansa (engl. *variance*), standardna devijacija (engl. *standard deviation*), medijana (engl. *median*), percentili (engl. *percentiles*), minimum, maksimum i koeficijent asimetrije (engl. *skewness*). Kada se računaju na osnovu instanci iz skupa podataka, za ove statistike kažemo da su uzoračke jer se izračunavaju na osnovu uzorka, a ne cele populacije.

Srednja vrednost predstavlja prosečnu vrednost u skupu podataka. Računa se po formuli:

$$\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i,$$

gde su x_i vrednosti podataka[3], a n njihov ukupan broj.

Varijansa predstavlja meru prosečnog kvadratnog odstupanja podataka od srednje vrednosti. Definiše se kao:

$$s_n^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x})^2.$$

Varijansa izražava stepen rasipanja podataka oko srednje vrednosti i izražena je u kvadratnim jedinicama u odnosu na vrednosti podataka. *Standardna devijacija* predstavlja kvadratni koren varijanse i računa se po formuli:

$$s_n = \sqrt{s_n^2}.$$

Standardna devijacija je izražena u istim jedinicama kao i podaci, pa je intuitivnija za interpretaciju. Što je veća standardna devijacija, to podaci više variraju oko srednje vrednosti.

Medijana je vrednost koja deli sortiran skup podataka na dve jednake polovine, odnosno 50% podataka je manje ili jednako toj vrednosti. Slično, *percentili* predstavljaju vrednosti koje dele raspodelu podataka na određene procente. Najčešće korišćeni percentili su:

- 25. percentil (prvi kvartil): vrednost ispod koje se nalazi 25% podataka,
- 50. percentil (medijana): centralna vrednost raspodele, koja deli podatke na dva jednaka dela tako da je 50% podataka manje a 50% veće od te vrednosti i
- 75. percentil (treći kvartil): vrednost ispod koje se nalazi 75% podataka.

Minimum i maksimum su najmanja, odnosno najveća vrednost u posmatranom skupu podataka. Jednake su nultom i stotom percentilu, redom.

[3]Pretpostavka je da su podaci numerički, na primer realni brojevi.

Koeficijent asimetrije je statistička mera koja opisuje stepen nesimetričnosti raspodele podataka u odnosu na srednju vrednost. Na primer, Fišer–Pirsov koeficijent asimetrije [153] računa se po formuli:

$$g_1 = \frac{m_3}{m_2^{3/2}},$$

gde je m_i i-ti centralni moment koji se računa na sledeći način:

$$m_i = \frac{1}{n}\sum_{j=1}^{n}(x_j - \overline{x})^i,$$

a gde je $\overline{x}$ srednja vrednost, a n ukupan broj instanci podataka. Ako je koeficijent asimetrije pozitivan, raspodela je *nagnuta ulevo*, odnosno ima duži *rep* na desnoj strani. Rep raspodele odnosi se na deo krive raspodele koji se proteže daleko od njenog *„tela"*, odnosno od glavne koncentracije podataka. Rep raspodele predstavlja ekstremne vrednosti koje se javljaju ređe, ali značajno utiču na oblik raspodele. Na primer, pozitivna asimetrija znači da se manji broj veoma velikih vrednosti nalazi na desnoj strani krive raspodele, dok je većina podataka smeštena levo od srednje vrednosti. Ako je koeficijent asimetrije negativan, tada je raspodela *nagnuta udesno*, sa dužim repom na levoj strani. Na primer, normalna raspodela $\mathcal{N}(\mu, \sigma)$ ima koeficijent asimetrije jednak nuli jer je simetrična.

Za numeričku karakterizaciju raspodele podataka, u slučaju kada su elementi pozitivni, mogu se koristiti geometrijska sredina (engl. *geometric mean*) i geometrijska standardna devijacija (engl. *geometric standard deviation*). *Geometrijska sredina* [308] niza pozitivnih vrednosti $x_1, x_2, \ldots, x_n$ računa se kao:

$$\mu_g = \left(\prod_{i=1}^{n} x_i\right)^{1/n}.$$

Geometrijska sredina je odgovarajuća mera kada se vrednosti menjaju eksponencijalno i u slučaju asimetrične raspodele koja može postati simetrična logaritamskom transformacijom [191].

Geometrijska standardna devijacija [90] niza pozitivnih vrednosti računa se kao:

$$\sigma_g = \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\left(\frac{x_i}{\mu_g}\right)\right)^2}\right).$$

Ona meri proporcionalno odstupanje od geometrijske sredine. Veće vrednosti ukazuju na veću disperziju podataka u odnosu na sredinu, dok ako su svi elementi jednaki, geometrijska standardna devijacija ima vrednost jednaku jedan.

Atributi i njihovo kodiranje

Atributi (engl. *features*) predstavljaju karakteristike kojima se opisuju instance u skupu podataka. Atributi kojima se opisuju instance najčešće su numerički ili kategorički. Numerički atributi predstavljeni su realnim ili celim brojevima i opisuju merljiva svojstva instance, na primer širina i dužina listova cveta. Kategorički atributi uzimaju jednu od konačno mnogo vrednosti koje pripadaju određenim kategorijama. Primer kategoričkog atributa može biti boja cveta, pri čemu vrednosti mogu biti crvena, žuta, bela i slično.

Da bi modeli mašinskog učenja mogli da koriste kategoričke atribute, iste je potrebno kodirati, odnosno konvertovati u oblik razumljiv modelima mašinskog učenja, najčešće u vektore numeričkih vrednosti. Tipičan način kodiranja kategoričkih atributa je jedinično kodiranje (engl. *one-hot encoding*). Jedinično kodiranje podrazumeva kodiranje kategoričkih promenljivih vektorom, gde je dimenzija vektora jednaka broju mogućih kategorija koje atribut može uzeti i gde svaka koordinata odgovara tačno jednoj vrednosti atributa. Na primer, kategorička promenljiva boja cveta koja može uzeti vrednosti *crvena*, *bela* i *žuta* kodira se sa vektorom dužine tri, gde se kategorija *crvena* kodira vektorom $[1, 0, 0]$, kategorija *bela* vektorom $[0, 1, 0]$ a kategorija *žuta* vektorom $[0, 0, 1]$.

Jedinično kodiranje pokazalo se kao dobar način kodiranja kategoričkih promenljivih u slučaju atributa koji uzimaju manji broj kategorija [250]. U slučaju kategoričkih promenljivih koje mogu uzeti veliki broj kategorija, preporučljivo je koristiti tehniku kao što je heširanje atributa (engl. *hashing trick*) [313, 99] ili Bajesovo kodiranje (engl. *target encoding, Bayesian target encoding*) [73, 237, 164].

Redukcija dimenzionalnosti

Tehnike redukcije (smanjenja) dimenzionalnosti koriste se kako bi se smanjio broj atributa kojima se opisuju instance skupa podataka, uz očuvanje većine bitnih informacija koje ti atributi nose. Redukcija dimenzionalnosti koristi se za poboljšanje performansi modela. Pored toga, tehnike redukcije dimenzionalnosti umanjuju preprilagođavanje modela. Preprilagođavanje (engl. *overfitting*) [85, 122] nastaje kada model, osim suštinskih obrazaca, nauči i šum ili slučajne nepravilnosti iz skupa podataka za obučavanje. To vodi ka učenju zavisnosti koje ne važe u raspodeli podataka koja se modeluje, već su slučajno prisutne samo u skupu za obučavanje modela. Preprilagođavanje dovodi do lošijih predviđanja modela na podacima koje nije video tokom obučavanja.

Dve najčešće korišćene tehnike redukcije dimenzionalnosti jesu redukcija dimenzionalnosti na osnovu varijanse u podacima (engl. *variance-based feature selection*) [179] i analiza glavnih komponenti (engl. *principal components analysis*, PCA) [1, 322].

Redukcija dimenzionalnosti na osnovu varijanse u podacima uključuje rangiranje atributa na osnovu njihove varijanse u skupu za obučavanje i čuvanja samo onih atributa sa najvećom varijansom [159, 41, 179]. U zavisnosti od implementacije mogu se čuvati samo oni atributi koji variraju više od unapred definisanog praga, ili se može čuvati samo n atributa koji najviše variraju, za željenu izlaznu dimenziju n. Tehnika redukcije dimenzionalnosti na osnovu varijanse atributa motivisana je činjenicom da ukoliko atribut ima malu varijansu, to znači da je on praktično konstantan atribut, čije vrednosti ne variraju puno od instance do instance u skupu za obučavanje modela, pa samim tim on ne može poslužiti da se njime objasni variranje ciljne promenljive.

Analiza glavnih komponenti transformiše skup podataka visoke dimenzionalnosti u prostor niže dimenzionalnosti identifikovanjem *glavnih komponenti*, tj. linearnih kombinacija originalnih atributa koje su međusobno ortogonalne [322, 162, 188, 1]. Analiza glavnih komponenti hvata pravce maksimalne varijanse podataka, koji kreiraju novi skup atributa kojima se opisuju instance. Komponente rangira prema objašnjenoj varijansi, pri čemu prva komponenta objašnjava najviše varijanse, dok preostale ortogonalne komponente objašnjavaju ostatak varijanse. Da bi se se smanjila dimenzionalnost skupa prodataka, bira se n najvažnijih komponenti tj. atributa, gde n odgovara željenoj dimenziji prostora.

Modeli nadgledanog mašinskog učenja

Modeli mašinskog učenja određeni su vrednostima parametara (engl. *learnable parameters*). U kontekstu nadgledanog učenja, obučavanje modela (engl. *model training*) odnosi na podešavanje parametara modela kako bi se minimizovala greška predviđanja modela. Cilj obučavanja modela jeste učenje zavisnosti između atributa kojima se opisuju instance podataka i vrednosti kojima su one označene. Obučavanje ima za cilj da proizvede modele koji dobro generalizuju, gde generalizacija označava sposobnost modela da precizno predviđa oznake za nove, prethodno neviđene podatke.

Regularizacija obuhvata skup tehnika koje se primenjuju tokom obučavanja modela sa ciljem poboljšanja njegove sposobnosti generalizacije. Regularizacione tehnike deluju tako što ograničavaju kompleksnost modela ili uvode dodatne izvore varijacije tokom učenja, čime se podstiče model da nauči opštije i robusnije reprezentacije koje bolje generalizuju.

Pored parametara, modeli mašinskog učenja sadrže i hiperparametre (engl. *hyperparameters*) koji konfigurišu model (npr. dubinu stabla odlučivanja) [93, 334, 269]. Tokom obučavanja modela sa hiperparametrima, deo skupa podataka za obučavanje se koristi kao validacioni skup da bi se ocenio izbor hiperparametara. Naprednije tehnike izbora odnosno evaluacije modela uključuju unakrsnu validacija (engl. *cross validation*) [289, 121, 19]. Unakrsna validacija je tehnika kod koje se skup podataka deli na podskupove za višestruko obučavanje i validaciju modela, pružajući pouzdaniju procenu kvaliteta modela korišćenjem većeg validacionog skupa.

U nastavku ćemo diskutovati mere kvaliteta modela mašinskog učenja kao i najčešće korišćene modele mašinskog učenja u kontekstu statičkih profajlera: stabla odlučivanja (engl. *decision tree*) [252], gradijentno pojačavanje (engl. *gradient boosting*, *XGBoost*) [18] i duboke neuronske mreže (engl. *deep neural networks*) [169].

Mere kvaliteta modela

Kvalitet predviđanja klasifikacionih modela mašinskog učenja obično se ocenjuje korišćenjem metrika kao što su tačnost (engl. *accuracy*), preciznost (engl. *precision*), odziv (engl. *recall*) i F1 skor (engl. *F1 score*) [333, 112]. Sve ove mere zasnivaju se na matrici konfuzije, koja prikazuje kako su instance stvarnih klasa raspoređene među predviđenim klasama. Idealna klasifikacija odgovara dijagonalnoj matrici, dok nedijagonalni elementi označavaju greške.

U slučaju regresionih modela, greška u predviđanjima modela se obično meri korišćenjem srednjekvadratne greške (engl. *mean squared error*, MSE), korena srednjekvadratne greške (engl. *root mean square error*, RMSE) [312, 299], koeficijenta determinacije poznatog i kao R^2 skor (engl. R^2 score) [6, 207, 119], kao i težinskih vrednosti MSE i RMSE metrika.

MSE izračunava prosečne kvadratne razlike između predviđenih i stvarnih vrednosti. Računa se po formuli:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2,$$

gde je y_i stvarna vrednost, $\hat{y}_i$ vrednost predviđena od strane modela, a n ukupan broj instanci.

RMSE izračunava kvadratni koren srednjekvadratne greške, nudeći prednost da se rezultat izražava u istim jedinicama kao i ciljne vrednosti. RMSE se izračunava po formuli:

$$\text{RMSE} = \sqrt{MSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}.$$

R^2 skor izračunava se kao:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \mu)^2},$$

gde je y_i stvarna vrednost, $\hat{y}_i$ predviđanje modela, a μ prosečna vrednost stvarnih izlaza. R^2 skor predstavlja procenat varijanse u podacima koji je model uspešno objasnio.

U procesu nadgledanog obučavanja modela mašinskog učenja, težine instanci mogu dati prioritet pojedinačnim instancama skupa za obučavanje, usmeravajući model ka preciznijem predviđanju instanci sa većim težinama. Na ovaj način, model može prioritetizovati pojedinačne instance i poboljšati performanse u određenim podskupovima skupa za obučavanje. Korišćenje težina instanci menja interpretaciju metrika za ocenjivanje kvaliteta modela.

U slučaju regresionih modela, MSE i RMSE više ne mere jednostavno prosečno odstupanje predviđanja od stvarnih vrednosti za sve instance, već daju veću važnost instancama sa većim težinama. Težinski MSE (engl. *weighted mean squared error*, WMSE) i težinski RMSE (engl. *weighted root mean squared error*, WRMSE) pružaju ponderisanu evaluaciju kvaliteta modela, uzimajući u obzir različitu važnost instanci, koja se kvantifikuje kroz težine w_i.

Težinski MSE se računa kao:

$$\text{WMSE} = \frac{\sum_{i=1}^{n} w_i(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} w_i},$$

gde w_i predstavlja težinu instance i, y_i stvarnu vrednost, a $\hat{y}_i$ predviđanje modela.

Analogno, *težinski RMSE* se računa kao:

$$\text{WRMSE} = \sqrt{\frac{\sum_{i=1}^{n} w_i(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} w_i}}.$$

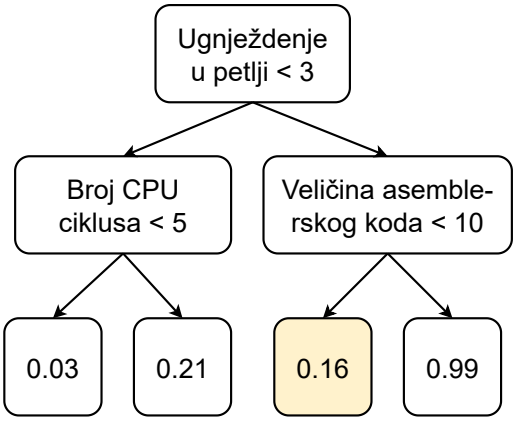
Ove ponderisane metrike omogućavaju pravedniju i ciljaniju procenu performansi modela, posebno u situacijama kada određene instance (npr. retki slučajevi ili kritični podaci) imaju veću važnost u praktičnoj primeni.

Stabla odlučivanja

Stablo odlučivanja (engl. *decision tree*) [252, 277] je jedan od najintuitivnijih modela mašinskog učenja koji koristi strukturu nalik stablu za modelovanje podataka. Stablo odlučivanja sastoji se od čvorova, grana i listova. Svaki čvor stabla ispituje jednu od karakteristika instance podataka odnosno jedan od atributa i, na osnovu vrednosti tog atributa, propagira donošenje odluke niz stablo. Listovi predviđaju ciljnu promenljivu, jednu od klasa u slučaju klasifikacije ili kontinualnu vrednost u slučaju regresije. Glavna prednost korišćenja stabala odlučivanja je njihova interpretabilnost [31, 100]. Još jedna prednost stabala odlučivanja je brzina obučavanja i predviđanja kao i to što su jednostavna za obučavanje i korišćenje [32].

Slika 2.1 ilustruje jedno plitko stablo odlučivanja koje predviđa verovatnoće izvršavanja delova programa ispitujući samo tri karakteristike koje opisuju delove programa: dubinu ugnežđenja u petljama, veličinu asemblerskog koda koji odgovara tom delu programa i broj CPU ciklusa potreban za izvršavanje tog dela programa. Na primer, pretpostavimo da postoji deo programa koji se nalazi na vrhu funkcije (tj. na nultoj dubini ugnežđenja u petljama) sa veličinom asemblerskog koda od 5 bajtova. U tom slučaju, stablo odlučivanja sa slike 2.1 predviđa da je verovatnoća izvršavanja tog dela programa 0.16.

Preprilagođavanje stabala odlučivanja može se desiti ako je stablo previše složeno i suviše dobro prati podatke skupa za obučavanje, što dovodi do loših performansi na novim podacima.



Slika 2.1: Primer modela stabla odlučivanja

Odsecanje stabla (engl. *tree pruning*) [30] pomaže u pojednostavljivanju modela i uključuje uklanjanje grana nakon što je stablo u potpunosti izgrađeno. Odsecanjem stabla iterativno se uklanjaju podstabla sa najmanjim uticajem na performanse modela.

Najpopularnije implementacije stabala odlučivanja [231] agresivnost odsecanja stabla kontrolišu sa parametrom cpp_α. Ovaj parametar predstavlja kompromis između kvaliteta predviđanja stabla na podacima skupa za obučavanje i njegove jednostavnosti, pri čemu veće vrednosti parametra cpp_α dovode do agresivnijeg odsecanja i jednostavnijih stabala.

Stabla odlučivanja mogu se koristiti za rangiranje atributa u odnosu na njihovu prediktivnu moć [3]. Ginijev značaj (engl. *Gini importance*) [177] atributa predstavlja (normalizovano) ukupno smanjenje nečistoće u podacima (engl. *data impurity*) [208, 338] koje se postiže kada model stabla vrši podelu podataka u čvoru na osnovu vrednosti tog atributa.

Gradijentno pojačavanje

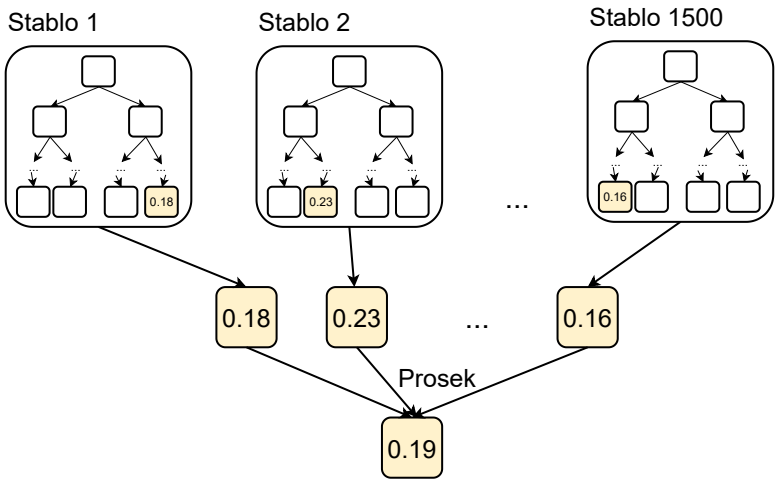
Model gradijentnog pojačanja predstavlja vrstu ansambl modela. Ansambli su grupe više modela mašinskog učenja koji kolektivno donose odluke, agregacijom (moguće sa težinama) predviđanja pojedinačnih modela iz ansambla. Ansambli su obično među najboljim modelima jer agregacija modela smanjuje varijansu, što dovodi do stabilnijih i pouzdanijih predviđanja [256, 15]. Osnovni modeli od kojih se sastoji ansambl nazivaju se slabi modeli (engl. *weak learners*).

Pri obučavanju modela gradijentnog pojačavanja iterativno se dodaju novi modeli u ansambl, pri čemu se svaki novi model prilagođava rezidualima (tj. greškama) prethodnih modela. Na taj način, ansambl postepeno poboljšava svoje performanse fokusirajući se na instance koje je najteže predvideti.

Model *XGBoost* [51] je skalabilna i fleksibilna implementacija algoritma gradijentnog pojačavanja. Kod modela XGBoost, ansambl se sastoji od stabala odlučivanja. Na slici 2.2 ilustrovan je ansambl za regresiju koji se sastoji od 1500 stabala. Model daje konačno predviđanje tako što uprosečava rezultate svih 1500 slabih modela od kojih se sastoji.

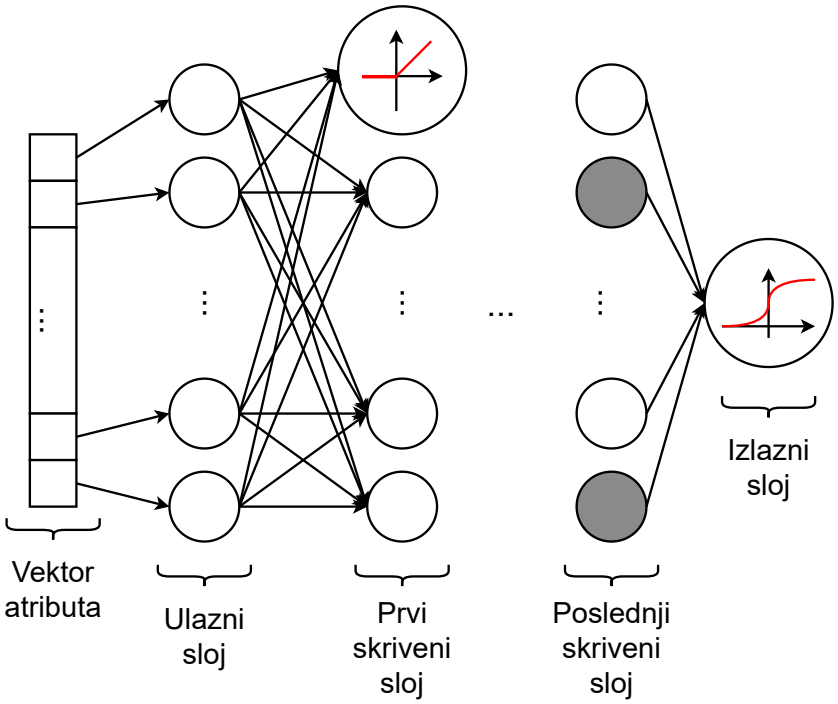
Duboka neuronska mreža

Neuronska mreža (engl. *neural network*) [169, 265, 293] sastoji se od slojeva neurona koji transformišu ulazni vektor artibuta u izlazni vektor. Neuron predstavlja osnovnu računsku jedi-



Slika 2.2: Model gradijentnog pojačavanja koji se sastoji od 1500 stabala odlučivanja

nicu u neuronskoj mreži. Svaki neuron prima jedan ili više ulaznih signala (najčešće numeričke vrednosti), izračunava njihovu ponderisanu (težinsku) sumu, eventualno dodaje pristrasnost (engl. *bias*) i zatim primenjuje nelinearnu aktivacionu funkciju na dobijenu vrednost. Rezultat ove transformacije prosleđuje se kao izlaz ka neuronima narednog sloja. Neuronska mreža ilustrovana je na slici 2.3.



Slika 2.3: Model duboke neuronske mreže za regresiju

Svaki neuron iz skrivenih slojeva mreže računa ponderisani zbir svojih ulaza koristeći težine, koje predstavljaju jačine veza sa neuronima iz prethodnog sloja. Ove težine čine parametre modela i ključne su za učenje jer određuju kako informacije protiču kroz mrežu. Težine povezane

sa jednim neuronom definišu koliko snažno taj neuron reaguje na izlaze neurona iz prethodnog sloja.

Aktivaciona funkcija je nelinearna funkcija koja uvodi nelinearnost u model, omogućavajući modelu da uči složene obrasce u podacima. Najčešće korišćene aktivacione funkcije uključuju funkciju *ReLU* (engl. *rectified linear unit*):

$$relu(x) = max(0, x)$$

i sigmoidnu aktivacionu funkciju:

$$\sigma(x) = \frac{1}{1 + e^{-x}},$$

koja predviđanje modela prilagođava opsegu $0 - 1$ [169]. Neuroni iz skrienih slojeva mreže sa slike 2.3 koriste aktivacionu funkciju *ReLU*, dok neuron iz izlaznog sloja koristi sigmoidnu aktivacionu funkciju[4].

Svi slojevi neuronske mreže čiji neuroni prosleđuju svoje izlaze neuronima u drugim slojevima nazivaju se skrivenim slojevima. Neuronska mreža sa više od jednog skrivenog sloja naziva se *duboka* neuronska mreža (engl. *deep neural network*, DNN). Danas postoje neuronske mreže sa više od 100 slojeva [123, 327]. Stoga je definicija duboke neuronske mreže postala relativna [120]. Poslednji sloj u mreži naziva se *izlazni* sloj mreže. Pri tom imenovanju, svi slojevi mreže osim izlaznog sloja nazivaju se skrivenim slojevima.

Svaki sloj mreže sadrži više neurona povezanih sa svim neuronima iz prethodnog sloja i svim neuronima iz narednog sloja. Izuzeci su ulazni i izlazni sloj. Neuroni ulaznog sloja povezani su samo sa neuronima narednog sloja jer ne postoji prethodni sloj, dok su neuroni izlaznog sloja povezani samo sa neuronima prethodnog sloja jer ne postoji naredni sloj. Kod neuronskih mreža za klasifikaciju, izlazni sloj ima onoliko neurona koliko ima klasa u zadatku klasifikacije, pri čemu svaki neuron predviđa verovatnoću da instanca pripada jednoj od klasa. Kod neuronskih mreža za regresiju, koje predviđaju neprekidnu ciljnu promenljivu, izlazni sloj obično sadrži samo jedan neuron.

Optimizacioni algoritmi prilagođavaju težine modela tokom obučavanja modela [27, 8]. Optimizator Adam (engl. *Adaptive Moment Estimation*) [147] jedan je od najčešće korišćenih optimizacionih algoritama za obučavanje dubokih neuronskih mreža. Pruža adaptivne stope učenja [75] i momentum [182, 181] za efikasnu i bržu konvergenciju tokom obučavanja modela. Pored osnovne verzije optimizatora Adam, razvijene su i varijacije poput optimizatora *ND-Adam* [343] ili *Adam W* [184], koje koriste dodatne tehnike za prilagođavanje brzine učenja ili poboljšanje stabilnosti i efikasnosti konvergencije tokom obučavanja neuronskih mreža.

Izostavljanje (engl. *dropout*) [281] je tehnika regularizacije u kojoj se nasumično odabrani neuroni privremeno zanemaruju odnosno „isključuju" tokom obučavanja duboke neuronske mreže. Izostavljanje neurona sprečava preprilagođavanje i poboljšava generalizaciju modela. Na slici 2.3 ilustrovano je izostavljanje neurona time što su izostavljeni neuroni u poslednjem skrivenom sloju obojeni sivom bojom.

2.2 Platforma *Oracle GraalVM*

U ovoj sekciji biće predstavljeno izvršavanje Java programa, kao i platforma *Oracle GraalVM*, sa posebnim fokusom na kompilator *GraalVM Native Image*.

[4]Iako se sigmoidna aktivaciona funkcija obično koristi u kontekstu klasifikacionih modela mašinskog učenja, može se koristiti i u regresionim modelima.

Izvršavanje Java programa

Programski jezik Java jedan je od najpopularnijih i najčešće korišćenih programskih jezika za razvoj poslovnih aplikacija. Kompanija *Sun Microsystems* je 1995. godine predstavila programski jezik Javu [9] kao programski jezik opšte namene. Kompanija *Oracle* je 2010. godine izvršila akviziciju kompanije *Sun Microsystems* i od tada se programski jezik Java razvija u okviru kompanije *Oracle*.

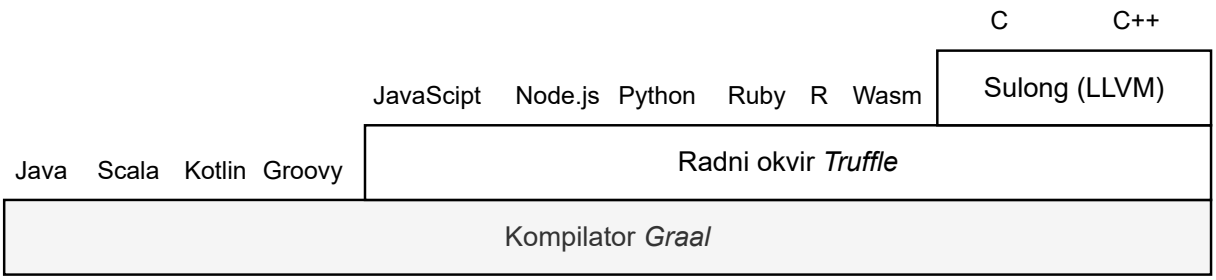
Java radno okruženje (engl. *Java runtime environment*, skraćeno *JRE*) koristi se za izvršavanje programa napisanih u programskom jeziku Java. Izvršavanje Java programa podrazumeva da se kôd najpre prevode u međureprezentaciju pod nazivom Java bajtkod (engl. *Java bytecode*) [335], koja je nezavisna od platforme i arhitekture (za sve sisteme generiše se isti bajtkod).

Java radno okruženje obuhvata Java virtuelnu mašinu (engl. *Java virtual machine*, skraćeno *JVM*), koja izvršava programe napisane u bajtkod formatu. Kako bi bajtkod bio nezavisan od operativnog sistema i arhitekture računara, Java virtuelna mašina se posebno implementira i prilagođava za svaku konkretnu platformu.

Danas najpoznatije Java virtuelne mašine su *HotSpot* JVM [227] (virtuelna mašina nastala 1999. godine u kompaniji *Syn Microsystems*, koja se danas razvija u okviru kompanije *Oracle*), *J9* JVM [246] kompanije *IBM* i JVM koja je deo platforme *GraalVM* [63] (koji takđe razvija kompanija *Oracle*, odnosno njen deo *Oracle Labs*). Platforma *GraalVM* podržava standardno izvršavanje Java programa, tj. kompilaciju u toku izvršavanja programa (engl. *just-in-time*, JIT kompilacija) [157] ali i kompilaciju pre vremena izvršavanja programa (engl. *ahead-of-time*, AOT kompilacija) [309]. AOT kompilacija podrazumeva integraciju koda aplikacije i dela virtuelne mašine koji je neophodan za izvršavanje same aplikacije, u jedinstven izvršivi fajl.

Pregled platforme *Oracle GraalVM*

Platforma *Oracle GraalVM* (ili skraćeno *GraalVM*) [63] omogućava prevođenje programa pisanih u različitim programskim jezicima u mašinski kôd za različite operativne sisteme i arhitekture. Platforma *GraalVM* koristi kompilator *Graal* [331, 282, 88, 89, 283, 284, 240, 239] za prevođenje i optimizaciju aplikacija koje su napisane u jezicima koji se kompajliraju u Java bajtkod, poput jezika Java, Skala [217, 218] (engl. *Scala*), Kotlin [132, 257] i Gruvi (engl. *Groovy*) [152, 16]. Šema platforme *GraalVM*, kao i pregled podržanih jezika, prikazani su na slici 2.4.



Slika 2.4: Šema platforme *GraalVM* i spisak podržanih jezika

Platforma *GraalVM* koristi radni okvir (engl. *framework*) *Truffle* [332, 324, 330] za implementaciju interpretatora zasnovanih na apstraktnim sintaksnim stablima (engl. *abstract syntax trees*, AST). *Truffle* implementacije programskih jezika koriste tehniku parcijalne evaluacije (engl. *partial evaluation*) [101] AST reprezentacije programa. Ovi interpretatori su napisani u

programskom jeziku Java. Platforma *GraalVM* efikasno prevodi i izvršava dinamičke jezike *JavaScript* [198, 66], *Node.js* [268], *Python* [307, 258], *Ruby* [296, 95], *R* [133, 65] i *WebAssembly* (*Wasm*) [347].

Platforma *GraalVM* koristi LLVM interpretator bajtkoda *Sulong* [248] da kompilira i izvršava programe napisane u programskim jezicima niskog nivoa, na primer, jezicima *C* [145], *C++* [290] i *Fortran* [2, 54]. Sulong koristi prednji deo kompilatorske infrastrukture *LLVM* [167] da kompilira programe u *LLVM* bajtkod, kog zatim izvršava korišćenjem radnog okvira *Truffle*. Integracijom bajtkod interpretatora *Sulong* i *Truffle* radnog okvira u platformu *GraalVM* omogućava se efikasno prevođenje, optimizovanje i izvršavanje programa nezavisno od ulaznog jezika na kom je program napisan. Na taj način platforma *GraalVM* postaje višejezična platforma obzirom da podržava prevođenje i izvršavanje većeg broja različitih programskih jezika.

Kompilacija u okviru platforme *GraalVM*

Platforma *GraalVM* podržava kompilaciju tokom izvršavanja programa i kompilaciju pre izvršavanja programa. Kompilacija u toku izvršavanja podrazumeva da se delovi programa prevode sekvencijalno, neposredno pre njihovog izvršavanja. Na taj način kompilator paralelno prevodi i izvršava program, što mu omogućava da lako prikuplja profil izvršavanja i koristi ga za izvođenje optimizacija.

Kompilacija pre vremena izvršavanja podrazumeva da se ceo program prevodi unapred, pre nego što se pokrene, pri čemu se iz izvornog (najčešće višeg) programskog jezika generiše izvršivi fajl. AOT kompilacija takođe može koristiti profil izvršavanja programa za optimizaciju, ali taj profil mora biti prikupljen tokom posebnog izvršavanja programa za sakupljanje profila, pre nego što se izgradi optimizovana verzija programa.

Kompilacija pre vremena izvršavanja ima nekoliko prednosti u odnosu na kompilaciju tokom izvršavanja [309]. Kompilatori koji koriste AOT pristup prevode program samo jednom, pre njegovog pokretanja, umesto da to rade tokom svakog izvršavanja. Time se eliminišu dodatni vremenski i memorijski zahtevi koje JIT kompilacija nameće zbog paralelnog prevođenja i izvršavanja koda. Kako AOT kompilacija proizvodi samostalne izvršive fajlove, ona time omogućava brže pokretanje aplikacija i manju potrošnju memorije, što je posebno značajno u okruženjima sa ograničenim resursima. Još jedna važna prednost AOT kompilacije je mogućnost primene šireg spektra optimizacija. Kako je ceo kod dostupan u trenutku prevođenja, moguće je raditi globalne optimizacije koje zahvataju više modula i funkcija.

Takođe i kompilacija tokom vremena izvršavanja ima neke prednosti u odnosu na kompilaciju pre vremena izvršavanja programa [309]. JIT kompilacija omogućava jednostavno prikupljanje i korišćenje profila izvršavanja programa za izvršavanje optimizacija. Po svom dizajnu, AOT kompilatorima je teže da prikupe profil izvršavanja programa jer to moraju učiniti u posebnom izvršavanju programa, a ne tokom izvršavanja samog programa.

Kompilator *GraalVM Native Image*

Kompilator *GraalVM Native Image* [320] omogućava AOT kompilaciju u okviru platforme *GraalVM*. Kompilator *GraalVM Native Image* implementiran je u programskom jeziku Java i koristi se za prevođenje aplikacija iz Java bajtkoda u izvršive programe. Za razliku od dotadašnje prakse, gde su kompilatori za Javu bili implementirani u jeziku C++[5], *GraalVM* je

[5] *HotSpot JVM* i kompilator *C2* implementirani su u programskom jeziku C++.

doneo novinu time što je kompilator razvijen u Javi. Ovakav pristup donosi brojne prednosti, među kojima su jednostavnije upravljanje memorijom, dostupnost savremenijih razvojnih alata i efikasnija implementacija naprednih jezičkih osobina, poput polimorfizma [172].

Kompilator *GraalVM Native Image* dostupan je u standardnom (engl. *community*) izdanju i naprednom (engl. *enterprise*) izdanju. Na napredno izdanje kompilatora će se u daljem tekstu referisati kao na kompilator *Enterprise GraalVM Native Image*. Kompilator *Enterprise GraalVM Native Image*, uključuje dinamičko profajliranje kao i naprednije tehnike optimizacije koje omogućavaju bolje performanse aplikacija.

Graal IR

Kompilator *GraalVM Native Image* koristi internu reprezentaciju *Graal IR* [88, 175]. *Graal IR* je grafovska reprezentacija visokog nivoa kojom se predstavlja bajtkod programa. Zasnovana je na konceptu mora čvorova (engl. *sea-of-nodes*) [79] i koristi svojstvo jedinstvene statičke dodele (engl. *static single-assignment form*, skraćeno *SSA form*) za definisanje instrukcija [72, 192]. *Graal IR* je struktuiran kao usmereni graf i na taj način pojednostavljuje implementaciju optimizacija kompilatora [88, 28].

Nad *Graal IR* grafom, kompilator *GraalVM Native Image* konstruiše graf kontrole toka programa (engl. *control flow graph*), koji predstavlja strukturu toka izvršavanja kroz različite delove programa. Graf kontrole toka sastoji se od blokova, pri čemu svaki blok predstavlja sekvencu naredbi koje se izvršavaju linearno, bez grananja, sve dok se ne dođe do tačke odlučivanja ili kraja bloka. Svaki blok grafa kontrole toka sadrži jedan ili više čvorova iz originalnog *Graal IR* grafa, odnosno konkretne naredbe koje se izvršavaju. Tipično, na kraju svakog bloka nalazi se najviše jedna naredba grananja, koja određuje dalji tok izvršavanja – na primer, uslovni skok ili povratak iz metode. Na osnovu tih grananja, blokovi su međusobno povezani granama koje predstavljaju moguće putanje izvršavanja programa.

Na listingu 1 prikazan je kôd metode `sum` koja koristi petlju `for` da sumira prvih n prirodnih brojeva, gde se broj n prosleđuje kao argument metode[6]. Na slici 2.5 prikazani su *Graal IR* graf i graf kontrole toka ove metode. Radi bolje preglednosti, pojedini detalji *Graal IR* grafa metode su pojednostavljeni. Na primer, izostavljena je numeracija svih čvorova. Čvorovi u *Graal IR* grafu mogu biti fiksni (engl. *fixed*) i pokretni (engl. *floating*). Fiksni čvorovi odgovaraju kontroli toka u programu, dok pokretni čvorovi odgovaraju toku podataka u programu. Pokretni čvorovi označeni su plavom bojom. Blokovi grafa kontrole toka obojeni su sivom bojom i numerisani od *B0* do *B3*.

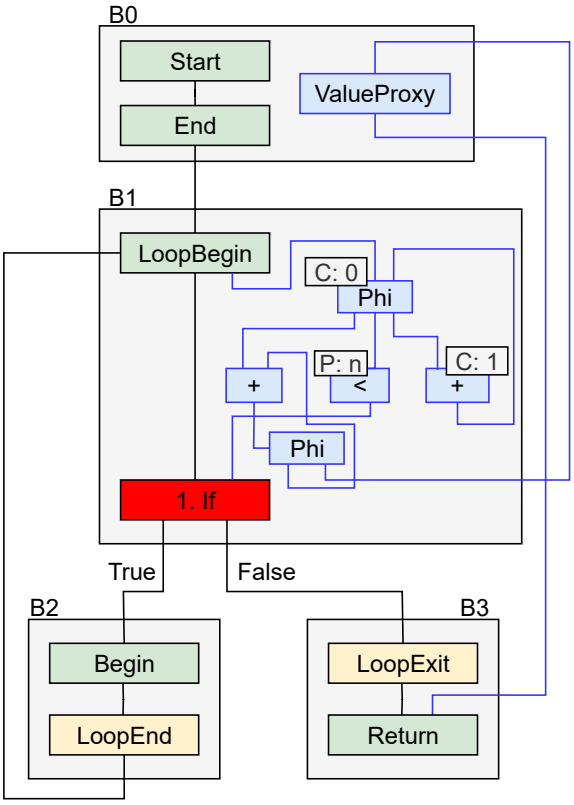
Pokretni čvorovi formiraju graf toka podataka. Oni predstavljaju operacije nad podacima, poput sabiranja, oduzimanja, množenja, deljenja, bitovskih operacija, konverzije tipova i slično. Pokretni čvorovi služe za prosleđivanje vrednosti između instrukcija. Oni mogu imati ulaze od drugih pokretnih čvorova ili konstanti i proizvode izlazne vrednosti koje koriste naredne instrukcije predstavljene pokretnim ili fiksnim čvorovima. Pokretni čvor poređenja sa uslovom (na primer, čvor poređenja na „manje od") predstavlja poređenje dve vrednosti koje do njega stižu putem pokretnih čvorova i bivaju upoređene u okviru tog čvora. Rezultat poređenja može se koristiti kao uslov grananja i proslediti fiksnom čvoru koji odgovara naredbi grananja, poput čvora `If`.

Neki od pokretnih čvorova prikazanih na slici 2.5 su pokretni čvorovi koji odgovaraju operaciji sabiranja, pokretni čvor koji odgovara poređenju sa uslovom, kao i pokretni čvorovi koji

[6]Ova metoda data je kao primer koji ilustruje jednostavnu petlju `for` i odgovarajuće konstrukte reprezentacije *Graal IR*. Iako se zbir prvih n prirodnih brojeva može izračunati formulom $n \cdot (n+1)/2$, sabiranje je ovde namerno implementirano pomoću petlje radi ilustracije.

```
1 private static int sum(int n) {
2     int s = 0;
3     for (int i = 0; i < n; i++) {
4         s += i;
5     }
6     return s;
7 }
```

Listing 1: Metoda koja u petlji sabira prvih n prirodnih brojeva



Slika 2.5: *Graal IR* graf i graf kontrole toka programa koji odgovaraju metodi `sum` sa prikaza koda 1

odgovaraju izboru vrednosti. Čvor izbora vrednosti (*Phi* čvor) se koristi na mestima gde se spajaju različiti tokovi kontrole programa (na primer, nakon grananja ili provere uslova petlje). Čvor izbora vrednosti određuje vrednost promenljive u zavisnosti od puta kojim je izvršavanje programa do tog mesta došlo. Na primer, prvi čvor izbora vrednosti u bloku *B1* bira između konstante 0, koja se koristi kada se prvi put uđe u petlju, i vrednosti dobijene inkrementiranjem, tj. rezultata sabiranja prethodne vrednosti sa konstantom 1, koja se koristi prilikom narednih prolazaka kroz petlju.

Fiksni čvorovi predstavljaju operacije koje direktno utiču na tok izvršavanja programa, kao što su instrukcije grananja, petlje, pozivi funkcija, spajanja tokova izvršavanja i slično. Čvorovi spajanja (*merge* čvorovi) označavaju mesta u programu gde se više tokova izvršavanja objedinjuju u jedan. Početak petlje u *Graal IR* grafu označen je čvorom početka petlje (čvor *LoopBegin*), dok se izlazak iz petlje označava čvorovima kraja petlje (čvor *LoopEnd*) i izlaska iz petlje (čvor *LoopExit*), u zavisnosti od toga da li se prelazi na sledeću iteraciju petlje ili se petlja napušta.

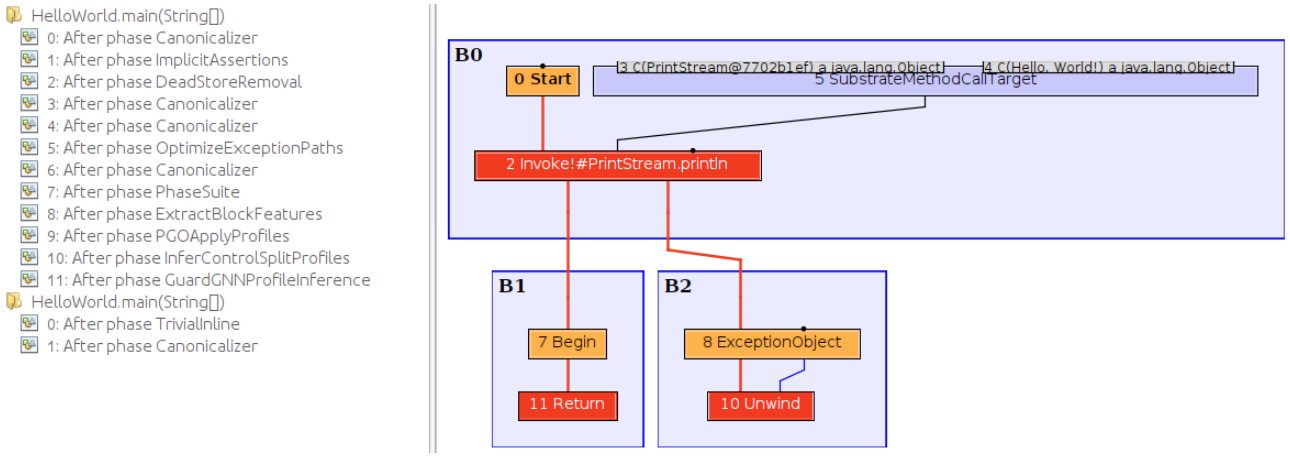
Alat *mx*

Kompilator *GraalVM Native Image* koristi alat *mx* za upravljanje projektom, izgradnju (engl. *build*), testiranje i pokretanje komponenti. Alat *mx* predstavlja napredni sistem za automatizaciju zadataka specifičan za platformu *GraalVM*. Alat *mx* integriše funkcionalnosti po-

put pokretanja test programa, upravljanja zavisnostima, konfiguracije okruženja i izvođenja prilagođenih skripti u toku razvoja i kompilacije programa. Komande alata *mx* značajno pojednostavljuju razvoj i testiranje unutar složene platforme *GraalVM* jer pružaju jedinstveno okruženje za upravljanje različitim fazama razvoja projekta.

Na primer, komanda `mx helloworld` pokreće kompilator *GraalVM Native Image* kako bi se kompilirao i kreirao izvršivi fajl jednostavnog programa koji na standardni izlaz ispisuje poruku *„Hello World“*. Komanda `mx build` koristi se za izgradnju celokupnog projekta, dok komanda `mx benchmark` omogućava pokretanje test programa (engl. *benchmarks*) za ispitivanje performansi kompilatora. Takođe, komanda `mx unittest` služi za izvršavanje testova i proveru korektnosti komponenti kompilatora.

Alat `mx` takođe omogućava pokretanje alata *Ideal Graph Visualiser* (*IGV*) [329, 64] za vizuelizaciju interne reprezentacije kompilatora *GraalVM Native Image*. Alat *IGV* je alat sa grafičkim korisničkim interfejsom koji omogućava vizuelno prikazivanje *Graal IR* grafova i grafova kontrole toka programa metoda tokom faza kompilacije. Na taj način alat *IGV* olakšava razumevanje uticaja optimizacionih faza na kompilaciju metoda. Na slici 2.6 prikazan je izlaz alata *IGV* koji prikazuje faze kompilacije metoda (levo), kao i *Graal IR* grafa metode i grafa kontrole toka programa (desno), za metodu koja na standardni izlaz ispisuje pozdravnu poruku *„Hello World!”*.



Slika 2.6: Grafički prikaz faza kompilacije i *Graal IR* grafa i grafa kontrole toka programa koji na standardni izlaz ispisuje poruku *„Hello World!”*

Proces kompilacije

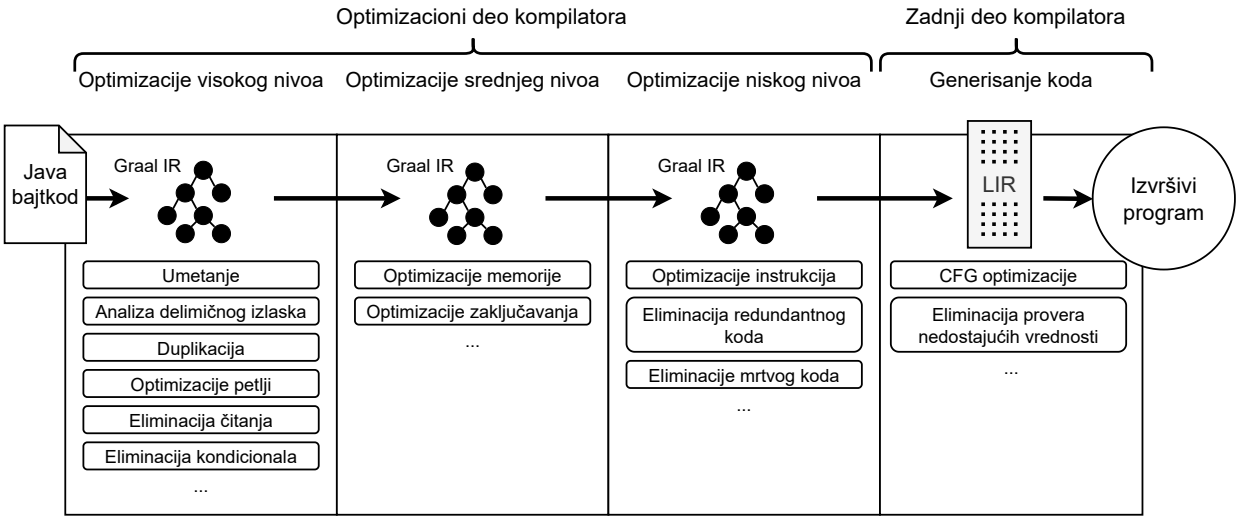
Standardni kompilatori sastoje se od prednjeg dela (engl. *frontend*), srednjeg dela (engl. *middle-end*) i zadnjeg dela (engl. *backend*). U prednjem delu kompilacije obavlja se leksička, sintaksna i semantička analiza, nakon čega se generiše odgovarajuća međureprezentacija. U srednjem, odnosno optimizacionom delu, vrše se optimizacije programa na nivou međureprezentacije koja je rezultat rada prednjeg dela kompilatora. U zadnjem delu, kompilator prevodi optimizovanu internu reprezentaciju programa na ciljnu platformu, pritom izvodeći dodatne optimizacije niskog nivoa, specifične za konkretnu platformu i arhitekturu.

Kompilator *GraalVM Native Image* koristi kompilator *Javac* za generisanje bajtkoda na osnovu ulaznog programa napisanog u nekom višem programskom jeziku. Na taj način, kompilator *GraalVM Native Image* preskače najveći deo prednjeg dela kompilacije, u kontekstu

standardne terminologije i vrši samo parsiranje bajtkoda i kreiranje *Graal IR* reprezentacije programa.

U srednjem delu kompilacije, kompilator *GraalVM Native Image* izvršava optimizacije na *Graal IR* reprezentaciji kroz tri nivoa optimizacija. Između svakog nivoa vrši se spuštanje (engl. *lowering*) koje uklanja apstrakcije iz prethodnog nivoa i priprema *Graal IR* graf za sledeći nivo optimizacija. Nakon slojeva optimizacije, u zadnjem delu kompilatora koristi se interna reprezentacija programa niskog nivoa (engl. *lowered intermediate representation*, LIR reprezentacija) za generisanje mašinskog koda odnosno izvršivog fajla.

Kompilator *GraalVM Native Image* organizuje optimizacije u faze kompilacije (engl. *compilation phases*). Svaka faza analizira *Graal IR* grafove metoda u potrazi za obrascima i primenjuje odgovarajuće optimizacije nad tim grafovima. Kompilator sadrži mnoge takve faze na svim nivoima kompilacije. Na slici 2.7 prikazana je šema kompilacije u okviru kompilatora *GraalVM Native Image*, zajedno sa nekim od ključnih optimizacija koje se sprovode u srednjoj i zadnjoj fazi kompilacije.



Slika 2.7: Šema kompilacije u kompilatoru *GraalVM Native Image*

Optimizacije visokog nivoa apstrakcije (engl. *high-tier optimizations*) uključuju optimizacije visokog nivoa bliske bajtkodu, poput umetanja (engl. *inlining*) [239], globalnog imenovanja promenljivih (engl. *global value numbering*) [58], sažimanja konstanti (engl. *constant folding*) [12], analiza izlaska (engl. *escape analysis*) [57, 24], analize delimičnog izlaska (engl. *partial escape analysis*) [284, 314], eliminacije čitanja (engl. *read elimination*), eliminacije kondicionala (engl. *conditional elimination*), duplikacije repa (engl. *tail replication*) [171, 173, 174] i slično.

Nakon optimizacija visokog nivoa interna reprezentacija programa se transformiše (engl. *lowering*) sa *Graal IR* reprezentacije koja odgovara bajtkodu na *Graal IR* reprezentaciju nešto nižeg nivoa, koja odgovara nivou mašinskih instrukcija, ali je i dalje nezavisna od platforme. Kompilator *GraalVM Native Image* u srednjem nivou apstrakcije (engl. *mid-tier*) izvodi optimizacije vezane za korišćenje memorije u Javi (engl. *Java memory semanctics optimizations*) poput optimizacija pristupa hip memoriji gde se pristupi hip memoriji modeluju kao pristupi memoriji sa adresnom semantikom. Dodatno, u srednjem nivou optimizacija kompilator izvodi optimizacije zaključavanja (engl. *lock optimizations*) i standardne optimizacije poput globalnog imenovanja promenljivih i sažimanja konstanti, koje su prisutne i među optimizacijama visokog nivoa.

Optimizacije niskog nivoa apstrakcije (engl. *low-tier optimizations*) izvode se nad *Graal IR* reprezentacijom niskog nivoa koje je specijalizovana za platformu i arhitekturu. Na ovom nivou se adresne operacije prilagođavaju odgovarajućoj arhitekturi, na primer arhitekturi *amd64* ili *SPARC* [135]. Ove optimizacije pripremaju *Graal IR* reprezenatciju niskog nivoa za transformisanje u LIR reprezentaciju iz koje se generiše finalni izvršivi program. Generisanje izvršivog programa kompilator *GraalVM Native Image* izvodi u svom zadnjem delu, na osnovu LIR reprezentacije, sprovodeći optimizacije poput optimizacija grafa kontrole toka ili eliminacije provera nedostajućih vrednosti (engl. *null check removal*).

2.3 Tehnike profajliranja

Profajliranje programa [345, 260] (engl. *program profiling*) je tehnika kojom se prikuplja profil izvršavanja programa. Profil izvršavanja programa je skup informacija o načinu izvršavanja programa. Na primer, profil izvršavanja programa može sadržati broj izvršavanja grana naredbi grananja, brojeve poziva metoda, frekvencije izvršavanja blokova u grafu kontrole toka programa, informacije o izvršenim implementacijama virtuelnih metoda i broj tih izvršavanja, informacije o otključavanju i zaključavanju katanaca i monitora i slično [44].

Kvalitetan profil precizno oslikava ponašanje programa tokom izvršavanja i odlikuju se visokom pokrivenosti koda i visokom preciznosti. To znači da kvalitetan profil sadrži precizne informacije o izvršavanju najvećeg dela koda. Postoje dva osnovna načina profajliranja: dinamičko i statičko profajliranje.

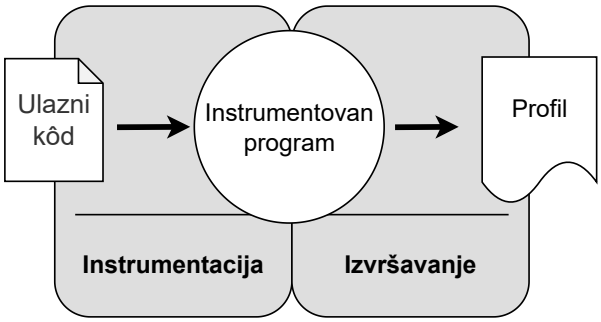
Dinamičko profajliranje

Dinamičko profajliranje (engl. *dynamic profiling*) [124] je standardni način profajliranja i odnosi se na prikupljanje profila prilikom izvršavanja programa. Dva osnovna tipa dinamičkih profajlera su profajleri zasnovani na instrumentaciji (engl. *instrumentation-based profilers*) [285] i profajleri zasnovani na uzorkovanju (engl. *sampling-based profilers*) [216]. Profajleri zasnovani na instrumentaciji prikupljaju kvalitetnije profile od profajlera zasnovanih na uzorkovanju. Profajleri zasnovani na uzorkovanju nude kompromis: oni prikupljaju profile nešto lošijeg kvaliteta od profajlera zasnovanih na instrumentaciji, ali smanjuju prostorne i vremenske zahteve profajliranja.

Profajleri zasnovani na instrumentaciji

Profajleri zasnovani na instrumentaciji od ulaznog programa kreiraju instrumentovan program. Ulazni program može biti u formi izvornog koda, programa na nekoj od međureprezentacija kompilatora, objektnog koda ili izvršivog koda. Da bi se kreirao instrumentovan program profajleri zasnovani na instrumentaciji tipično proširuju program kodom za prikupljanje i čuvanje informacija o izvršavanju programa. Nakon toga, izvršava se instrumentovan program i prikuplja se profil izvršavanja programa. Profajleri zasnovani na instrumentaciji ilustrovani su na slici 2.8.

Valgrind [211] je široko korišćena platforma za implementaciju profajlera zasnovanih na dinamičkoj binarnoj instrumentaciji. *Memcheck* [303] je najpoznatiji alat u okviru platforme *Valgrind*, namenjen otkrivanju grešaka u upravljanju memorijom kao što su curenja memorije, korišćenje neinicijalizovane memorije i ilegalni pristupi memoriji. Pored alata *Memcheck*, platforma *Valgrind* nudi i specijalizovane alate kao što su *Callgrind* [302] i *Cachegrind* [301].



Slika 2.8: Proces prikupljanja profila izvršavanja programa profajlera zasnovanih na instrumentaciji

Callgrind se fokusira na prikupljanje podataka pozivima funkcija, dok *Cachegrind* simulira ponašanje keš memorije procesora, što omogućava identifikaciju uskih grla u performansama vezanih za keširanje. Pored ovih alata, platforma Valgrind nudi i druge alate za instrumentaciono profajliranje različitih tipova profila.

Pored Valgrinda i moderni Java i *.NET* profajleri *JProfiler* [55], *NetBeans* [315, 206], *YourKit* [336] i *Honest* [220] podržavaju instrumentaciono profajlranje. Slično, profajleri namenjeni jezicima poput jezika C i C++, kao što je *gprof* [238], takođe podržavaju instrumentaciono profajliranje. Oni mogu funkcionisati kao samostalne aplikacije [55, 220] ili mogu biti integrisani u radno okruženje za razvoj aplikacija poput alata InteliJ [156] i NetBeans [209, 143]. Takođe i kompilatorske infrastrukure poput infrastrukture LLVM [167] i platforme poput platforme *GraalVM* [320] implementiraju instrumentaciono profajliranje. Kompilator *GraalVM Native Image* koristi instrumentaciono profajliranje da prikupi bojeve izvršavanja grana naredbi grananja, brojeve poziva funkcija, informacije o virtuelnim pozivima, kao i informacije o otključavanju i zaključavanju monitora.

Instrumentacija i uticaj instrumentacije

Instrumentacija programa je proširivanje programa kodom koji služi za prikupljanje i čuvanje profila izvršavanja programa. Na primer, instrumentacija uključuje dodavanje brojača (engl. *counters*) u program koji će pamtiti koliko puta se izvršio koji deo programa. Da bi se profajlirale naredbe grananja, najjednostavniji način je da se na početku svake grane doda po brojač koji će se inkrementirati svaki put kada se izvrši ta grana u programu. Međutim, predloženo rešenje nije efikasno jer bi na ovaj način neki brojači u kodu bili redundantni. Obzirom da brojači i njihovo inkrementiranje troše resurse, prilikom implementacije instrumentacije koriste se napredni algoritmi za minimizaciu broja potrebnih brojača. Primer takvog algoritma je Knutov[7] algoritam [151].

Knutov algoritam za profajliranje grana na osnovu grafa kontrole toka programa kreira minimalno razapinjuće stablo (engl. *minimal cost spanning tree*) [114, 113] a zatim svim granama koje ne pripadaju kreiranom stablu dodaje brojače. Svaki put kada se izvrši neka od instrumentovanih grana, vrednost brojača se uveća. Na osnovu broja poseta svake od instrumentovanih grana računa se broj poseta svake od grana u programu.

Knutov algoritam za profajliranje grana [151] samo je jedan od primera algoritama koji se koriste u instrumentaciji. Svi ovi algoritmi povećavaju kompleksnost, kao i vremenske i memo-

[7]Donald E. Knuth

rijske zahteve tokom procesa kompilacije. Na primer, najčešće korišćeni algoritmi za izračunavanje minimalnog razapinjućeg stabla su Primov[8] algoritam [7], koji uz korišćenje odgovarajućih struktura podataka može da postigne vremensku složenost od $O(|E| \log |V|)$, i Kraskelov[9] algoritam [146] sa istom složenošću, gde je $|V|$ broj čvorova, a $|E|$ broj grana u grafu. Napredniji pristupi, poput Karger[10]–Klajn[11]–Tarjan[12] algoritma [142, 45], uspevaju da smanje složenost na skoro linearnu — $O(|E|\,\alpha(|E|, |V|))$, gde je α inverzna Akermanova funkcija [266], koja u praksi ima vrlo male vrednosti. I pored ovih optimizacija, sama instrumentacija i dalje može imati značajan uticaj na vreme prevođenja programa, posebno kod velikih ulaznih grafova.

Prikupljanje profila izvršavanjem instrumentovanog programa traje značajno duže i troši više memorije od izvršavanja odgovarajućeg programa bez instrumentacije. U zavisnosti od same aplikacije, izvršavanje instrumentovanog programa može trajati 20% [185], 105% [14, 317] ili čak 22.2 puta duže [210] od izvršavanja odgovarajućeg optimizovanog programa. Usled toga instrumentaciju i izvršavanje instrumentovanog progama nekada nije moguće izvesti. Na primer, u slučaju sistema sa ugrađenim računarima (engl. *embedded systems*) [126, 128] često postoje striktna memorijska ograničenja [233] koja ne dozvoljavaju povećanja koja instrumentacija nameće, dok za sisteme za rad u realnom vremenu instrumentacija može da uspori izvršavanje programa i na taj način učini program neupotrebljivim [310].

Smanjenje troškova instrumentacije može se postići naizmeničnim smenjivanjem izvršavanja instrumentovanog i neinstrumentovanog programa. Ova vrsta profajliranja naziva se i *instant* profajliranje (engl. *instant profilers*) [56]. Instant profajliranje je posebno korisno u slučaju velikih aplikacija, na primer centara za obradu podataka (engl. *data centers*) gde smanjuje usporenje programa prilikom prikupljanja profila na 6% [56].

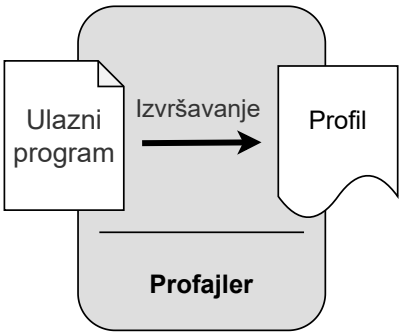
Profajleri zasnovani na uzorkovanju

Alternativa profajlerima zasnovanim na instrumentaciji su profajleri zasnovani na uzorkovanju [216]. Ovi profajleri ne zahtevaju posebno prevođenje zarad kreiranja instrumentovane verzije programa čime se pojednostavljuje proces profajliranja. Umesto toga, oni vrše prekide izvršavanja programa u određenim vremeniskim intervalima sa ciljem prikupljanja informacija na osnovu stanja u kom se program nađe u trenutku prekida. Time se smanjuju troškovi izvršavanja programa koji se profajlira, ali se time dobija i manje precizan profil. Slika 2.9 prikazuje proces prikupljanja profila korišćenjem profajlera zasnovanog na uzorkovanju.

Profajler *perf* [232] jedan je od najpoznatijih profajlera zasnovanih na uzorkovanju. Omogućava efikasno profajliranje zasnovano na brojačima ugrađenim u procesore. Pored profajlera *perf*, opšte poznati profajleri koji podržavaju profajliranje uzorkovanjem jesu profajleri *vtune* [244, 298] i *gprof* [238], prvi razvijen u okviru kompanije *Intel*, a drugi kao deo *GNU* zajednice. Pored toga, profajleri poput profajlera *Honest* [220] i *JProfiler* [55] pored profajliranja zasnovanog na instrumentaciji implementiraju i profajliranje zasnovano na uzorkovanju.

Guglovo široko profajliranje (engl. *Google-wide profiling*) [245] predstavlja infrastrukturu zasnovanu na profajliranju uzorkovanjem, namenjenu profajliranju centara za obradu podataka i aplikacija u oblaku (engl. *cloud applications*). Ovaj alat je skalabilan i ima minimalan uticaj na performanse, prikupljajući pritom stabilne i precizne profile. Slično, profajleri zasnovani na

[8] Robert C. Prim III
[9] Joseph B. Kruskal
[10] David R. Karger
[11] Philip Klein
[12] Robert E. Tarjan



Slika 2.9: Proces prikupljanja profila izvršavanja programa korišćenjem profajlera zasnovanih na uzorkovanju

uzorkovanju mogu se efikasno koristiti i u produkcionim okruženjima, sa niskim usporenjem aplikacija između 2% i 4% [316].

Hardverski zasnovani profajleri (engl. *hardware-based profilers*) koriste hardverske brojače u procesorima da prikupe profil prilikom izvršavanja programa. Time se dodatno smanjuju troškovi prikupljanja profila u slučajevima kada procesor ima integrisane hardverske brojače. Hardverski zasnovani profajleri svode usporenje prikupljanja profila u opseg od 0.4% do 4.6% [61].

Hardverski brojači za prikupljanje podataka o izvršavanju programa omogućavaju prikupljanje visokokvalitetnih profila [216]. Na složenijim aplikacijama, kao što su obrada slika i videa i veb serveri za pretragu i procesiranje logova, profajleri zasnovani na ovim hardverskim brojačima mogu prikupiti profile koji, kada se iskoriste za odgovarajuće optimizacije, dovode do poboljšanja performansi programa kompiliranih kompilatorom *GCC* [97] i do 30%.

Mane dinamičkih profajlera

Vremenska i memorijska zahtevnost dinamičkog profajliranja dolazi posebno do izražaja u kontekstu modernog razvoja softvera koji uključuje kontinuiranu integraciju (engl. *continuous integration*) [98, 197] i kontinuirano isporučivanje (engl. *continuous deployment*) [270, 48, 49]. Kontinuirana integracija i kontinuirano isporučivanje podrazumevaju konstantno integrisanje novih promena u kôd, izvršavanje odgovarajućih testova i isporučivanje programa krajnjim korisnicima. U slučajevima kada kreiranje stabilne i optimizovane verzije programa uključuje profajliranje, potrošnja resursa još više dobija na značaju. Računska zahtevnost profajliranja posebno je izražena u slučaju velikih aplikacija koje se dugo izvršavaju, poput centara za obradu podataka [228, 270].

Pored povećane kompleksnosti i povećanih vremenskih i memorijskih zahteva kompilacije, dinamički profajleri imaju još nedostataka. Odabir kvalitetnih ulaza za izvršavanje programa za testiranje veoma je važan jer izvršavanje programa u velikoj meri zavisi od ulaznih podataka sa kojima se taj program pokreće. Promenom ulaza, menjaju se i tokovi izvršavanja, odnosno odabir grana u programu. Na primer, u slučaju osnovne verzije algoritama sortiranja izborom (engl. *selection sort*) [150], broj izvršavanja spoljašnje petlje direktno zavisi od veličine ulaznog niza koji se sortira. Slično, kod algoritama poput sortiranja spajanjem (engl. *merge sort*) izvršavanje programa ne zavisi samo od dužine ulaznog niza niza, već i od rasporeda njegovih elemenata. Gotovo sortiran niz zahteva minimalnu obradu, dok obrnut redosled elemenata u ulaznom nizu može značajno povećati broj operacija. Zbog toga je važno da se prilikom pokre-

tanja programa koriste ulazi koji odražavaju tipičan način upotrebe aplikacija i verno simuliraju scenarije upotrebe koji se zaista javljaju u praksi. Prema tome, jedan od velikih izazova dinamičkih profajlera jeste pronalaženje informativnog skupa ulaza sa kojima će biti prikupljan profil. Ovo posebno dolazi do izražaja u kontekstu velikih aplikacija, obzirom da broj mogućih slučajeva upotrebe raste eksponencijalno sa povećanjem samog programa.

Statičko profajliranje

Da bi se prevazišli izazovi dinamičkog profajliranja, potrebno je promeniti pristup profajliranju i umesto prikupljanja profila izvršavanja programa, iste predvideti, bez prokretanja samog programa. Statički profajleri [326] su alati koji statički predviđaju verovatnoće izvršavanja grana u programu. Iako je moguće koristiti i širu definiciju, prema kojoj su statički profajleri alati za statičko predviđanje bilo koje od karakteristika izvršavanja programa, u ovoj disertaciji zadržaćemo se na originalnoj definiciji, budući da je ona široko prihvaćena u literaturi [326, 254, 202].

Statički profajleri definišu skup atributa kojima opisuju grane naredbi grananja, na osnovu kojih predviđaju verovatnoće njihovog izvršavanja. Na taj način statički profajleri profil izvršavanja programa generiše statički, bez izvršavanja samog programa.

Statičko profajliranje ne treba mešati sa statičkim predviđanjem izvršavanja grana. Statičko predviđanje grana u programu (engl. *static branch prediction*), ili skraćeno predviđanje grana, odnosi se na predviđanje koje će se grane nakon naredbi grananja u programu verovatno izvršiti, a koje neće — dakle, na zadatak klasifikacije grana na one koje će biti i one koje neće biti izvršene. S druge strane, statički profajleri ne donose binarne odluke, već predviđaju verovatnoće izvršavanja svake od grana.

Statičko profajliranje zasnovano na heuristikma

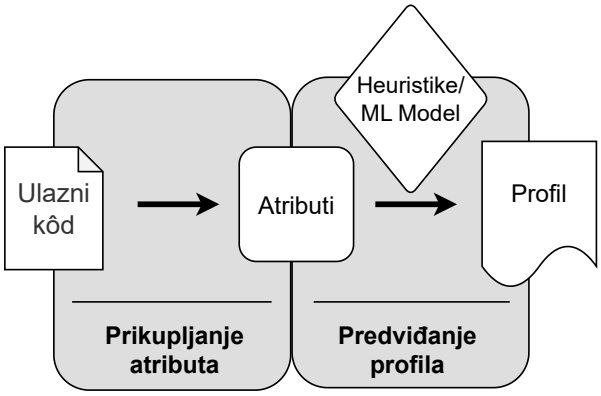
Još tokom devedesetih godina prošlog veka uočeno je da se na osnovu samog ulaznog koda programa mogu izvesti značajni zaključci o načinu na koji će se taj program izvršavati [13]. Na primer, u petljama se povratna grana obično izvršava mnogo češće nego grana koja vodi van petlje. Slično, grane koje odgovaraju obradama izuzetnih situacija (engl. *exception handlers*) obično se retko izvršavaju. Ovakva opažanja dovela su do definisanja skupa heuristika koje su korišćene za predviđanje profila izvršavanja programa. Iako je skup korišćenih heuristika bio relativno mali, njihova primena za predviđanje profila u kombinaciji sa optimizacijama vođenim tim profilom mogla je značajno da poboljša performanse programa [78].

Statičko profajliranje zasnovano na mašinskom učenju

Osnovni nedostatak heuristika za statičko predviđanje profila je što su one ograničene na opisivanje jednostavnih pravila. Na ovaj način nije moguće uhvatiti kompleksne zavisnosti koje postoje u ulaznom kodu programa. Modeli mašinskog učenja [139] nametnuli su se kao naslednik heuristika za predviđanje profila i prirodno unapređenje statičkih profajlera. Zahvaljujući izražajnosti modela mašinskog učenja i sposobnosti otkrivanja najsuptilnijih zakonitosti u podacima, modeli mašinskog učenja omogućavaju kvalitetnije predviđanje profila izvršavanja programa od statičkih heuristika.

Na slici 2.10 prikazani su koraci predviđanja profila statičkih profajlera. Statički profajleri u prvom koraku izdvajaju atribute (engl. *feature extraction*) kojima se opisuju delovi koda.

Ovo izdvajanje atributa znatno je jeftinije nego dinamičko prikupljanje profila. U drugom koraku, statički profajleri na osnovu izdvojenih atributa predviđaju profil izvršavanja programa heuristikama ili modelima mašinskog učenja.



Slika 2.10: Proces predviđanja profila izvršavanja programa statičkih profajlera

Prvi radovi koji su kombinovali mašinsko učenje i statičko predviđanje profila fokusirali su se na predviđanje verovatnoća izvršavanja različitih putanja u kodu [94, 13, 37]. Korišćeni su klasifikatori poput stabala odlučivanja (engl. *decision tree*) [255, 81], logističke regresije (engl. *logistic regression*) [325, 36] i potpuno-povezanih dubokih neuronskih mreža (engl. *fully-connected feed-forward deep neural networks*) [292, 169, 212, 37]. S razvojem oblasti, uvedeni su novi klasifikacioni modeli, kao što su metoda k najbližih suseda (engl. *k-nearest neighbors algorithm*) [144, 288] i rekurentne neuronske mreže (engl. *recurrent neural networks*) [195, 115, 340]. Pored toga, rešavani su i drugi problemi, poput predviđanja faktora odmotavanja petlji (engl. *unroll factor*) [288].

U kontekstu statičkog predviđanja profila, klasifikacija ima ograničenja jer se putanje odnosno grane u kodu često svrstavaju u dve kategorije: kao izvršena putanja odnosno grana (engl. *taken*) ili kao putanja odnosno grana koja nije izvršena (engl. *not-taken*). Profil izvršavanja programa predstavlja kontinualne vrednosti, na primer, za grane naredbi grananja profil sadrži informaciju o broju koliko je puta ta grana izvršena. Stoga klasifikacija u dve kategorije vodi do gubitka informacija. Zato se u novije vreme pojavljuju radovi koji koriste regresiju za predviđanje verovatnoća izvršavanja grana [202].

Što se tiče reprezentacije programa, atribute koji opisuju delove koda statički profajleri mogu izdvajati iz različitih izvora — od grafovskih međureprezentacija programa [340] do izvršivog koda programa [202, 228].

Odnos dinamičkog i statičkog profajliranja

Statički profajleri značajno su jeftiniji od dinamičkih profajlera, obzirom da izbegavaju vremenski i memorijski skupu instrumentaciju, uzorkovanje i prikupljanje profila. Kako statički profajleri eliminišu potrebu za komplikovanim procesom dinamičkog prikupljanja profila koji, pored kompilacije optimizovanog programa zahteva i izvršavanje programa radi prikupljanja profila, statički profajleri se znatno jednostavnije integrišu u kontekst modernog razvoja softvera koji se konstantno integriše i isporučuje. Pored toga, statički profajleri prevazilaze i problem pronalaska ulaznih programa za prikupljanje profila, obzirom da ne uključuju korak prikupljanja profila.

Ipak, statički profajleri predviđaju profil nešto lošijeg kvaliteta od dinamičkih profajlera. Jedan od osnovnih razloga za to je činjenica da statički profajleri na raspolaganju imaju manju količinu informacija, tj. ne uzimaju u obzir ulazne podatke sa kojima se izvršavaju aplikacije. Ove informacije nedostupne su statičkim profajlerima po njihovom dizajnu jer su to informacije iz izvršavanja programa (engl. *runtime*), dok statički profajleri rade u vreme kompilacije. Prema tome, statički profajleri predstavljaju kompromis u odnosu na dinamičke profajlere nudeći profile lošijeg kvaliteta po značajno manjoj ceni.

2.4 Optimizacije vođene profilom

Performanse prevedenih programa u značajnoj meri zavise od vrste i kvaliteta optimizacija koje omogućava kompilator koji taj program prevodi. Jedna od najvažnijih klasa optimizacija su optimizacije vođene profilom (engl. *profile-guided optimizations*) [118, 321].

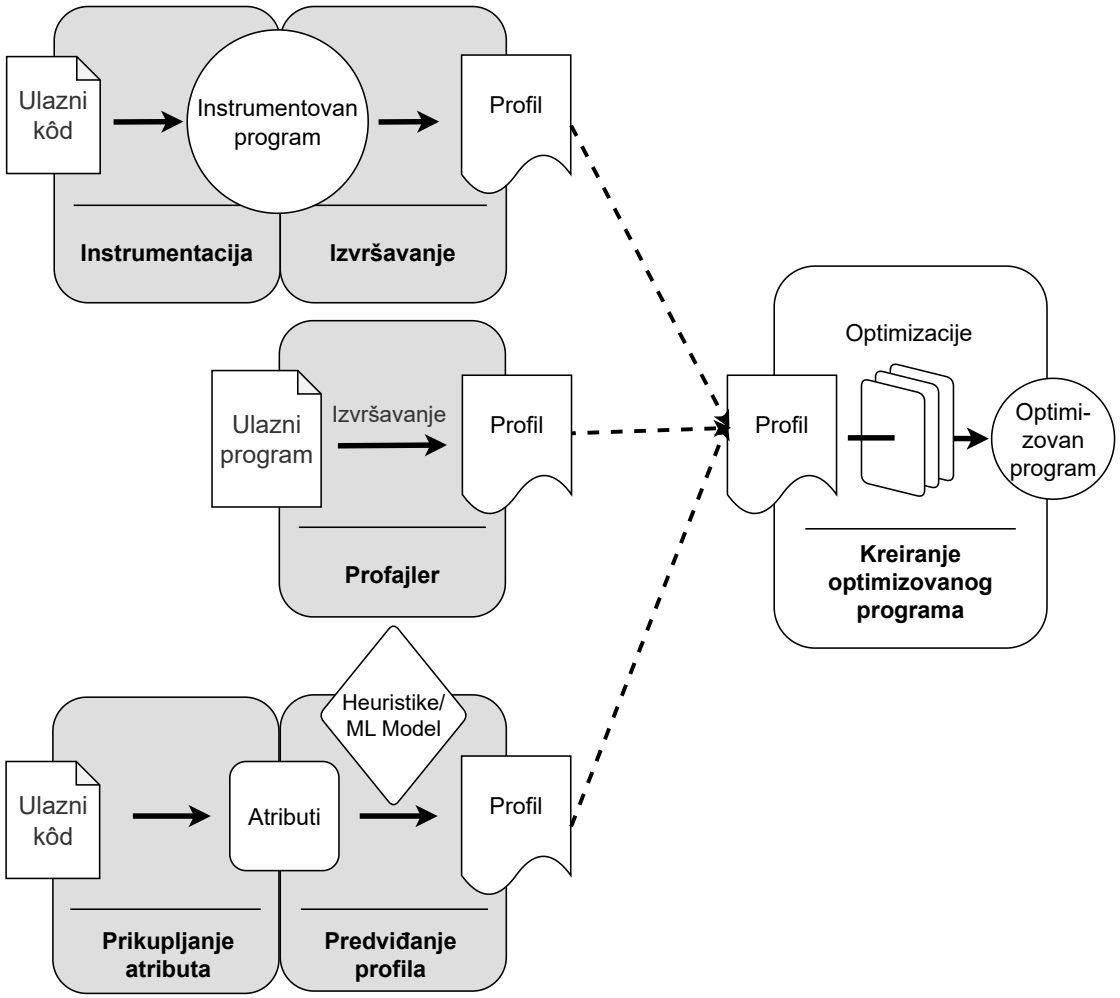
U klasu optimizacija vođenih profilom spadaju sve optimizacije koje koriste profil izvršavanja programa. Na slici 2.11 ilustrovan je proces kreiranja optimizovanih programa u slučaju optimizacija vođenih profilom. Da bi kompilator primenio optimizacije vođene profilom, prethodno je potrebno na neki način generisati njegov profil izvršavanja. Profil izvršavanja programa može biti prikupljen nekom od tehnika dinamičkog profajliranja, ili statički predviđen. Nakon toga, optimizacije vođene profilom koriste generisan profil da optimizuju programe.

Primeri optimizacija vođenih profilom

Korišćenjem informacija o tome koji delovi koda se češće izvršavati, kompilator može primeniti agresivne optimizacije kao što su umetanje (engl. *inlining*) [10, 42], duplikacije repa [171, 176], pozicioniranje koda [235], optimizacije keša [261], optimizacija petlje [12] i analizu delimičnog izlaska (engl. *partial escape analysis*) [284]. Ove tehnike mogu značajno unaprediti performanse programa. Za razliku od statičkih optimizacija, koje se oslanjaju samo na analizu izvornog, prekompajliranog koda ili interne reprezentacije programa, optimizacije vođene profilom koriste dodatne informacije o tome na koji način će se program najverovatnije izvršavati [216, 228].

Optimizacije zasnovane na duplikaciji mogu značajno unaprediti performanse programa, ali dolaze uz cenu povećanja veličine izvršivog programa i dužeg vremena kompilacije. Zato kompilatori moraju pažljivo vagati koristi i troškove ovakvih optimizacija, procenjujući njihov uticaj na performanse izvršavanja programa i veličinu izvršivog fajla. Iako se odluke o duplikaciji često oslanjaju na statičke heuristike, one su ograničene jer ne uzimaju u obzir ponašanje programa pri izvršavanju. Profil izvršavanja programa igra ključnu ulogu u vođenju odluka o duplikaciji. Pružajući informacije o učestalosti izvršavanja različitih grana i broju izvršavanja metoda, oni omogućavaju identifikaciju onih mesta u kodu gde će duplikacija imati najveći uticaj, a da se pri tom izbegne nepotrebno povećanje veličine izvršivih fajlova.

Umetanje funkcija [295, 10, 42] je jedna od ključnih optimizacija u kompilatorima, kojom se pozivi funkcija zamenjuju njihovim telima direktno na mestima poziva. Time se smanjuje trošak poziva funkcije, ali se istovremeno otvara prostor za dodatne unutarproceduralne optimizacije (engl. *interprocedural optimizations*) [247, 35, 62]. Umetanje može omogućiti optimizacije poput eliminacije mrtvog koda (engl. *dead code elimination*) i time dovesti do smanjenja veličine optimizovanih programa. Ipak, ako se preterano koristi, ono može dovesti i do značajnog rasta veličine izvršivog fajla.



Slika 2.11: Kreiranje optimizovanog programa na osnovu profila izvršavanja programa (desno). Profil izvršavanja programa može biti dinamički prikupljen ili statički predviđen (levo).

Donošenje odluka o tome gde umetnuti funkcije predstavlja složen zadatak jer ne zavisi samo od karakteristika konkretne funkcije, već i od toga kako će to umetanje uticati na ostatak optimizacija koje slede u lancu kompilacije. Sam problem izbora koje funkcije umetnuti, a koje ne, spada u NP-kompletne probleme i može se posmatrati kao varijanta problema ranca (engl. *knapsack problem*) [264]. Razvijene su razne heuristike koje procenjuju karakteristike funkcija, kao što su broj potrebnih instrukcija procesora za izvršavanje funkcije, kontekst mesta poziva ili procena uticaja na vreme kompilacije. Slično, razvijeniji su i napredniji pristupi poput onih zasnovanih na probama (engl. *inlining and trials*) [77] koji omogućavaju kompilatorima da privremeno umetnu funkcije, procene njihov uticaj i ponište promene ako je potrebno. Profil izvršavanja omogućava umetanje često pozivanih funkcija i izbegavanje umetanja retko pozivanih funkcija, čime se postiže bolji balans između performansi i veličine izvršivog fajla.

Duplikacija repa [171, 44], često nazivana i replikacija repa [204, 205], predstavlja optimizaciju kompilatora koja uklanja kôd nakon spajanja više putanja u programu i kopira ga u spojene putanje. Ova tehnika omogućava kompilatoru da specijalizuje duplirani kôd u skladu sa tipovima i vrednostima koje se koriste u putanjama koje se spajaju. Specijalizacija koda može otvoriti dodatne mogućnosti za optimizaciju, zbog čega je neophodna sofisticirana strategija

duplikacije koja će izabrati one duplikacije koje donose maksimalnu korist. Profil izvršavanja programa igra ključnu ulogu u ovom procesu jer pruža informacije o frekvenciji i obrascima izvršavanja, omogućavajući identifikaciju delova koda gde će duplikacija najviše doprineti poboljšanju performansi.

Još jedna optimizacija koja može koristiti profil izvršavanja programa jeste odmotavanje petlje (engl. *loop unrolling*) [76, 130]. Odmotavanje petlje podrazumeva dupliranje tela petlje kako bi se smanjio broj iteracija i kontrolnih prelaza, što može poboljšati iskorišćenost procesorskih resursa i smanjiti grananje. Međutim, agresivno odmotavanje može dovesti do značajnog povećanja veličine koda, pa informacije iz profila pomažu u identifikaciji najčešće izvršavanih petlji koje su dobri kandidati za ovu optimizaciju, dok se izbegava bespotrebno uvećanje veličine optimizovanih programa.

Optimizacija pozicioniranja koda (engl. *code positioning*) [235] odnosi se na pozicioniranje delova programa, funkcija i instrukcija u memoriju na način da se poboljša lokalnost informacija (engl. *information locality*), smanji kašnjenje memorije (engl. *memory latency*) i smanje promašaji u kešu[13]. Pozicioniranje koda koristi profil izvršavanja programa da grupiše delove koda kojima će biti pristupano istovremeno, time smanjujući promašaje u kešu i smanjujući kašnjenje uzrokovano neefikasnim čitanjem iz memorije. Ovo je posebno važno u modernim arhitekturama, gde brzine procesora značajno nadmašuju brzine memorije, a loša lokalnost može postati glavno usko grlo u performansama.

Uticaj profila na optimizacije

Kvalitet profila izvršavanja programa presudan je za rezultate optimizacija vođenih profilom. Kvalitetan profil obezbeđuje precizne informacije o izvršavanju najvećeg dela programa, što omogućava kompilatoru da tačno predvidi koje optimizacije će doneti najviše koristi. Na primer, ukoliko profil pokazuje da će se određena putanja u programu često izvršavati, kompilator može umetnuti pozive funkcija sa te putanje. Na taj način se smanjuju troškovi poziva funkcija koje se često izvršavaju, što rezultuje poboljšavanjem performansi izvršavanja programa.

Sa druge strane, nekvalitetan profil može dovesti do pogrešnih odluka prilikom primene optimizacija, što može rezultirati smanjenjem performansi programa. Na primer, ako profil pogrešno označi retko izvršavane putanje kao frekventne, kompilator može nepotrebno uložiti resurse u umetanje funkcija sa tih putanja, zanemarujući one koje su zaista važne. To može dovesti ili do znatnog povećanja veličine izvršivog fajla zbog umetanja suvišnih funkcija, ili do propuštanja prilike da se optimizuju najfrekventnije metode i time poboljšaju performanse optimizovanog programa.

U slučaju profila izvršavanja programa dobijenog profajliranjem zasnovanim na uzorkovanju, veliki izazov predstavlja i preslikavanje tog profila iz izvršivog koda nazad u različite delove lanca kompilacije i omogućavanje primene optimizacija. Profil sakupljeni prilikom izvršavanja programa može se preslikati u različite delove lanca kompilacije: u vreme same kompilacije (engl. *compile time*) [46], pri linkovanju (engl. *link time*) [178, 226] ili nakon linkovanja (engl. *post-link time*) [187, 228].

Odabir faze lanca kompilacije u koju će se profil preslikati predstavlja kompromis između njihove preciznosti i broja optimizacija koje mogu da koriste profil. Profil dostupan u ranim fazama kompilacije može se koristiti u većem broju optimizacija, ali je obično manje precizan.

[13]Keš memorije pomažu u prevazilaženju vremenskog jaza između brzih mikroprocesora i relativno spore memorije računara [168, 53]. Čuvajući nedavno korišćene delove memorije, keš memorije smanjuju broj ciklusa tokom kojih procesor mora da čeka na podatke. Kako razlika u brzini ciklusa između procesora i glavne memorije raste – za više od 40% godišnje – efikasna upotreba keš memorije postaje sve važnija.

S druge strane, profil preslikan bliže izvršivom kodu nudi veću preciznost, ali se može primeniti u manjem broju optimizacionih faza [228].

Glava 3

Statički profajleri

U ovoj sekciji dat je pregled radova koji se bave statičkim predviđanjem grana, kao i prikaz relevantnih statičkih profajlera. Statičko predviđanje grana predstavlja zadatak binarne klasifikacije i odnosi se na predviđanje koje će se grane naredbi grananja u programu češće izvršavati. S druge strane, statički profajleri predviđaju verovatnoće izvršavanja grana.

3.1 Statičko predviđanje grana

Fišer[1] i Frojdenberger[2] [94] su još 1992. godine pokazali da je moguće predvideti koje će grane naredbi grananja biti najčešće izvršavane u programu. Pokazali su da se, čak i u složenim aplikacijama poput jezičkih procesora (engl. *language processors*) i sistemskih biblioteka (engl. *system utilities*), nakon većine naredbi grananja u programu najčešće izvršava samo jedna od grana. Kao primer, može se uzeti obrada izuzetaka u kodu. Grana predviđena za obradu izuzetka je najčešće retko posećena, na primer, samo u slučajevima kada nije moguće alocirati resurse zbog nedostatka memorije na računaru. Zaključci iz ovog rada motivisali su istraživanja u oblasti statičkog predviđanja grana u programima.

Bal[3] i Larus [13] su 1993. godine razvili statički prediktor grana zasnovan na karakteristikama programa (engl. *program-based branch predictor*). Definisali su skup od sedam jednostavnih i efikasnih heuristika koje statički predviđaju izvršavanje grana direktno na osnovu izvršivog fajla programa. Definisane heuristike su motivisane intuicijom i posmatranjima načina na koje se programi izvršavaju, a opsane su u nastavku.

Heuristika petlje (engl. *loop heuristic*) predviđa izvršavanje tela petlje, pre nego njeno preskakanje. *Heuristika izlaska iz programa* (engl. *return heuristic*) predviđa da grane koje sadrže naredbu izlaska iz funkcije (engl. *return*) neće biti izvršene. Slično, *heuristika čuvanja u memoriji* (engl. *store heuristic*) predviđa da grane koje sadrže instrukciju čuvanja u memoriji (engl. *store instruction*) neće biti izvršene. *Heuristika zaštite registara* (engl. *guard heuristic*) predviđa izvršavanje grana koje odgovaraju uspešnim poređenjima u kojima je jedan od operanada registar.

Heuristika poziva (engl. *call heuristic*) predviđa da se grane u programu koje sadrže pozive metoda uglavnom neće izvršavati. Autori napominju da ova heuristika može delovati neintuitivno jer programi obično pozivaju metode da izvrše konkretne zadatke. Međutim, metode se

[1] Joseph A. Fisher
[2] Stefan M. Freudenberger
[3] Thomas Ball

često i pozivaju za obradu izuzetnih situacija, pa su takvi pozivi retki, zbog čega su Bal i Larus definisali heuristiku poziva baš na ovaj način.

Heuristika koda operacije (engl. *opcode heuristic*) koristi činjenicu da procesor *MIPS R2000* [74], kog su autori koristili prilikom pisanja rada, podržava instrukcije grananja zasnovane na poređenju vrednosti registra sa nulom. Kako negativne vrednosti često označavaju greške, ova heuristika predviđa da grane koje odgovaraju uspešnom poređenju vrednosti u registru sa *„manje od nule“* ili *„manje ili jednako nuli“* neće biti izvršene. Nasuprot tome, za grane koje odgovaraju uspešnom poređenju vrednosti u registru sa *„veće od nule“* ili *„veće ili jednako nuli“* predviđa se da će biti izvršene.

Poređenja pokazivača mogu biti ili poređenje dva pokazivača na jednakost ili poređenje pokazivača sa nedostajućom vrednosti (engl. *null pointer*). *Heuristika poređenja pokazivača* (engl. *pointer heuristic*) predviđa izvršavanje grana koje odgovaraju uspešnom poređenju dva pokazivača, a preskakanje grana koje odgovaraju poređenju pokazivača sa nedostajućom vrednosti jer su ovakve grane uglavnom vezane za obradu izuzetaka, i kao takve retko se izvršavaju.

U slučajevima kada se više heuristika može primeniti na istu granu, Bal i Larus koriste prvu primenjivu heuristiku, pri čemu su redosled heuristika odredili eksperimentalno. Za optimizaciju redosleda primena ručno definisanih heuristika i za otkrivanje novih heuristika za statičko predviđanje grana može se koristiti i model stabla odlučivanja [81].

Bal i Larus su evaluaciju uradili na programima napisanim u jezicima C i Fortran [11]. Autori prijavljuju grešku klasifikacije od 26% na test skupu koji se sastoji od 23 *Unix* programa, uključujući kompilatore poput kompilatora *GCC* [97] i sistemskih alata poput alata *compress*, *grep*, *dcg* i drugih. Slične heuristike se i danas podrazumevano koriste u kompilatoru *Clang* [165, 136, 202].

Ditrih[4] i ostali [78] unapredili su statičke heuristike za predviđanje grana Bala i Larusa, između ostalog, korišćenjem informacija iz izvornog koda programa, a ne samo iz izvršivog fajla programa, kako su ove heuristike izvorno definisane. Predloženo rešenje integrisali su u kompilator *IMPACT* [43], i ocenjivali na skupovima test programa *SPEC CINT92* i *SPEC CINT95* [86, 40]. Pored Ditriha i ostalih i drugi radovi dizajnirali su statičke heuristike za predviđanje grana koje koriste atribute izdvojene iz izvornog koda programa [323].

Kalder[5] i ostali [37] su još 1997. godine razvijali modele mašinskog učenja za statičko predviđanje grana. U svom radu koristili su klasifikacione modele stabla odlučivanja i duboke neuronske mreže, koje su trenirali na osnovu atributa izdvojenih iz interne reprezentacije niskog nivoa kompilatora jezika Fortran i C. Njihova metoda predviđanja grana nazvana *ESP* (engl. *evidence-based static branch prediction*) ostvarila je grešku klasifikacije od 20%. Poređenja radi, najbolje tada dostupne heuristike za statičko predviđanje grana ostvarivale su grešku klasifikacije od 25%.

Shih[6] i ostali [272] izdvajaju atribute iz LLVM međureprezentacije i obučavaju pet klasičnih modela mašinskog učenja za statičko predviđanje grana: model stabla odlučivanja, metod k-najbližih suseda, slučajnu šumu (engl. *random forest*) [22], metod poptpornih vektora [287] i ansambl gradijentnog pojačavanja [18]. Pored toga, koristili su i model grafovske neuronske mreže [346, 328] dizajniran na osnovu interne grafovske međureprezentacije kompilatora *ProGraML* [71]. Autori pokazuju da model grafovske neuronske mreže postiže rezultate uporedive sa rezultatima klasičnih modela mašinskog učenja.

Heuristike za predviđanje grana bile su prve tehnike koje su omogućile statičko predviđanje

[4]Brian L. Deitrich
[5]Brad Calder
[6]Ching-Yen Shih

izvršavanja delova programa. Nakon njih, razvijeni su statički prediktori koji su predviđali i druge karakteristike programa, kao što su faktori odmotavanja petlji [288], frekventne putanje u programu [36, 340], frekventne metode [138, 190] i virtuelne pozive [341]. Pored toga, u novije vreme modeli mašinskog učenja korišćeni su i za odabir hardvera (engl. *heterogeneous compute device mappings*) i klasifikaciju algoritama [71].

3.2 Statički profajleri

Najsavremeniji statički profajleri koriste ručno definisane heuristike (engl. *hand-crafted heuristics*) [326], klasifikacione modele mašinskog učenja [254] i regresione modele dubokog učenja koji izdvajaju atribute iz binarnih odnosno izvršivih fajlova (engl. *program binary*) [202].

Statički profajler Vua i Larusa

Vu i Larus su [326] su definisali statički profajler koji koristi devet heuristika za predviđanje verovatnoća izvršavanja grana u programu. Oni su prilagodili heuristike iz rada Bala i Larusa, i definisali još dve nove heuristike. U slučaju heuristika Bala i Larusa heuristike su predviđale da li će grane biti izvršene. Vu i Larus su prilagodili te heuristike tako da predviđaju izvršavanje grana, ali sa određenim, eksperimentalno utvrđenim verovatnoćama izvršavanja. Verovatnoće izvršavanja grana u slučaju da se heuristika primenjuje na granu date su u tabeli 3.1.

Tabela 3.1: Heuristike za predviđanje verovatnoća izvršavanja grana u programu. Druga kolona odnosi se na verovatnoću izvršavanja (p) koja se dodeljuje grani ukoliko se heuristika primenjuje na granu.

Heuristika	p
Heuristika petlje (engl. *loop branch heuristic*)	0.88
Heuristika izlaska iz programa (engl. *return heuristic*)	0.72
Heuristika čuvanja u memoriji (engl. *store heuristic*)	0.55
Heuristika zaštite registara (engl. *guard heuristic*)	0.62
Heuristika poziva (engl. *call heuristic*)	0.78
Heuristika koda operacije (engl. *opcode heuristic*)	0.84
Heuristika poređenja pokazivača (engl. *pointer comparison heuristic*)	0.60
Heuristika početka petlje (engl. *loop header heuristic*)	0.75
Heuristika izlaska iz petlje (engl. *loop exit heuristic*)	0.80

Vu i Larus su definisali heuristiku početka petlje (engl. *loop header heuristic*) i heuristiku izlaska iz petlje (engl. *loop exit heuristic*). *Heuristika početka petlje* predviđa da će se izvršiti povratna grana naredbe grananja koja pripada petlji, odnosno grana koja vodi ka njenom početku. Sa druge strane, *heuristika izlaska iz petlje* predviđa da će grananja unutar tela petlje, čija jedna grana vodi van nje, najčešće ostati unutar petlje. Heuristika početka petlje zasniva se na grafu toka kontrole i predviđa izvršavanje bloka koji predstavlja početak petlje.

Vu i Larus predlažu korišćenje Dempster-Šafer teorije [111] kako bi prevazišli problem kombinovanja više heuristika. Svaka heuristika h, ukoliko je primenljiva, predviđa izabranu granu b sa unapred definisanom verovatnoćom $h(b)$. Kada su dve heuristike (h_1 i h_2) primenljive za granu b, one se kombinuju koristeći formulu:

$$(h_1 \oplus h_2)(b) = \frac{h_1(b) \cdot h_2(b)}{h_1(b) \cdot h_2(b) + (1 - h_1(b)) \cdot (1 - h_2(b))}.$$

Ova formula definisana je tako da favorizuje slučajeve kada obe heuristike snažno podržavaju istu odluku, i da vodi ka neutralnijem predviđanju u slučaju kada su predviđanja heuristika suprotstavljena. Na taj način se omogućava kombinovanje heuristika koje nagrađuje slaganje, a kažnjava neslaganje. Na primer, ako prva heuristika procenjuje verovatnoću izvršavanja grane b kao $h_1(b) = 0.9$, a druga kao $h_2(b) = 0.8$, njihovo kombinovanje daje:

$$(h_1 \oplus h_2)(b) = \frac{0.9 \cdot 0.8}{0.9 \cdot 0.8 + 0.1 \cdot 0.2} = \frac{0.72}{0.72 + 0.02} = \frac{0.72}{0.74} \approx 0.973.$$

Dakle, kombinovana verovatnoća je veća nego kod bilo koje pojedinačne heuristike, što odražava njihovu međusobnu saglasnost. S druge strane, ako se heuristike ne slažu (npr. $h_1(b) = 0.9$, $h_2(b) = 0.1$), rezultat kombinovanja je:

$$(h_1 \oplus h_2)(b) = \frac{0.9 \cdot 0.1}{0.9 \cdot 0.1 + 0.1 \cdot 0.9} = \frac{0.09}{0.18} = 0.5,$$

što vodi do neutralnog predviđanja verovatnoće izvršavanja grane.

Statički profajler Vua i Larusa ima široku primenu. Kompilator *GCC* [286] koristi heuristike slične onima koje su predložili Vu i Larus, uz dodatne heuristike prilagođene programskom jeziku C [5]. Na primer, jedna od njih jeste predviđanje da je povratna vrednost funkcije `malloc` skoro uvek različita od nule [103].

Kompilator za programski jezik Haskel (engl. *Haskell*) kompanije *Intel* (engl. *Intel Labs Haskell Research Compiler*) primenjuje heuristike Bala i Larusa za predviđanje profila kako bi poboljšao umetanje frekventih metoda [180]. Bogerd[7] i Monen[8] primenili su pet heuristika Vua i Larusa zajedno sa njihovom metodologijom za kombinovanje više heuristika da razviju alat *ELAN*, koji pomaže korisnicima da prioritizuju informacije prikupljene iz alata za inspekciju softvera (engl. *software inspection tool*) [92]. Sun[9] i ostali koriste heuristike Vua i Larusa za predviđanje verovatnoća izvršavanja grana tokom razvoja alata za statičku analizu *Lukewarm* [291].

Statički profajler Rotema i Kuminsa

Rotem i Kumins razvili su statički profajler zasnovan na ansamblu stabala odlučivanja za višeklasnu klasifikaciju. Kako bi kreirali označen skup podataka za obučavanje, oni su profajlirali naredbe grananja sa dve grane i prikupili podatke o broju izvršavanja njihovih grana. Naredbe grananja označavali su odnosom brojeva izvršavanja njenih grana. Prikupljene odnose brojeva izvršavanja grana autori su diskretizovali i svaku naredbu grananja označavali jednom od 11 kategorija, u zavisnosti od intervala kom pripada izračunata vrednost. Kategorije se kreću od 0.0 do 1.0, sa korakom od 0.1.

Rotem i Kumins koriste 54 atributa kojima opisuju naredbe grananja. Atribute definišu na nivou binarne reprezentacije programa. Atributi opisuju naredbu grananja, grane naredbe grananja ali i blok grafa kontrole toka u kome se nalazi naredba grananja. Neki od atributa koji opisuju blok u kome se nalazi naredba grananja su: broj instrukcija u bloku, broj poziva funkcija u bloku, broj instrukcija čitanja iz memorije (engl. *load instructions*), broj instrukcija pisanja u memoriju (engl. *store instructions*), broj blokova prethodnika u grafu kontrole toka

[7] Cathal Boogerd
[8] Leon Moonen
[9] Qiang Sun

(engl. *number of predecessors blocks*) i broj blokova sledbenika u grafu kontrole toka (engl. *number of successors blocks*). Autori naredbu grananja opisuju atributima kao što su dubina ugnežđenja u petljama (engl. *loop depth*) i broj blokova u grafu kontrole toka nad kojima naredba grananja ima dominaciju. Takođe je opisuju i prema tipu grananja, odnosno da li je grananje rezultat poređenja po jednakosti ili poređenja pokazivača. Dodatni atribut je i informacija o tome da li leva ili desna grana sadrži instrukciju za izlazak iz metode (engl. *exit instruction*).

Autori su obučili model *XGBoost* za višeklasnu klasifikaciju, koji postiže tačnost od 75%. Integracijom obučenog ansambla u kompilatorsku infrastrukturu LLVM [167] ostvareno je poboljšanje performansi na 6 od 10 testiranih programa. Najveće ubrzanje, od 16%, zabeleženo je kod programa napisanih u programskom jeziku Pajton, dok je smanjenje performansi od 7% uočeno na test programu *bzip2* [127].

Statički profajler Ramana i Lija

Raman i Li [242] su razvili statički profajler za predviđanje verovatnoća izvršavanja grana u velikim aplikacijama namenjenim centrima za obradu podataka. Profajler je razvijen za programski jezik *C++*, a atribute koji opisuju naredbe grananja autori izdvajaju iz interne reprezentacije kompilatorske infrastrukture *LLVM*. Raman i Li koriste nekoliko grupa atributa kojima opisuju naredbe grananja: atribute koji opisuju tok podataka u programu, atribute koji opisuju kontrolu toka u programu, atribute koji opisuju petlje i atribute koji opisuju funkcije.

Atributi koji opisuju tok podataka u programu služe za opis podataka koji učestvuju u naredbi grananja. Autori konstrušu drvo toka podataka (engl. *dataflow tree*) kojim modeluju konstante, operatore i promenljive, a čvorovi tog drveta kodiraju se kao kategorički atributi i dodaju u skup atributa naredbe grananja. Atributi koji opisuju kontrolu toka u programu koriste se da opišu oblik grafa kontrole toka, počevši od bloka koji sadrži naredbu grananja. Autori posmatraju samo jednostavne obrasce, poput trougla (blok koji ima dva sledbenika, A i B, pri čemu A ima B kao jedinog sledbenika) i dijamanta (blok koji ima dva sledbenika, A i B, koji oba imaju C kao jedinog sledbenika).

Petlje se opisuju atributima poput dubine ugnežđenja, brojem osnovnih blokova od koji se sastoji telo petlje, brojem izlaza iz petlje i slično. Atributi kojima se opisuju funkcije su broj instrukcija, broj osnovnih blokova i brojem grana u grafu kontrole toka funkcije. Ovi atributi izdvajaju se za funkciju u kojoj se nalazi naredba grananja koja se opisuje. Pored toga, autori izdvajaju još i ime fajla u kome se nalazi naredba grananja kao jedan od atributa.

Raman i Li koriste Guglovo široko profajliranje [245] za prikupljanje profila izvršavanja programa i označavanje naredbi grananja. Kako bi kreirali skup podataka za obučavanje modela, kompiliraju više od 750 programa koji se sastoje od ukupno preko 37 000 *C* i *C++* fajlova. Na taj način kreiraju skup podataka koji obuhvata više od sedam miliona jedinstvenih označenih naredbi grananja.

Za predviđanje profila izvršavanja programa Raman i Li obučavaju duboku neuronsku mrežu za regresiju. Njihov statički profajler postiže prosečno ubrzanje od 1% (geometrijska sredina) na 40 test programa, pri čemu pojedinačna poboljšanja dostižu i do 8.1%. Takođe, primenom razvijenog profajlera dolazi do poboljšanja performansi Guglove aplikacije za pretragu (aplikacije *Search*) za 1.2%.

VESPA

Statički profajler *VESPA* [202] predviđa verovatnoće izvršavanja grana naredbi grananja korišćenjem duboke neuronske mreže za regresiju. Kao i prethodno opisivani statički profajle-

ri, statički profajler *VESPA* se takođe ograničava na predviđanje profila izvršavanja binarnih naredbi grananja.

Atributi kojima se opisuju naredbe grananja izdvajaju se iz izvršivih fajlova odnosno binarne reprezentacije programa. Konkretno, radi se o binarnoj reprezentaciji za arhitekturu *x86*. *VESPA* koristi skup od 56 atributa od kojih je 21 atribut preuzet iz Kalderovog rada [37]. Preuzeti su atributi kao što su kôd operacije naredbe grananja (engl. *opcode instruction*), smer grane, odnosno informacija o tome da li je grana direktna ili povratna, zatim informacija da li blok koji sadrži naredbu grananja predstavlja početak petlje, kao i da li grana naredbe grananja sadrži poziv funkcije ili naredbu za izlazak iz programa.

Pored preuzetih atributa, definiše se dodatnih 35 atributa, kao što su broj instrukcija pisanja u memoriju (engl. *store instruction*), broj instrukcija čitanja iz memorije (engl. *load instruction*) i broj poziva metoda u bloku grafa kontrole toka koji sadrži naredbu grananja. Takođe se koriste atributi poput broja instrukcija u bloku koji sadrži naredbu grananja, broja indirektnih poziva i broja rekurzivnih poziva u funkciji u kojoj se nalazi naredba grananja. Za opisivanje funkcije u kojoj se nalazi naredba grananja koriste se i atributi poput maksimalnog nivoa ugnežđenja petlji u okviru funkcije, veličine funkcije izražene brojem osnovnih blokova, ukupnog broja petlji u funkciji i broja izlazaka iz petlji unutar funkcije.

Prilikom razvoja statičkog profajlera *VESPA* korišćena je rekurzivna eliminacija atributa, čime je odabrano 26 najinformativnijih atributa za obučavanje neuronske mreže. Od tih 26 atributa, 16 potiče iz Kalderovog rada, dok je preostalih 9 iz skupa novodefinisanih atributa.

Model duboke neuronske mreže korišćen u statičkom profajleru *VESPA* obučavan je na osnovu podataka prikupljenih iz 11 programa iz skupa test programa *SPEC CINT2006* [273] i 226 programa iz skupa LLVM test programa. Za prikupljanje profila izvršavanja korišćeno je instrumentaciono profajliranje. Tokom tog procesa prikupljeno je ukupno 2 093 873 naredbi grananja sa dve grane, od kojih je samo 513 316 bilo izvršeno i označeno. Ovaj skup od nešto više od pola miliona označenih binarnih naredbi grananja iskorišćen je za obučavanje modela.

Statički profajler *VESPA* integrisan je u binarni optimizator *BOLT* [228]. Za evaluaciju, autori su koristili skup od četiri test programa: kompilatore *Clang* [165] i *GCC* [97], kao i sisteme baza podataka *MySQL* [87] i *PostgreSQL* [276]. *BOLT*, kada koristi statički prediktor *VESPA*, na ovim test programima generiše izvršive fajlove koji su u proseku 5.47% brži nego izvršivi fajlovi koje generiše bez statičkog predviđanja verovatnoća izvršavanja grana.

Glava 4

Statički profajler *GraalSP*

Statički profajler *GraalSP* predviđa verovatnoće izvršavanja grana naredbi grananja u grafu kontrole toka programa. Ove verovatnoće se zatim koriste u okviru optimizacionih faza kompilatora za kreriranje efikasnih programa. Razvoj profajlera *GraalSP* obuhvata proces kreiranja skupa podataka za obučavanje modela mašinskog učenja i dizajn modela mašinskog učenja za statičko predviđanje verovatnoća izvršavanja grana u programu. Dodatno, obuhvata i razvoj statičkih heuristika za korigovanje predviđanja modela i osiguravanje performansi i u slučajevima odudarajućih podataka.

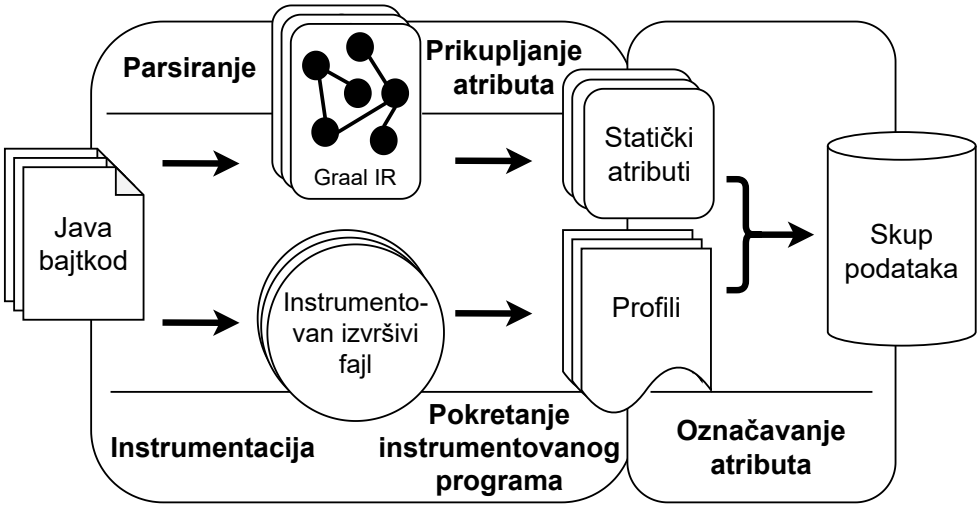
Statički profajler *GraalSP* predviđa verovatnoće izvršavanja grana naredbi grananja sa dve grane, predviđajući verovatnoću izvršavanja *prve grane* (p_1). Verovatnoću izvršavanja *druge grane* *GraalSP* računa kao $p_2 = 1 - p_1$. Termini *prva grana* i *druga grana* odnose se na redolsed u kome se grane naredbi grananja obrađuju prilikom parsiranja.

GraalSP koristi uniformnu raspodelu da modeluje verovatnoće izvršavanja grana kod naredbi grananja sa više od dve grane. Na ovaj način pojednostavljuje se dizajn statičkog profajlera i obučavanje modela mašinskog učenja. Dodatno, s obzirom na to da velika većina naredbi grananja ima samo dve grane, činjenica da statički profajler ne predviđa profil za naredbe grananja sa više od dve grane ne predstavlja nedostatak. Ovaj pristup prihvaćen je i primenjivan u razvoju ranijih relevantnih statičkih profajlera [13, 37, 202, 254].

4.1 Kreiranje skupa podataka

Slika 4.1 prikazuje proces kreiranja skupa podataka za obučavanje modela mašinskog učenja za statičko predviđanje verovatnoća izvršavanja grana u programu. Kompilator *GraalVM Native Image* parsira Java bajtkod kako bi kreirao internu međureprezentaciju *Graal IR* i odgovarajući graf kontrole toka programa. Proces kreiranja skupa podataka može se podeliti u dve faze.

U prvoj fazi kreiranja skupa podataka izdvajaju se atributi koji karakterišu naredbe grananja u reprezentaciji *Graal IR*. Za prikupljanje profila izvršavanja naredbi grananja koristi se dinamičko profajliranje zasnovano na instrumentaciji jer ono prikuplja profile najvišeg kvaliteta. Ovaj pristup takođe je korišćen u sličnim istraživanjima [254, 37]. Da bi se dobili reprezentativni i kvalitetni profili, programi su pokretani sa ulazima koji definisani u okviru kompilatora *Graal Native Image*. Ovi ulazi za prikupljanje profila su pažljivo konstruisani kako bi pokrili širok spektar funkcionalnosti i realističnih scenarija korišćenja aplikacija iz skupa programa za obučavanje modela. Na taj način prikupljeni profili precizno odražavaju način izvršavanja naredbi grananja u programima. U drugoj fazi procesa kreiranja skupa podataka koriste se prethodno



Slika 4.1: Kreiranje označenog skupa podataka za nadgledano obučavanje modela mašinskog učenja za statičko predviđanje verovatnoća izvršavanja grana u programima

prikupljeni profili kako bi se označili sakupljeni atributi. Na taj način kreira se označen skup podataka namenjen za nadgledano obučavanje modela mašinskog učenja.

Skup atributa

Prilikom kreiranja skupa podataka za obučavanje modela atributi kojima se opisuju naredbe grananja izdvajaju se iz *Graal IR* grafa i grafa kontrole toka programa (engl. *control flow graph*). Čvorovi u *Graal IR* grafu koji odgovaraju binarnim naredbama grananja (engl. *control flow split node*, skraćeno *cfs*-čvor) opisuju se atributima koji karakterišu:

(i) sam čvor naredbe grananja u *Graal IR* grafu,

(ii) blok B u grafu kontrole toka koji sadrži čvor naredbe grananja,

(iii) blokove u grafu kontrole toka koji prethode bloku B (engl. *predecessors blocks*),

(iv) blokove u grafu kontrole toka koji su dominirani[1] blokovima na koje grane iz bloka B (odnosno grane naredbe grananja) upućuju.

Naredbe grananja opisane su pomoću šesnaest atributa. Ovi atributi prikazani su u tabeli 4.1. Od šesnaest atributa, njih deset je preuzeto iz literature, dva atributa su prilagođena i modifikovana za potrebe razvoja alata *GraalSP*, dok se u ovoj disertaciji uvode četiri nova atributa.

Od šesnaest korišćenih atributa, njih jedanaest jesu numerički atributi a njih pet složeni atributi, koji se kodiraju u vektore celobrojnih vrednosti. Kada je reč o složenim atributima, tri atributa su kategorički atributi, dok su dva atributa rečnici u kojima su ključevi tipovi čvorova u reprezentaciji *Graal IR*, a vrednosti celi brojevi.

[1]U grafu kontrole toka programa blok A dominira blok B ako svaki put od početnog bloka grafa do bloka B mora da prolazi kroz blok A. Drugim rečima, izvršavanje bloka B nije moguće bez prethodnog izvršavanja bloka A.

Pored svakog od atributa, u koloni *Predmet* označeno je na šta se atribut odnosi. Atribut može da se odnosi na sâm čvor naredbe grananja (u tabeli obeleženo sa *cfs*-čvor), blok koji sadrži tu naredbu (u tabeli obeleženo sa *Blok*), blokove koji prethode tom bloku (u tabeli obeleženo sa *Prethodni blokovi*) ili blokove dominirane granama naredbe grananja (u tabeli obeleženo sa *Dominirani blokovi*).

Tabela 4.1: Statički atributi kojima se opisuju naredbe grananja. Kod kategoričkih atributa u zagradama je naveden broj različitih kategorija (tip atributa). Za atribute predstavljene rečnicima, u zagradi je dat maksimalan broj ključeva tih rečnika. Svi numerički atributi su celobrojne vrednosti.

#	Naziv	Vrsta	Tip	Predmet
1.	`CSType`	Preuzet	Kategorički (2)	*cfs*-čvor
2.	`CSBranchType`	Preuzet	Kategorički (14)	*cfs*-čvor
3.	`Header`	Preuzet	Kategorički (2)	Dominirani blokovi
4.	`BlocksCount`	Preuzet	Numerički	Dominirani blokovi
5.	`CSDepth`	Preuzet	Numerički	*cfs*-čvor
6.	`MaxCSDepth`	Preuzet	Numerički	Dominirani blokovi
7.	`LoopDepth`	Preuzet	Numerički	Blok, Dominirani blokovi
8.	`MaxLoopDepth`	Preuzet	Numerički	Dominirani blokovi
9.	`DominatorDepth`	Preuzet	Numerički	Blok
10.	`PCount`	Preuzet	Numerički	Blok
11.	`NodeCountMap`	Prilagođen	Rečnik (165)	Dominirani blokovi
12.	`FNodeCountMap`	Prilagođen	Rečnik (56)	Blok, Prethodni blokovi
13.	`AssemblySize`	Nov	Numerički	Blok, Dominirani blokovi
14.	`CPUCycles`	Nov	Numerički	Blok, Dominirani blokovi
15.	`CPUCheap`	Nov	Numerički	Blok, Dominirani blokovi
16.	`CPUExpensive`	Nov	Numerički	Blok, Dominirani blokovi

Preuzeti atributi. Prvih 10 atributa prikazanih u tabeli 4.1 preuzeti su iz relevantne literature [37, 202, 81, 254]. Atributi `CSType` i `CSBranchType` odnose se na sam čvor naredbe grananja u *Graal IR* grafu i tip čvora kojim se definiše grananje. Atribut `CSType` karakteriše tip čvora naredbe grananja i može biti *If* ili *Switch* jer su to čvorovi koji definišu grananje u reprezentaciji *Graal IR*. Atribut `CSBranchType` karakteriše logičku operaciju koja definiše grananje, npr. provera da li je broj negativan ili manji ili jednak od nule. U reprezentaciji *Graal IR* postoji 14 različitih takvih operacija.

Treći kategorički atribut `Header` odnosi se na prvi čvor u grani i izdvaja se za obe grane naredbe grananja odnosno za odgovarajuće dominirane blokove u grafu kontrole toka programa. Kako svaka grana u reprezentaciji *Graal IR* može započeti sa čvorom *LoopExit* ili čvorom *Begin*, ovaj kategorički atribut kodira se vektorom dužine dva. Atribut `BlocksCount` koristi se za opisivanje veličine grana, odnosno broja blokova u svakoj od njih.

Atributi numerisani rednim brojevima od pet do deset iz tabele 4.1 koriste se da bi se njima opisala kontrola toka u pogramu. Atribut `CSDepth` predstavlja dubinu ugnežđenja *cfs*-čvora u naredbama grananja, dok atribut `MaxCSDepth` predstavlja maksimalnu dubinu ugnežđenja u naredbama grananja i izdvaja se za blokove dominirane grananma naredbe grananja. Slično, atribut `LoopDepth` predstavlja dubinu ugnežđenja *cfs*-čvora u petljama, dok atribut `MaxLoopDepth`

predstavlja maksimalnu dubinu ugnežđenja u petljama i izdvaja se za grane naredbe grananja odnosno za odgovarajuće dominirane blokove.

Atribut `DominatorDepth` predstavlja nivo bloka u *stablu dominatora* grafa kontrole toka programa. *Stablo dominatora* predstavlja hijerarhijsku strukturu u kojoj svaki blok ima jedinstveni nadređeni dominatorski blok, a koren stabla je početni blok programa. Početni blok ima dubinu nula, a svaki naredni blok ima dubinu koja odgovara broju blokova koji ga dominiraju. Atribut `DominatorDepth` je atribut kojim se opisuje relacija dominacije među blokovima u grafu kontrole toka programa. Intuitivno, ovaj atribut meri koliko je blok udaljen od početka programa.

Atribut `PCount` odnosi se na blok u grafu kontrole toka programa koji sadrži *cfs*-čvor i označava broj njegovih prethodnika u tom grafu. Ovaj atribut koristi se za opis lokalne strukture grafa i karakteriše broj ulaznih blokova, odnosno različitih tokova programa koji se na tom mestu spajaju u jedan.

Prilagođeni atributi. Zekani i ostali [340] izdvajali su atribute iz blokova LLVM interne reprezentacije. Njihov skup atributa broji različite vrste LLVM instrukcija u okviru blokova grafa kontrole toka programa. Prilikom razvoja statičkog profajlera *GraalSP* izdvajani su atributi iz *Graal IR* grafova, što je grafovska međureprezentacija, pa je prethodna ideja prilagođena brojanjem različitih vrsta čvorova *Graal IR* grafa unutar blokova grafa kontrole toka (tabela 4.1, atributi `NodeCountMap` i `FNodeCountMap`). Atribut `NodeCountMap` je rečnik čiji ključevi predstavljaju sve fiksne čvorove u *Graal IR* grafu, dok vrednosti označavaju broj njihovih pojavljivanja u odgovarajućem delu grafa. Ovaj atribut koristi se za opisivanje kontrole toka programa i izdvaja se za obe grane naredbe grananja odnosno za odgovarajuće domomirane blokove. Slično, atribut `FNodeCountMap` je rečnik čiji ključevi predstavljaju sve pokretne čvorove u *Graal IR* grafu, dok vrednosti označavaju broj njihovih pojavljivanja u odgovarajućem delu grafa. Ovaj atribut koristi se za opisivanje toka podataka u programu i izdvaja se za obe grane naredbe grananja odnosno za odgovarajuće domomirane blokove i čvorove u bloku koji sadrži naredbu grananja.

Široko korišćeni atributi, kao što su broj *load* i *store* instrukcija, pokriveni su brojevima *Load* i *Store* čvorova u *Graal IR* grafu, a koje su sačuvane unutar atributa `NodeCountMap`. Slično tome, broj poziva funkcija i broj *Phi* instrukcija u bloku pokriveni su brojem odgovarajućih čvorova u atributima `NodeCountMap` i `FNodeCountMap`. Na ovaj način izbegnuto je u potpunosti ručno definisanje atributa za opisivanje naredbi grananja.

Pored toga što doprinose izražajnosti skupa atributa kojima se opisuju naredbe grananja, atributi `NodeCountMap` i `FNodeCountMap` povećavaju nivo zrelosti radnog okvira za obučavanje modela mašinskog učenja (engl. *ML training pipeline*). Kompilator *Enterprise GraalVM Native Image*, kao komercijalni proizvod, stalno se razvija, a unapređenja kompilatora se odražavaju kroz promene u internoj reprezentaciji *Graal IR*. Te promene podrazumevaju dodavanje novih čvorova, brisanje postojećih i promenu uloga postojećih čvorova (npr. promena u optimizaciji može uticati na način na koji optimizacija koristi čvorove interne reprezentacije). Korišćenjem atributa `NodeCountMap` i `FNodeCountMap`, model mašinskog učenja može redovno da se ponovo obučava (engl. *model re-training*), pri čemu se odabir najinformativnijih atributa automatizuje. Ovakav pristup obezbeđuje otpornost radnog okvira za obučavanje modela na promene u internoj reprezentaciji i eliminiše potrebu za manuelnim redefinisanjem atributa svaki put kada *Graal IR* doživi izmene.

Novi atributi. Pored preuzetih i prilagođenih atributa, u ovom radu uvedena su i četiri nova atributa kojima se opisuju naredbe grananja (tabela 4.1, atributi 13–16). Ovi atributi koriste se za opis bloka koji sadrži čvor naredbe grananja i blokova koji odgovaraju granama naredbi grananja.

Novi atributi opisuju kôd niskog nivoa koji će biti generisan na osnovu *Graal IR* grafa. Leopoldseder i ostali [174] kreirali su statički model koji za pojedinačne čvorove *Graal IR* grafa procenjuje broj CPU ciklusa potrebnih za izvršavanje instrukcija koje nastaju kompilacijom tih čvorova kao i veličinu odgovarajućeg asemblerskog koda. Na osnovu ovog modela definisani su atributi `CPUCycles` i `AssemblySize`.

Pored toga, skup atributa proširen je brojanjem čvorova koji su procenjeni kao CPU jeftini (atribut `CPUCheap`) i čvorova koji su procenjeni kao CPU skupi (atribut `CPUExpensive`). CPU jeftini čvorovi su oni čvorovi za čije izvršavanje je procenjeno da troši nula ili jedan CPU ciklus, a CPU skupi čvorovi oni čvorovi za čije izvršavanje je procenjeno da troši više od 64 CPU ciklusa. Izvršavanje čvora u ovom kontekstu odnosi se na izvršavanje odgovarajućih instrukcija niskog nivoa koje će biti generisane na osnovu tog čvora.

Kodiranje atributa

Kako *GraalSP* koristi samo tri kategorička atributa koji uzimaju mali mogući broj kategorija (atributi `CSType` i `Header` uzimaju po dve moguće kategorije dok atribut `CSBranchType` može uzeti četrnaest mogućih kategorija), ovi atributi kodirani su korišćenjem jediničnog kodiranja.

Slično, atributi predstavljeni kao rečnici, `NodeCountMap` i `FNodeCountMap`, kodirani su vektorima celih brojeva i to, atribut `NodeCountMap` vektorom dužine 165, a atribut `FNodeCountMap` vektorom dužine 56. Broj 165 jednak je broju različitih čvorova u *Graal IR* grafu dok je broj 56 jednak broju pokretnih čvorova u *Graal IR* grafu. Prilikom kodiranja atributa predstavljenih rečnicima svakoj vrednosti ključa rečnika odgovara tačno jedna pozicija u kodiranom vektoru atributa. Redosled čvorva određen je jedinstveno leksikografskim sortiranjem ključeva rečnika odnosno leksikografskim sortiranjem naziva čvorova u *Graal IR* grafu.

Nakon što se kategorički atributi kodiraju jediničnim kodiranjem i nakon što se atributi predstavljeni rečnicima kodiraju u odgovarajuće vektore, ovi vektori se konkateniraju sa numeričkim atributima. Na taj način se naredba grananja opisuje vektorom dimenzije 468.

Proces označavanja podataka

Naredbe grananja označavaju se frekvencijom izvršavanja njihove prve grane. Frekvencija izvršavanja prve grane računa se na osnovu broja izvršavanja grana, prikupljenog pomoću profajliranja zasnovanog na instrumentaciji. Kod naredbi grananja sa dve grane od kojih je prva izvršena n_1 a druga n_2 puta, frekvencija izvršavanja prve grane računa se po formuli:

$$\frac{n_1}{n_1 + n_2}.$$

Primer izdvajanja atributa

Implementacija sortiranja pomoću hipa iz standardne Java biblioteke je data na listingu 2. Da bi niz celobrojnih vrednosti bio sortiran u rastućem redosledu, metoda *heapSort* koristi pomoćnu metodu *pushDown*, koja smešta elemente u hip. Statički profajler predviđa verovatnoće izvršavanja tela `for` petlje na liniji 9, kao i tela *while* petlje na liniji 12. Te predviđene verovatnoće izvršavanja grana zatim se koriste tokom faza optimizacija vođenih profilom.

```
1  /**
2  * Sorts the specified range of the array using heap sort.
3  *
4  * @param a the array to be sorted
5  * @param low the index of the first element, inclusive, to be sorted
6  * @param high the index of the last element, exclusive, to be sorted
7  */
8  private void heapSort(int[] a, int low, int high){
9    for(int k = (low + high) >> 1; k > low; ){
10     pushDown(a, --k, a[k], low, high);
11   }
12   while(--high > low){
13     int max = a[low];
14     pushDown(a, low, a[high], low, high);
15     a[high] = max;
16   }
17 }
```

Listing 2: Implementacija sortiranja pomoću hipa u Java standardnoj biblioteci

Na slici 4.2 prikazani su graf interne međureprezentacije kompilatora *GraalVM Native Image* i graf kontrole toka, koji se odnose na petlju `for` sa listinga 2 (linije 9–11). Radi bolje preglednosti, određeni detalji *Graal IR* grafa su pojednostavljeni, kao što je, na primer, izostavljanje numeracije svih čvorova.

Cfs-čvor *1. If*, koji se nalazi u bloku *B1*, odgovara naredbi grananja sa dve izlazne grane. Prva grana povezuje čvor *1. If* sa čvorom koji predstavlja početak bloka *B2* (čvor *Begin*). U grafu kontrole toka, toj grani odgovara veza između blokova *B1* i *B2*. Druga grana naredbe grananja vodi ka čvoru koji predstavlja izlaz iz petlje (čvor *LoopExit*), kojim počinje blok *B9*. U grafu kontrole toka, toj grani odgovara veza između blokova *B1* i *B9*.

Da bi se opisala naredba grananja predstavljena *cfs*-čvorom *1. If*, izdvajaju se atributi iz:

(i) čvora naredbe grananja u *Graal IR* grafu (*1. If* čvora),

(ii) bloka *B1* u grafu kontrole toka koji sadrži čvor *1. If*,

(iii) bloka *B0* u grafu kontrole toka koji prethodi bloku *B1*,

(iv-a) blokova *B2*–*B8* u grafu kontrole toka koji su dominirani blokom *B2* (grana koja odgovara telu `for` petlje), i

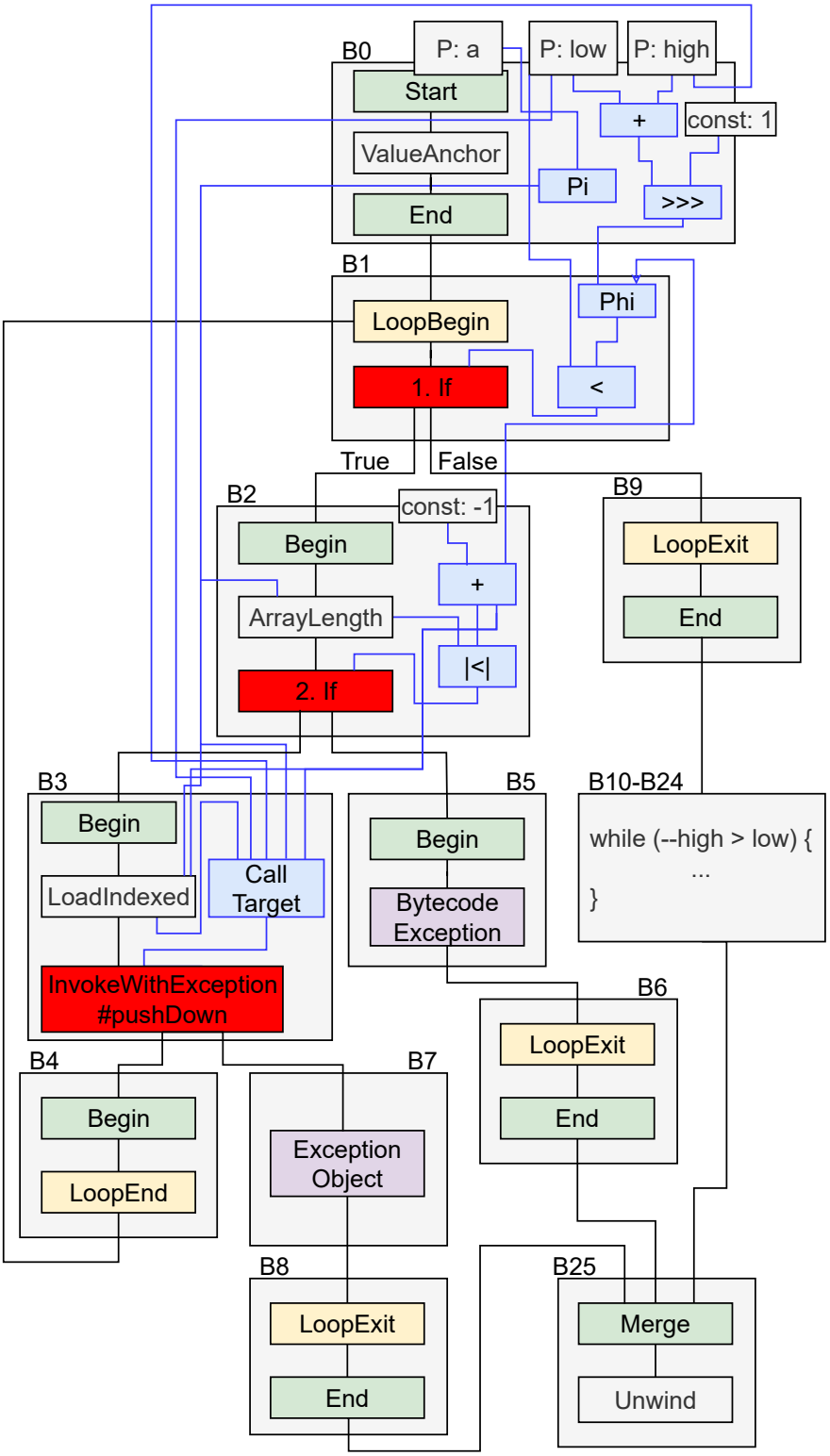
(iv-b) blokova *B9*–*B24* koji su dominirani blokom *B9* (grana koja odgovara izlasku iz petlje).

Na slici 4.3 prikazani su atributi izdvojeni za ovu naredbu grananja. Atributi su organizovani u tri vektora koji se odnose redom na

1) čvor naredbe grananja (i), blok koji sadrži taj čvor (ii) i blok koji mu prethodi (iii),

2) dominirane blokove *B2*–*B8* (iv-a) i

3) dominirane blokove *B9*–*B24* (iv-b).



Slika 4.2: *Graal IR* graf i graf kontrole toka koji odgovaraju `for` petlji funkcije *heapSort* (funkcija je prikazana na listingu 2)

Numeracija atributa na slici prati numeraciju iz tabele 4.1 i ponovljena je u nastavku teksta u zagradama (uz ime atributa).

Vektor atributa 1) obuhvata atribute `CSType`, `CSBranchType`, `CSDepth` (koji se odnose na čvor naredbe grananja), atribute `LoopDepth`, `DominatorDepth`, `PCount`, `FNodeCountMap`, `AssemblySize`, `CPUCycles`, `CPUCheap`, `CPUExpensive` (koji se odnose na blok koji sadrži čvor naredbe grananja) i dodatno još jednom atribut `FNodeCountMap` (koji se izdvaja i za blok koji prethodi bloku koji sadrži naredbu grananja).

CSType (1), CSBranchType (2), CSDepth (5). Vrednost atributa `CSType` za *cfs*-čvor *1. If* jeste kategorija `If` koja odgovara tipu naredbe grananja ovog čvora. Kako ova naredba grananja odgovara proveri ulaska u petlju i kako se ova provera svodi na proveru da li je brojač iteracija petlje manji od neke vrednosti, to vrednost atributa tipa grananja odnosno atributa `CSBranchType` jeste kategorija poređenja na manje (čvor `IntegerLessThan`, na slici predstavljen kao znak `<`). Vrednost atributa nivoa ugnežđenja u naredbama grananja (atribut `CSDepth`) jeste nula, budući da se naredba grananja koja odgovara *cfs*-čvoru *1. If* ne nalazi u okviru drugih naredbi grananja.

LoopDepth (7), DominatorDepth (9), PCount (10). Vrednost atributa nivoa ugnežđenja u petljama (atribut `LoopDepth`) je nula, obzirom da se naredba grananja koja odgovara *cfs*-čvoru *1. If* ne nalazi u okviru petlji. Kako blok *B0* dominira nad blokom *B1*, vrednost atributa `DominatorDepth` iznosi jedan. Takođe, broj blokova koji prethode bloku *B1* u kome se nalazi naredba grananja `PCount` jednak je jedan.

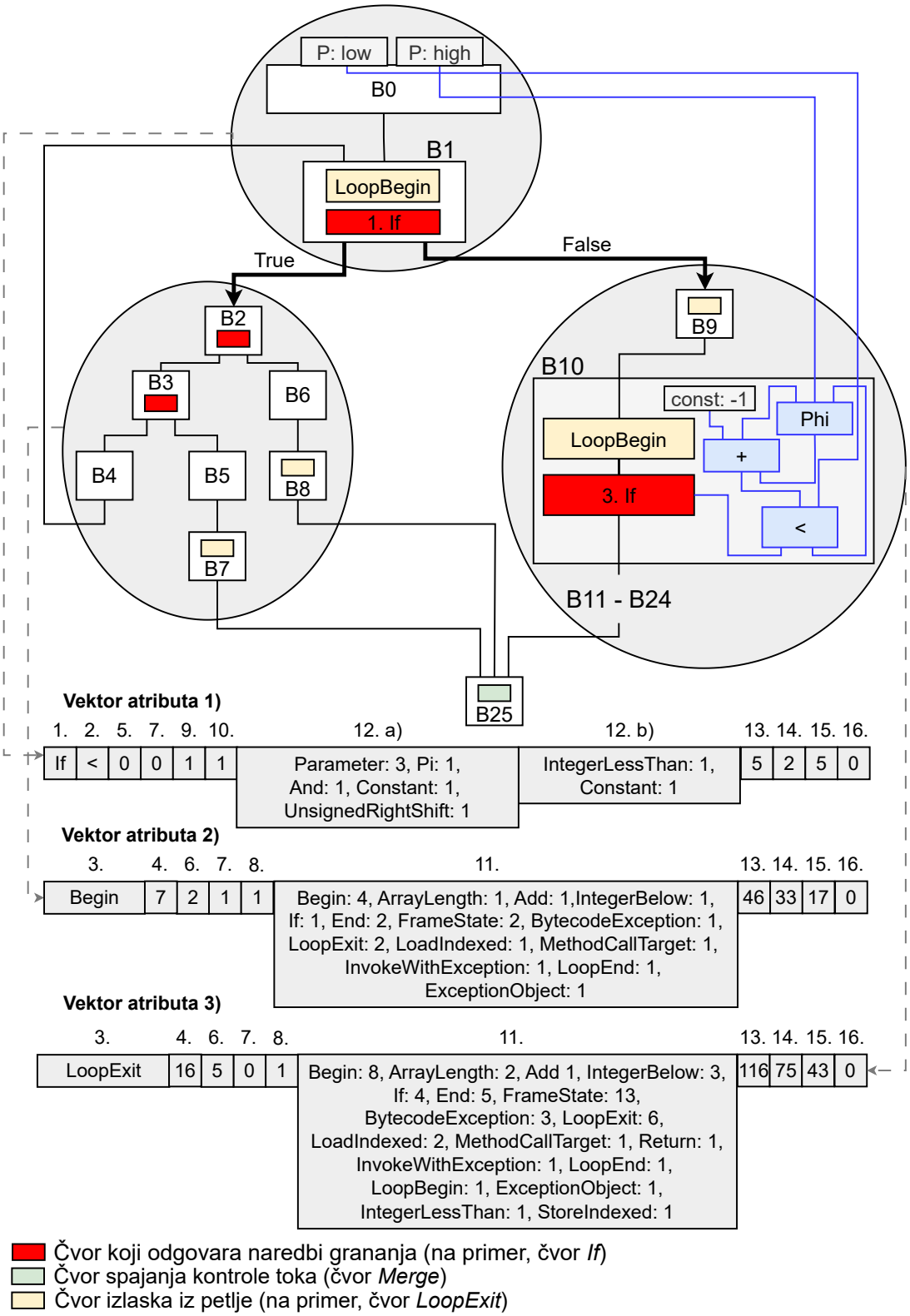
FNodeCountMap (12). Atribut `FNodeCountMap` izdvaja se za blok *B0* koji prethodi bloku *B1*, i predstavlja rečnik sa ključevima `Parameter`, `Pi`, `And`, `Constant` i `UnsignedRightShift`, obzirom da su ovo tipovi pokretnih čvorova koji se nalaze u bloku *B0*. Vrednosti u rečniku jednake su broju pojavljivanja tih čvorova u bloku *B0*. Na primer, u bloku *B0* nalaze se tri parametra i jedno neoznačeno šiftovanje u desno. Na slici 4.3 ovaj atribut označen je kao 12. a).

Atribut `FNodeCountMap` izdvaja se i za blok *B1* (na slci 4.3 označen kao 12. b)). Kako se u bloku *B1* nalaze samo dva pokretna čvora, to je izdvojena vrednost atributa rečnik sa dva ključa, čvorovima `IntegerLessThan` i `Constant`, i vrednostima jedan, za oba čvora.

AssemblySize (13), CPUCycles (14), CPUCheap (15), CPUExpensive (16). U vektoru atributa 1) vrednosti ovih atributa izdvajaju se za blok *B1* koji sadrži naredbu grananja. Vrednost atributa `AssemblySize` jeste 5 dok vrednost atributa `CPUCycles` iznosi 2. Pored toga, u bloku *B1* postoji pet CPU jeftinih čvorova i nijedan CPU skup čvor (atributi `CPUCheap` i `CPUExpensive`, redom).

Vektor atributa 2) obuhvata atribute `Header`, `BlocksCount`, `MaxCSDepth`, `LoopDepth`, `DominatorDepth`, `NodeCountMap`, `AssemblySize`, `CPUCycles`, `CPUCheap` i `CPUExpensive` (ovi atributi izdvajaju iz dominiranih blokova *B2–B8*).

Header (3), BlocksCount (4). Kako je prvi čvor u bloku *B2* čvor početka bloka (čvor `Begin`), to je on upravo i vrednost kategoričkog atributa `Header`. Broj blokova dominiranih prvom granom naredbe grananja jeste 7, što je i vrednost atributa `BlocksCount`.



Slika 4.3: Atributi kojima se opisuje naredba grananja koja odgovara `for` petlji funkcije *heapSort* (listing 2). Numeracija atributa odgovara numeraciji u tabeli 4.1.

MaxCSDepth (6), MaxLoopDepth (7), DominatorDepth (8). Maksimalni nivo ugnežđenja u naredbama gananja (atribut `MaxCSDepth`) u blokovima *B2–B8* iznosi dva jer se čvor koji predstavlja poziv metode *pushDown* (čvor *InvokeWithException* u bloku *B3* na slici 4.2) nalazi na dubini ugnežđenja dva. Takođe, blokovi *B2–B8* nalaze se na dubini ugnežđenja u petljama (atribut `MaxLoopDepth`) jednakoj jedan, budući da čine telo `for` petlje. Kako se blok *B2* nalazi na dubini jedan u stablu dominatora grafa kontrole toka metode *heapSort*, to je vrednost atributa `DominatorDepth` jednaka 1.

NodeCountMap (11). Blokovi *B2–B8* sadrže poziv metode (čvor *InvokeWithException*), bacanje bajtkod izuzetaka (čvorovi *BytecodeException* i *ExceptionObject*), učitavanje elemenata iz niza (čvor *LoadIndexed*), proveru dužine niza (čvor *ArrayLength*), dva izlaska iz petlje (čvorovi *LoopExit*) i slično. Spisak svih izdvojenih čvorova i brojevi njihovih pojavljivanja prikazani su na slici 4.3.

AssemblySize (13), CPUCycles (14), CPUCheap (15), CPUExpensive (16). Procenjena veličina asemblerskog koda (atribut `AssemblySize`) blokova *B2–B8* izračunata je sabiranjem procenjenih veličina asemblerskog koda čvorova *Graal IR* grafa koji se nalaze u ovim blokovima i ona iznosi 46. Slično, procenjeni broj CPU ciklusa (atribut `CPUCycles`) potrebnih za izvršavanje instrukcija koje odgovaraju čvorovima iz ovih blokova iznosi 33. Blokovi *B2–B8* sadrže 17 čvorova procenjenih kao CPU jeftini (atribut `CPUCheap`) i nijedan čvor procenjen kao CPU skup (atribut `CPUExpensive`).

Vektor atributa 3) obuhvata atribute `Header`, `BlocksCount`, `MaxCSDepth`, `LoopDepth`, `DominatorDepth`, `NodeCountMap`, `AssemblySize`, `CPUCycles`, `CPUCheap` i `CPUExpensive` (ovi atributi izdvajaju iz dominiranih blokova *B9–B24*).

Header (3), BlocksCount (4). Kako blok *B9* počinje čvorom izlaska iz petlje (čvor *LoopExit*) to je upravo on vrednost kategoričkog atributa `Header`. Broj blokova dominiranih drugom granom naredbe grananja jednak je 16, što je vrednost atributa `BlocksCount`.

MaxCSDepth (6), MaxLoopDepth (7), DominatorDepth (8). Maksimalna dubina ugnežđenja u naredbama grananja (atribut `MaxCSDepth`) u blokovima *B9–B24* iznosi 5, dok je, kako se blokovi *B9–B24* ne nalaze unutar petlje, vrednost atributa `LoopDepth` za drugu granu naredbe grananja jednaka 0. Blok *B9* nalazi se na dubini jedan u stablu dominatora grafa kontrole toka metode *heapSort*, pa je vrednost atributa `DominatorDepth` koja se izdvaja za drugu granu naredbe grananja jednaka 1.

NodeCountMap (11). Blokovi *B9–B24* sadrže petlju `while` sa listinga 2. Stoga se u rečniku `NodeCountMap` nalazi jedan čvor tipa `LoopBegin` koji odgovara početku te petlje, šest čvorova tipa `LoopExit` koji odgovaraju izlascima iz te petlje i jednom izlasku iz `for` petlje, kojim inače i počinje blok *B9*. Kako se u telu `while` petlje poziva metod `pushDown`, to se u ovom rečniku nalazi i jedan čvor tipa `InvokeWithException`. Pored toga, u blokovima *B9–B24* nalaze se i četiri naredbe grananja predstavljene čvorovima tipa `If`, čvorovi koji odgovaraju obradama izuzetaka (tri čvora tipa `BytecodeException` i jedan čvor tipa `ExceptionObject`) i slično.

AssemblySize (13), CPUCycles (14), CPUCheap (15), CPUExpensive (16). Procenjena veličina asemblerskog koda (atribut `AssemblySize`) blokova *B9—B24* iznosi 116, dok je procenjeni broj CPU ciklusa (atribut `CPUCycles`) potrebnih za izvršavanje instrukcija koje odgovaraju čvorovima iz ovih blokova 75. Blokovi *B9–B24* sadrže 43 *Graal*

IR čvora koji su procenjeni kao CPU jeftini (atriibut `CPUCheap`) i nijedan čvor koji je procenjen kao CPU skup (atribut `CPUExpensive`).

4.2 Modeli mašinskog učenja za predviđanje verovatnoća izvršavanja grana

Prilikom razvoja statičkog profajlera *GraalSP* najbolji rezultati postignuti su korišćenjem modela stabala odlučivanja, modela gradijentnog pojačavanja *XGBoost* i modela duboke neuronskie mreže. Pored ova tri modela, eksperimenti su vršeni i sa drugim modelima poput metoda potpornih vektora [125, 215] i algoritma k najbližih suseda [234, 342]. U ovoj sekciji dati su detalji razvoja i obučavanja najboljih modela mašinskog učenja za predviđanje verovatnoća izvršavanja grana u statičkom profajleru *GraalSP*.

Model stabla odlučivanja. Da bi se izbeglo preprilagođavanje iskorišćeno je odsecanje stabla sa parametrom odsecanja $cpp_\alpha = 10^{-6}$. Konkretna vrednost ovog parametra određena je eksperimentalno, na osnovu kvaliteta predviđanja modela na skupu za validaciju.

Model gradijentnog pojačavanja *XGBoost*. Model gradijentnog pojačavanja *XGBoost* predviđa verovatnoće izvršavanja grana naredbi grananja agregiranjem predviđanja 1500 stabala odlučivanja maksimalne dubine 10. Broj stabala određen je eksperimentalno, na osnovu ocene kvaliteta predviđanja modela na skupu za validaciju. Dublja stabla vode preprilagođavanju modela, dok ograničavanje maksimalne dubine stabala na manje od deset onemogućava model da modeluje podatke iz skupa za obučavanje na adekvatan način.

Model duboke neuronske mreže. Za predviđanje verovatnoća izvršavanja grana dizajnirana je duboka neuronska mreža koja se sastoji od šest skrivenih slojeva sa 512, 512, 512, 256, 256 i 256 neurona, redom. Neuroni u skrivenim slojevima koriste aktivacionu funkciju *ReLU*. Izlazni sloj sastoji se od jednog neurona koji koristi sigmoidnu aktivacionu funkciju. Predloženi model duboke neuronske mreže sastoji se od ukupno 832 513 parametara. Model je obučavan korišćenjem optimizatora *Adam*. Kako bi se sprečilo preprilagođavanje, primenjena je regularizacija izostavljanjem na poslednja dva skrivena sloja modela, nasumičnim isključivanjem 5% neurona iz tih slojeva. Model je obučavan korišćenjem srednjekvadratne funkcije greške. Prilikom obučavanja modela eksperimentisano je i binarnom unakrsnom entropijom (engl. *binary cross-entropy loss*) kao funkckijom greške, ali su nešto bolji rezultati postignuti korišćenjem srednjekvadratne greške.

Redukcija dimenzionalnosti

Korišćenje složenih atributa (kategorički atributi i atributi predstavljeni rečnicima) rezultuje visokodimenzionim vektorima atributa kojima su opisane naredbe grananja. Da bi se poboljšalo i ubrzalo obučavanje modela mašinskog učenja, potrebno je preprocesirati visokodimenzioni vektor atributa, smanjiti njegovu dimenzionalnost i eliminisati redundantne informacije [84, 154, 202].

Prilikom razvoja statičkog profajlera *GraalSP* korišćene su tehnike redukcije dimenzionalnosti zasnovane na varijansi, kao i analiza glavnih komponenti. Za model gradijentnog pojačavanja *XGBoost* i model stabla odlučivanja implementirana je redukcija dimenzionalnosti zasnovana

na varijansi. Na taj način se nakon smanjenja dimenzionalnosti zadržavaju originalni atributi i omogućava se interpretabilnost predviđanja. Ovakav izbor napravljen je i prilikom razvoja relevantnih statičkih profajlera zasnovanih na stablima odlučivanja [37, 254].

Prilikom obučavanja modela zasnovanih na stablima odlučivanja eliminisani su atributi čija je varijansa u skupu podataka manja od 0.5. Na taj način zadržano je 78 atributa. Vrednost praga za eliminaciju atributa određena je empirijski, korišćenjem unakrsne validacije za izbor hiperparametara ovih modela.

Za pripremu podataka za obučavanje modela duboke neuronske mreže iskorišćena je analiza glavnih komponenti. Korišćenjem analize glavnih komponenti zadržano je 78 najinformativnijih komponenti podataka, gde je broj komponenti odabran tako da bude jednak broju atributa na kojima su obučavani modeli zasnovani na stablima.

Težine instanci

Model mašinskog učenja koji se koristi za predviđanje verovatnoća izvršavanja grana mora precizno predviđati verovatnoće izvršavanja grana koje vode do frekventnih putanja u programu jer optimizacija ovakvih putanja značajno utiče na performanse optimizovanih programa. Da bi se model fokusirao na frekventne putanje, prilikom njegovog obučavanja korišćene su težine instanci.

Težina instanci određuje se na osnovu broja izvršavanja naredbi grananja. Broj izvršavanja neke naredbe grananja predstavlja zbir izvršavanja svih njenih grana. Na primer, ako naredba grananja ima dve grane, b_1 i b_2, pri čemu je b_1 izvršena n_1 puta, a b_2 je izvršena n_2 puta, ukupan broj izvršavanja te naredbe je $n_1 + n_2$. Težina jedne instance (odnosno naredbe grananja u programu) računa se kao odnos između broja izvršavanja te naredbe i ukupnog broja izvršavanja svih naredbi grananja u programu. Na taj način, težine instanci su realni brojevi u intervalu od nula do jedan, a njihov zbir u okviru jednog programa jednak je jedan.

4.3 Heuristike za korekciju predviđanja verovatnoća izvršavanja grana

Jedna od osnovnih pretpostavki mašinskog učenja jeste da će podaci na kojiima će korisnici obučenog modela koristiti model pripadati istoj raspodeli kao i podaci na kojima je obučavan model. Međutim, odudarajući podaci se uvek mogu pojaviti [105]. Ovo postaje posebno važno kada se model mašinskog učenja koristi u komercijalnom proizvodu. Da bi statički profajler *GraalSP* bio jos robustniji, unapređen je sa dve heuristike za korekciju predviđanja verovatnoća izvršavanja grana, koje se odnose na situacije kada se model mašinskog ulenja susretne sa odstupanjima u podacima. Tačne vrednosti parametara korišćene u heuristikama odabrane su eksperimentalno.

Heuristika petlje (engl. *loop heuristic*) osigurava da grana koja vodi ka telu petlje uvek ima predviđenu verovatnoću izvršavanja veću ili jednaku od 0.20. Ako model mašinskog učenja predvidi verovatnoću izvršavanja manju od 0.20, heuristika petlje će korigovati predviđanje i postaviti je na 0.20. Ova heuristika obezbeđuje da, čak i u slučaju pogrešnih predviđanja modela u vezi sa ponašanjem petlji, delovi koda koji se odnose na frekventno izvršavane petlje ne budu zanemareni tokom optimizacija. Na taj način sprečava se potencijalni gubitak performansi usled neadekvatnih predviđanja modela.

Heuristika zaustavljanja programa (engl. *dead-end heuristic*) osigurava da svaka grana koja vodi ka blokovima koji završavaju program ima predviđenu verovatnoću manju ili jednaku od

0.80. Ova heuristika primenjuje se samo u slučaju kada druga grana (ona koja ne vodi ka završetku programa) vodi ka blokovima čija je procenjena veličina asemblerskog koda odnosno vrednost atributa `AssemblySize` najmanje 50 bajtova. Cilj ove heuristike je da se izbegne favorizovanje grane koja vodi ka završetku programa kada postoji alternativna grana koja omogućava nastavak izvršavanja programa.

Važno je istaći da ove heuristike predstavljaju kompromis između veličine izvršivih fajlova programa i performansi tih programa. Na primer, osiguravanje da verovatnoća izvršavanja tela petlje bude najmanje 0.20 pomaže da se izbegnu greške u predviđanjima niskih verovatnoća izvršavanja petlji koje se zapravo često izvršavaju. Sa druge strane, to dovodi do agresivnije duplikacije koda unutar tela svake petlje, uključujući i one petlje koje pripadaju *hladnim* delovima koda, odnosno delovima koda koji se retko izvršavaju. Ovo rezultuje povećanjem veličine izvršivih fajlova optimizovanih programa. Dakle, heuristike za statičku korekciju predviđanja verovatnoća izvršavanja grana u programu donose dobitak u performansama, ali po cenu povećanja veličine optimizovanih programa.

4.4 Alat za analizu i otkrivanje grešaka u predviđanjima

Analiziranje predviđanja statičkog profaljera i otkrivanje grešaka u tim predviđanjima veoma je izazovno. Čak i u jednostavim programima poput programa koji na standardni izlaz ispisuje poruku *„Zdravo svete!"* (engl. *HelloWorld program*) izvrši se 1689 naredbi grananja[2]. Predviđena verovatnoća izvršavanja svake od grana ovih naredbi grananja može uticati na performanse programa, u zavisnosti od odluka koje se na osnovu ovih verovatnoća donesu tokom kompilacije i optimizacije programa.

Alat *GraalSP-PLog*

U okviru ove disertacije razvijen je alat *GraalSP-PLog* (skraćeno od *GraalSP Profiles Logger*), koji služi za analizu predviđanja i otkrivanje grešaka u predviđanjima statičkog profajlera *GraalSP*. Alat *GraalSP-PLog* je alat koji kompilira ulazni program, pokreće statički profajler *GraalSP*, beleži predviđanja modela mašinskog učenja i korekcije tih predviđanja dobijene statičkim heuristikama. Nakon toga, pokreće profajliranje zasnovano na instrumentaciji i beleži dinamički prikupljene frekvencije izvršavanja grana naredbi grananja. Na osnovu svih prikupljenih informacija alat generiše izveštaj o predviđanjima statičkog profajlera. Na taj način alat *GraalSP-PLog* predstavlja značajnu podršku u procesu evaluacije statičkog profajlera jer omogućava detaljan uvid u mesta u programima gde su predviđanja profajlera potencijalno netačna ili neusklađena sa očekivanjem.

Izveštaj alata *GraalSP-PLog* sadrži pregled svih predviđanja modela, zajedno sa relevantnim informacijama o kontekstu u kojem su predviđanja izvršena, uključujući naziv metode, pripadajuću klasu kao i poziciju unutar *Graal IR* grafa. Pored tekstualnog izveštaja, alat omogućava

[2]Prebrojavanje izvršenih naredbi grananja izvedeno je korišćenjem programa koji na standardni izlaz ispisuje poruku *„Zdravo svete!"* napisanog u progamskog jeziku Java i izvršenog korišćenjem kompilatora *Enterprise GraalVM Native Image*. Broj izvršenih naredbi grananja u ovom slučaju znatno je veći od broja izvršenih naredbi grananja u programima pisanim u jezicima niskog nivoa, na primer u programskom jeziku C. U Javi, izvršavanje programa uključuje čitav niz bibliotečkih i sistemskih funkcija: učitavanje izvršivog fajla, inicijalizacija sistema za upravljanje nitima i sakupljača otpadaka (engl. *garbage collector*), zatim formatiranje stringa, upis u izlazni bafer, eventualno zaključavanje I/O resursa i slično. Radi poređenja, broj izvršenih naredbi grananja u C programu koji na standardni izlaz ispisuje pozdravnu poruku iznosi oko 20.

i vizuelizaciju *Graal IR* grafova metoda, čime se dodatno olakšava identifikacija i razumevanje tačaka u kojima je došlo do greške u predviđanju.

Izveštaj alata *GraalSP-PLog* predstavlja fajl sa tabelarnim podacima sačuvan kao *.csv* dokument. Ovo omogućava jednostavno otvaranje, učitavanje, filtriranje i analizu podataka korišćenjem standardnih alata za obradu tabelarnih podataka poput Majkrosoftovog Eksela (engl. *Excel*) [319], ili javno dostupnog softvera poput alata *Libre Office* [294] ili *WPS Office* [148]. Na slici 4.4 prikazano je zaglavlje izvestaja kao i prvih deset instanci u izvestaju koje se odnose na pokretanje alata na primeru programa koji na standardni izlaz ispisuje poruku *„Zdravo svete!”*.

Declaring Class	Method Name	Node	NodeSource Position	Block	Block Frequency	Profiled Probability	Predicted Probability	Guarded Probability	Global Execution Frequency	Execution Count	Absolute Error	Weighted Absolute Error
java.nio.Buffer	position	29|If	37	B1	1	0	0.8747		0.060826	2807531	0.87465	0.053202
sun.nio.cs.StreamEncoder	implWrite	6|If	4	B0	1	1	0.0035		0.023199	1070808	0.99648	0.023118
java.nio.charset.CharsetEncoder	encode	99|If	97	B17	0.2832	1	0.4283		0.023204	1071006	0.57172	0.013266
sun.nio.cs.StreamEncoder	implWrite	52|If	25	B8	1.3333	0	0.1848		0.023348	1077649	0.18482	0.004315
sun.nio.cs.UTF_8$Encoder	encodeArrayLoop	73|If	103	B7	0.9999	0.0064	0.1623		0.023428	1081347	0.15592	0.003653
sun.nio.cs.StreamEncoder	implWrite	85|If	46	B13	0.6666	0.0064	0.1175	0.2	0.023348	1077671	0.11108	0.002593
jdk.internal.util.Preconditions	checkIndex	11|If	6	B2	1	1	0.9402		0.037947	1751488	0.0598	0.002269
java.nio.Buffer	limit	31|If	42	B1	1	0	0.1229		0.016424	758075	0.12294	0.002019
java.nio.charset.CharsetEncoder	encode	78|If	82	B13	1.1327	0.9936	0.9074		0.023353	1077886	0.08617	0.002012
sun.nio.cs.StreamEncoder	write	44|If	49	B14	1	0	0.2728		0.006794	313586	0.27276	0.001853

Slika 4.4: Izvestaj alata *GraalSP-PLog* za program koji na standardni izlaz ispisuje poruku „Zdravo svete!”

Za svaku naredbu grananja u ulaznom programu alat *GraalSP-PLog* prikazuje odgovarajući *cfs*-čvor u internoj reprezentaciji *Graal IR*, zajedno sa metodom u kojoj se taj *cfs*-čvor nalazi — prvo naziv klase kojoj metoda pripada (kolona *Declaring Class*), zatim naziv metode (kolona *Method Name*) i na kraju sam *cfs*-čvor (kolona *Node*). Pored toga, alat prikazuje i poziciju čvora u izvornom kodu programa (kolona *Node Source Position*), kao i naziv i frekvenciju bloka u grafu kontrole toka koji sadrži taj čvor (kolone *Block* i *Block Frequency*). Dodatno, alat prikazuje i dinamički prikupljenu frekvenciju izvršavanja prve grane naredbe grananja (kolona *Profiled Probability*), predviđenu verovatnoću izvršavanja te grane (kolona *Predicted Probability*), kao i korekciju tog predviđanja statičkim heuristikama (kolona *Guarded Probability*).

Pored prethodnog, alat za svaku naredbu grananja prikazuje i njenu normalizovanu frekvenciju izvršavanja (kolona *Global Execution Frequency*) i broj izvršavanja njenih grana (kolona *Execution Count*). Broj izvršavanja grana naredbe grananja računa se sabiranjem brojeva izvršavanja svih njenih grana i može značajno varirati u zavisnosti od ulaznih podataka i petlji u programu. Zbog toga alat prikazuje i normalizovanu frekvenciju izvršavanja naredbi grananja, koja se računa prema formuli:

$$\frac{b}{\sum_{b_i \in program} b_i},$$

gde je b broj izvršavanja grana konkretne naredbe grananja, a imenilac predstavlja ukupan zbir brojeva izvršavanja grana svih naredbi grananja u programu. Na taj način dobija se relativna vrednost jer zbir svih normalizovanih frekvencija izvršavanja na nivou celog programa iznosi jedan. Normalizovana frekvencija izvršavanja naredbi grananja omogućava lakšu procenu udela u izvršavanju programa pojedinačnih naredbi grananja.

Pored toga, izveštaj prikazuje i apsolutnu grešku predviđanja (kolona *Absolute Error*) koja predstavlja apsolutnu vrednost razlike predviđene verovatnoće izvršavanja prve grane naredbe grananja i njene dinamički prikupljene frekvencije izvršavanja. Radi bolje procene uticaja na performanse, pored apsolutne greške predviđanja alat prijavljuje i težinsku apsolutnu grešku predviđanja (kolona *Weighted Absolute Error*). Težinska apsolutna greška predviđanja računa

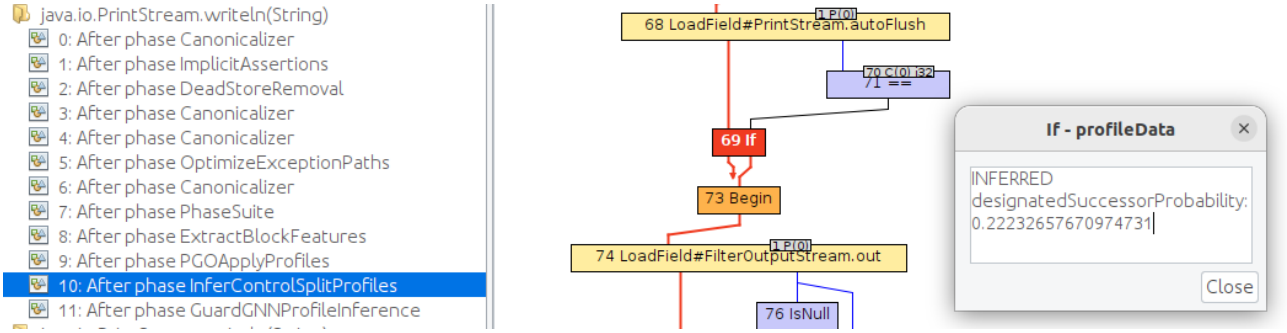
se kao proizvod apsolutne greške predviđanja i normalizovane frekvencije izvršavanja naredbe grananja.

Čvorovi u izveštaju sortirani su opadajućim redosledom prema težinskoj apsolutnoj grešci predviđanja. Budući da naredbe grananja sa većim težinama imaju veći uticaj na ukupne performanse programa, ovakav način beleženja omogućava lako prepoznavanje kritičnih tačnih i netačnih predviđanja sa aspekta performansi.

Vizuelizacija predviđanja uz pomoć alata *Ideal Graph Visualizer*

Alat *GraalSP-PLog* koristi alat *Ideal Graph Visualiser* za vizuelizaciju predviđanja modela. Korisnici alata *GraalSP-PLog* na taj način mogu videti *Graal IR* grafove u različitim fazama kompilacije odnosno u različitim fazama optimizacije kao i čvorove koji odgovaraju naredbama grananja i predviđene verovatnoće izvršavanja njihovih grana.

Primer izlaza alata IGV za vizuelizaciju metoda *writeline* iz Java standardne biblioteke koji se poziva prilikom ispisa poruke *Zdravo svete!* na standardni izlaz prikazan je na slici 4.5. Na levoj strani slike prikazane su različite faze kompilacije, dok je na centralnom delu slike prikazan deo *Graal IR* grafa ove metode. Na desnoj strani slike prikazana je predviđena verovatnoća izvršavanja prve grane čvora *69. If*. Faza kompilacije se bira dvostrukim klikom miša na odgovarajući naziv u padajućoj listi koja se nalazi sa leve strane. Na taj način korisnici alata mogu videti *Graal IR* grafa metode u odgovarajučoj fazi i predviđene verovatnoće izvršavanja grana ili prikupljene frekvencije izvršavanja (na primer u fazi *PGOApplyProfilesPhase*).



Slika 4.5: Grafički prikaz faza kompilacije (levo), dela *Graal IR* grafa metoda *writeline* iz Java standardne biblioteke nakon faze predviđanja verovatnoća izvršavanja grana modelom *XGBoost* (sredina) i predviđena verovatnoća izvršavanja prve grane čvora *69. If* (desno)

Glava 5

Implementacija

Statički profajler *GraalSP* razvijen je i prvi put integrisan u verziju 23.0 kompilatora *Enterprise GraalVM Native Image*, koja je objavljena u junu 2023. godine. Prilikom kompilacije programa statički profajler *GraalSP* podrazumevano je uključen.

Implementacija statičkog profajlera *GraalSP* sastoji se od nekoliko delova: kreiranja skupa podatka za obučavanje i validaciju modela, obučavanja modela mašinskog učenja za predviđanje verovatnoća izvršavanja grana u programu, implementacije heuristika za korekciju predviđanja modela mašinskog učenja i integracije statičkog profajlera u kompilator *Enterprise GraalVM Native Image*. Za kreiranje skupa podataka za obučavanje modela korišćeno je instrumentaciono profajliranje kompilatora *Enterprise GraalVM Native Image*, dok je programski jezik Pajton korišćen i za prikupljanje podataka i za obučavanje modela. Za integraciju statičkog profajlera korišćena je biblioteka *ONNX* biblioteka, i programski jezik Java.

5.1 Skupovi podataka za obučavanje i validaciju modela

Skup podataka za obučavanje modela kreiran je na osnovu podataka iz 44 Java aplikacije iz repozitorijuma *GraalVM Reachability Metadata* [222][1]. Ovaj repozitorijum je projekat otvorenog koda i javno je dostupan na servisu *GitHub*. Sastoji se od raznovrsnih aplikacija, uključujući različite biblioteke za kreiranje i upravljanje bazama podataka, biblioteke za kompresiju podataka, biblioteke za logovanje, mrežne biblioteke, veb servere, HTTP protokole, aplikacije za slanje i primenje e-pošte. Na taj način, skup podataka za obučavanje modela zasnovan je na savremenim aplikacijama i bibliotekama. Time se izbegava čest problem oslanjanja skupa podataka za obučavanje na test programe (engl. *benchmarks*) umesto na projekte iz stvarnog sveta (engl. *real-world programs*), što dovodi do razvoja statičkih profajlera lošijeg kvaliteta [202].

Repozitorijum *GraalVM Reachability Metadata* sadrži više verzija istih biblioteka. Na primer, postoje dve različite verzije biblioteke *com.mysql* (verzije *8.0.29* i *8.0.31*). Kako ove dve verzije biblioteke dele većinu koda, starija verzija biblioteke se ne koristi prilikom kreiranja skupa podataka za obučavanje modela. Korišćenje više verzija iste biblioteke prilikom kreiranja skupa podataka za obučavanje modela može se tumačiti kao augmentacija podataka [306, 337], koja bi model usmerila ka ponovljenim instancama. Zobg toga je, prilikom kreiranja skupa podataka za obučavanje modela, obezbeđeno da svaka biblioteka bude uključena samo u svojoj najnovijoj verziji. Korišćene biblioteke navedene su u tabeli 5.1.

[1]Skup podataka za testiranje sastoji se od standardnih test programa kompilatora *GraalVM Native Image* i potpuno je odvojen od skupa podataka za obučavanje i validaciju modela (Sekcija 6.1).

Tabela 5.1: Aplikacije i biblioteke iz skupa programa *GraalVM Reachability Metadata Repository* koje su korišćene za kreiranje skupa podataka za obučavanje modela

#	Biblioteka	Program	Verzija
1.	org.opengauss	opengauss-jdb	3.1.0
2.	io.netty	netty-common	4.1.80
3.	io.grpc	grpc-netty	1.51.0
4.	com.sun.mail	jakarta.mail	2.0.1
5.	io.opentelemetry	opentelemetry-exporter-jaeger	1.19.0
6.	com.mysql	mysql-connector-j	8.0.31
7.	com.hazelcast	hazelcast	5.2.1
8.	com.github.ben-manes.caffeine	caffeine	3.1.2
9.	org.jboss.logging	jboss-logging	3.5.0
10.	org.apache.tomcat.embed	tomcat-embed-core	10.0.20
11.	org.apache.httpcomponents	httpclient	4.5.14
12.	org.liquibase	liquibase-core	4.17.0
13.	org.thymeleaf	thymeleaf	3.1.0.RC1
14.	org.glassfish.jaxb	jaxb-runtime	3.0.2
15.	io.jsonwebtoken	jjwt-gson	0.11.5
16.	com.google.protobuf	protobuf-java-util	3.21.12
17.	com.graphql-java	graphql-java	19.2
18.	com.github.luben	zstd-jni	1.5.2
19.	org.apache.activemq	activemq-broker	5.18.1
20.	javax.cache	cache-api	1.1.1
21.	org.eclipse.jetty	jetty-client	11.0.12
22.	org.hibernate.orm	hibernate-core	6.2.0
23.	io.undertow	undertow-core	2.2.19
24.	org.ehcache	ehcache-jakarta	3.10.8
25.	com.ecwid.consul	consul-api	1.4.5
26.	org.apache.commons	commons-pool2	2.11.1
27.	org.quartz-scheduler	quartz	2.3.2
28.	commons-logging	commons-logging	1.2
29.	org.hdrhistogram	HdrHistogram	2.1.12
30.	com.zaxxer	HikariCP	5.0.1
31.	ch.qos.logback	logback-classic	1.4.1
32.	jakarta.servlet	jakarta.servlet-api	5.0.0
33.	org.freemarker	freemarker	2.3.31
34.	org.postgresql	postgresql	42.3.4
35.	org.mockito	mockito-core	4.8.1
36.	org.testcontainers	testcontainers	1.17.6
37.	org.flywaydb	flyway-core	9.0.1
38.	org.example	library	0.0.1
39.	com.microsoft.sqlserver	mssql-jdbc	12.2.0.jre11
40.	org.jline	jline	3.21.0
41.	org.jetbrains.kotlin	kotlin-stdlib	1.7.10
42.	com.h2database	h2	2.1.210
43.	org.jooq	jooq	3.18.2
44.	net.java.dev.jna	jna	5.8.0

Skup podataka za obučavanje modela sastoji se od 323 350 naredbi grananja. Svaka nared-

ba grananja označena je verovatnoćom izvršavanja svoje prve grane. Naredbe grananja koje odgovaraju obradama izuzetaka (engl. *bytecode exception*) isključene su iz skupa podataka za obučavanje modela s obzirom na to da su verovatnoće izvršavanja ovakvih grana očekivano veoma niske[2].

Tabela 5.2 prikazuje glavne karakteristike raspodele oznaka u skupu podataka za obučavanje modela. To je raspodela podataka koju model treba da nauči. Raspodela se odlikuje izraženom simetrijom, što potvrđuje koeficijent asimetrije od približno 0.1, kao i bliske vrednosti srednje vrednosti i medijane.

Na slici 5.1 prikazana je raspodela oznaka u skupu podataka za obučavanje modela. Ova raspodela potvrđuje očekivanje da se tokom izvršavanja programa, u većini slučajeva, izvršava samo jedna od grana naredbe grananja [13]. Međutim, kao što je prikazano na slici 5.2, kod 18.39% naredbi frekvencija izvršavanja prve grane nije blizu vrednostima 0 ili 1. Ovo opravdava odluku da se statički profajler *GraalSP* dizajnira tako da za predviđanje verovatnoća izvršavanja grana koristi model mašinskog učenja za regresiju, a ne klasifikaciju.

Tabela 5.2: Osnovne statistike raspodele oznaka iz skupa podataka za obučavanje modela

Naziv	Vrednost
Srednja vrednost	4.7315×10^{-1}
Standardna devijacija	4.5774×10^{-1}
Koeficijent asimetrije	1.0381×10^{-1}
Minimum	0.0
25. percentil	$1.000000000029 \times 10^{-6}$
50. percentil	$4.222222222222 \times 10^{-1}$
75. percentil	$9.999990000000 \times 10^{-1}$
Maksimum	1.0

Skup podataka za validaciju modela. Prilikom obučavanja modela stabla odlučivanja i modela gradijentnog pojačavanja korišćena je unakrsna validacija za podešavanje hiperparametara modela. Zbog računske složenosti, prilikom obučavanja modela duboke neuronske mreže nije korišćena unakrsna validacija. Umesto toga, hiperparametri modela podešavani su na nasumično izdvojenih 10% podataka iz skupa podataka za obučavanje modela.

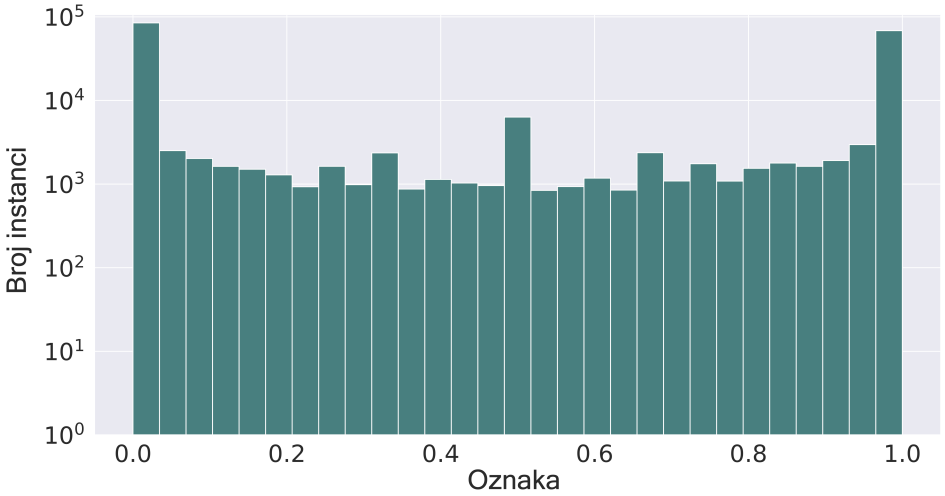
5.2 Implementacija modela mašinskog učenja

Prilikom obučavanja modela mašinskog učenja korišćene su težine instanci da bi se model fokusirao na najvažnije naredbe grananja. Automatizacija procesa obučavanja modela mašinskog učenja ostvarena je primenom kontinuirane integracije. Za obučavanje modela korišćene su biblioteke programskog jezika *Python*.

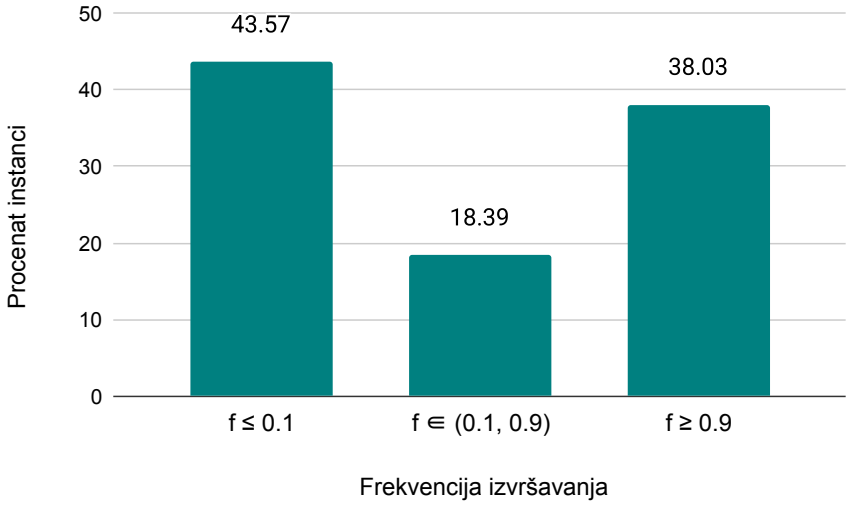
Težine instanci

Tabela 5.3 prikazuje glavne karakteristike raspodele težina instanci u skupu podataka za obučavanje modela, dok je ta raspodela vizuelno prikazana na slici 5.3. Raspodela je izrazito

[2]Verovatnoće izvršavanja grana naredbi grananja koje odgovaraju obradama izuzetaka su unapred definisane u kompilatoru *GraalVM Native Image*. Verovatnoća izvršavanja grane koja predstavlja izuzetak definisana je sa 0.000001, dok je verovatnoća izvršavanja grane kojom se nastavlja izvršavanje programa definisana sa 0.999999.



Slika 5.1: Raspodela oznaka iz skupa podataka za obučavanje modela (logaritamska skala, prikaz histogramom sa 50 stubića)



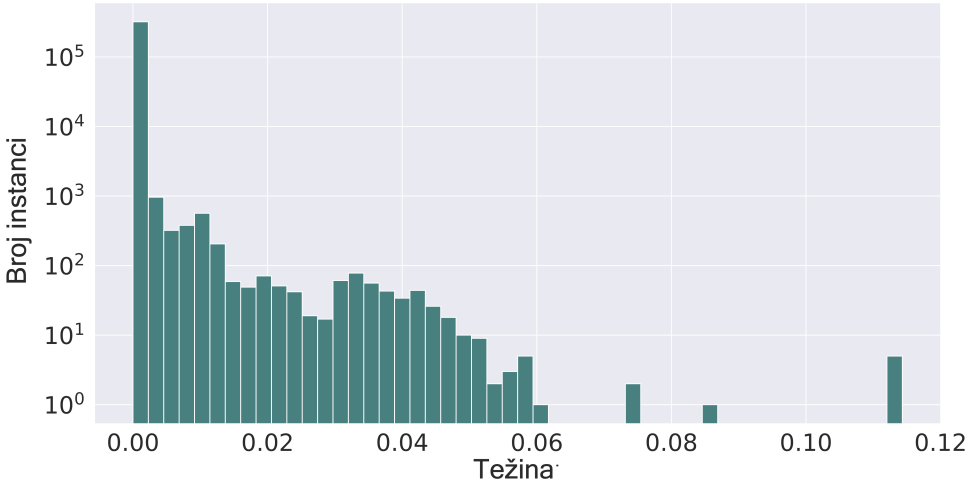
Slika 5.2: Klasifikacija oznaka u tri kategorije u zavisnosti od profajlirane frekvencije izvršavanja grane f

pozitivno asimetrična, sa dugim repom na desnoj strani. Leva grupa stubića histograma na slici 5.3 obuhvata težine manje od 0.002 i sadrži 320 212 instanci, što čini 99% skupa podataka za obučavanje modela.

Značajna razlika između 95. percentila i maksimalne vrednosti ukazuje na postojanje nekoliko izuzetno visokih vrednosti u raspodeli. Instance koje odgovaraju tim težinama nisu odudarajući podaci. Naprotiv, to su naredbe grananja za koje želimo da predviđanja modela budu što preciznija. Na primer, algoritam klasterovanja k-sredina (engl. *K-Means clustering algorithm*) [183, 189] je iterativni algoritam čije se performanse značajno oslanjaju na performanse izvršavanja njegovih glavnih petlji. Grane koje odgovaraju tim petljama često se izvršavaju, zbog čega imaju visoku frekvenciju izvršavanja. Zbog toga su ovakvim instancama u skupu podataka

Tabela 5.3: Osnovne statistike raspodele težina instanci iz skupa podataka za obučavanje modela

Naziv	Vrednost
Srednja vrednost	1.3515×10^{-4}
Standardna devijacija	1.6963×10^{-3}
Koeficijent asimetrije	$2.2889 \times 10^{+1}$
Minimum	1.6038×10^{-10}
25. percentil	1.4748×10^{-8}
50. percentil	6.0001×10^{-8}
75. percentil	4.0563×10^{-7}
90. percentil	4.4665×10^{-6}
95. percentil	2.5704×10^{-5}
Maksimum	1.1431×10^{-1}



Slika 5.3: Histogram normalizovanih težina instanci iz skupa podataka za obučavanje modela (logaritamska skala, prikaz na 50 stubića)

za obučavanje dodeljene visoke težine.

Kontinuirana integracija

Prilikom razvoja statičkog profajlera *GraalSP* korišćena je infrastruktura projekta *GraalVM* za kontinuiranu integraciju i automatizaciju kreiranja skupa podataka i obučavanja modela mašinskog učenja. Ovaj proces se redovno izvršava (jednom mesečno) kako bi model mašinskog učenja uvek bio ažuran i obučavan na najnovijim verzijama interne reprezentacije kompilatora *Enterprise GraalVM Native Image*.

Korišćenje složenih atributa, uz redovno ponovno kreiranje skupa podataka i obučavanje modela mašinskog učenja, čini tok kontinuirane integracije i isporučivanja (engl. *continuous integration and deployment pipeline*) otpornim na promene u internoj reprezentaciji kompilatora. Java zajednica konstantno ažurira repozitorijuim *GraalVM Reachability Metadata* novim aplikacijama i najnovijim verzijama biblioteka. Zahvaljujući tome, korišćenje programa iz ovog repozitorijuma za kreiranje skupa podataka u kombinaciji sa automatizacijom procesa kreiranja

i obučavanja modela mašinskog učenja, omogućava da se komercijalna upotrebljivost statičkog profajlera *GraalSP* vremenom poboljšava. Za obeležavanje instanci u skupu podataka korišćen je profajler zasnovan na instrumentaciji koji je integrisan u kompilator *Enterprise GraalVM Native Image*.

Softverska i hardverska podrška

Prilikom razvoja statičkog profajlera *GraalSP* za trening modela mašinskog učenja korišćen je programski jezik *Python* u verziji 3.8 i:

1. biblioteka *SciKit Library*, verzija 1.1.2 [231] (korišćena za implementaciju preprocesiranja podataka i razvoj modela stabla odlučivanja),
2. biblioteka *DLMC XGBoost Library*, verzija 1.6.1 [50] (korišćena za razvoj modela gradijentnog pojačavanja),
3. ML radni okvir *PyTorch*, verzija 1.12.1 [230] (korišćen za razvoj modela duboke neuronske mreže).

Implementirani modeli obučavani su na laptop računaru sa šestojezgarnim procesorom Intel(R) Core(TM) i7-9850H sa radnom frekvencijom od 2.60 GHz i 32 GB DDR4 radne memorije. Obučavanje modela traje manje od sat vremena. To omogućava integraciju obučavanja modela u sisteme kontinuirane integracije projekta *GraalVM* i redovno ažuriranje i ponovno obučavanje modela.

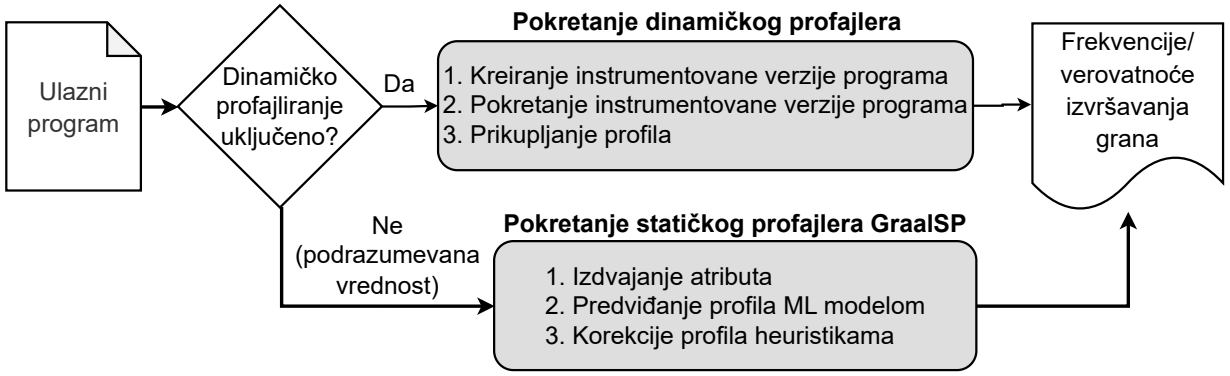
5.3 Implementacija heuristka za korekciju predviđanja modela mašinskog učenja

Heuristike za statičku korekciju predviđanja verovatnoća izvršavanja grana u programu implementirane su kao posebna faza kompilacije kompilatora *GraalVM Native Image*. Ova faza pozicionirana je odmah nakon faze predviđanja profila obučenim modelom mašinskog učenja. Nakon što model predvidi verovatnoće izvršavanja grana, faza korekcije predviđanja modela statičkiim heuristikama u linearnom vremenu prolazi kroz sve binarne naredbe grananja u okviru interne reprezentacije metoda i na odgovarajućim mestima poziva heuristiku petlje i heuristiku zaustavljanja programa. Kako se ove dve heuristike primenjuju na različite naredbe grananja u programu, redosled primena ovih heuristika nije važan jer je na svaku naredbu grananja primenljiva najviše jedna od njih.

Heuristika zaustavljanja programa primenjuje se u slučaju da jedna grana vodi ka dominiranim blokovima koji zaustavljaju program pri čemu alternativna grana odnosno ona grana koja ne vodi zaustavljanju programa vodi ka blokovima čija je procenjena veličina asemblerskog koda najmanje 50 bajtova. Kako se veličina asemblerskog koda dominiranih blokova već računa prilikom izdvajanja atributa koji karakterišu naredbe grananja, heuristika zaustavljanja programa koristi atribut `AssemblySize` modela mašinskog učenja. Heuristika petlje ne koristi posebne atribute, već u konstantnom broju operacija proverava da li uslov grananja ispunjava kriterijume za njenu primenu. Na taj način heuristike za korekciju predviđanja modela mašinskog učenja izvršavaju se efikasno, bez potrebe za novim, računski zahtevnim operacijama.

5.4 Integracija u kompilator *Enterprise GraalVM Native Image*

Na slici 5.4 prikazana je integracija statčkog profajlera *GraalSP* sa dinamičkim profajlerom zasnovanim na instrumentaciji. Ako je dinamičko profajliranje uključeno, kreira se instrumentovana verzija programa koja se pokreće kako bi se prikupio profil izvršavanja programa. U tom slučaju optimizacije vođene profilom izvršavaće se korišćenjem prikupljenih frekvencija izvršavanja grana. U suprotnom, kompilator će upotrebiti statički profajler *GraalSP* da predvidi verovatnoće izvršavanja grana i optimizacije vođene profilom izvršavaće se na osnovu predviđenih verovatnoća.



Slika 5.4: Integracija statičkog profajlera *GraalSP* sa dinamičkim profajlerom u kompilatoru *GraalVM Native Image*

U kompilatoru *Enterprise GraalVM Native Image* dinamičko profajliranje podrazumevano je isključeno. Kada je profil izvršavanja programa sakupljen dinamičkim profajliranjem[3], statičko profajliranje se isključuje. Ova odluka je zasnovana na činjenici da dinamički profajler kompilatora *Enterprise GraalVM Native Image* prikuplja kvalitetan profil za one delove programa koji će zaista biti izvršeni. Zbog toga, predviđanje verovatnoća izvršavanja grana u delovima programa koji se neće izvršiti prilikom izvršavanja programa je suvišno. Pored toga, takvo predviđanje samo dodatno povećava vreme kompilacije programa.

Za integrisanje obučenog modela mašinskog učenja u kompilator *Enterprise GraalVM Native Image*, korišćena je biblioteka *ONNX Java Runtime* [83], verzija 1.20.0. Ova biblioteka omogućava integrisanje obučenih modela mašinskog učenja u kompilator *Enterprise GraalVM Native Image* na operativnim sistemima *Windows*, *Linux* i *Darwin*.

Izdvajanje atributa, predviđanje verovatnoća izvršavanja grana naredbi grananja modelom mašinskog učenja i korekcije predviđanja modela statičkim heuristikama implementirani su kao faze u procesu kompilacije. Izdvajanje atributa i predviđanje verovatnoća izvršavanja grana naredbi grananja vrše se nakon faze parsiranja koda i kreiranja *Graal IR* grafova metoda. To je urađeno da bi se omogućilo što većem broju faza da koriste predviđene verovatnoće odnostno da bi se omogućilo izvršavanje što većeg broja optimizacija vođenih profilom. Takođe, moguće je klonirati faze izdvajanja atributa i predviđanja verovatnoća izvršavanja grana naredbi grananja i uključiti ih na različita mesta u lancu faza kompilacije programa. Kako se interna reprezentaci-

[3]U kompilatoru *Enterprise GraalVM Native Image* dinamičko profajliranje može se eksplicitno uključiti navođenjem opcije `--pgo-instrument`. Optimizacija programa korišćenjem dinamički prikupljenog profila izvršavanja programa može se uključiti pomoću opcije `--pgo`.

ja programa menja tokom faza kompilacije, ovo dovodi do preciznijih profila u kasnijim fazama kompilacije. Eksperimentalna evaluacija pokazala je samo blagi porast performansi, dok su se troškovi vremena kompilacije značajno povećavali. Zato tokom kompilacije nisu klonirane faze za izdvajanje atributa i predviđanje verovatnoće izvršavanja grana naredbi grananja.

Statički profajler *GraalSP* istovremeno predviđa verovatnoće izvršavanja svih grana u okviru jedne metode, čime se smanjuje broj poziva predviđanja modela i ukupno vreme kompilacije. Ovakav pristup, koji predviđanja verovatnoća vrši na nivou metode, odgovara načinu rada kompilatora *Oracle GraalVM Native Image*, gde su metode osnovna jedinica paralelizacije procesa kompilacije.

5.5 Implementacija alata *GraalSP-PLog*

Alat *GraalSP-PLog* implementiran je kao deo *mx* okruženja. Mehanizam za čuvanje predviđanja statičkog profajlera *GraalSP* realizovan je kroz izmenu faze predviđanja profila i faze korekcije predviđanja modela pomoću statičkih heuristika. Kada je alat *GraalSP-PLogger* aktiviran, ove faze automatski beleže sva svoja predviđanja, što se kasnije koristi za generisanje izveštaja. Pored toga, alat *GraalSP-PLog* pokreće izgradnju instrumentovane verzije programa, koju zatim izvršava kako bi prikupio dinamički profil. Nakon izvršavanja, *GraalSP-PLog* upisuje dinamički profil u izveštaj i omogućava njegovo poređenje sa statički predviđenim profilom.

Primer komande za pokretanje alata dat je na listingu 3. Prilikom pokretanja alata, potrebno je navesti putanju do instalacije Jave koja će se koristiti za kompilaciju (u ovom slučaju Java 25), kao i putanju do Jave 21, koju zahteva alat IGV. Osim toga, navodi se i relativna putanja do izvorne datoteke (u ovom slučaju *HelloWorld.java*), kao i regularni izraz za filtriranje metoda čiji *Graal IR* grafovi će biti prikazani u alatu IGV (u ovom slučaju *Graal IR* graf metode *main*). Prilikom pokretanja alata IGV, prikazuju se IR grafovi svih metoda čija imena odgovaraju navedenom regularnom izrazu.

```
mx --java-home=$JAVA25_HOME --tools-java-home=$JAVA21_HOME ml debug \
--input-file HelloWorld.java --method-filter=main
```

Listing 3: Komanda za pokretanje alata *GraalSP-PLog* na primeru programa koji na standardni izlaz ispisuje poruku *Zdravo svete!*

Glava 6

Evaluacija

Prilikom evaluacije statičkog profajlera *GraalSP* ocenjivani su kvalitet skupa atributa kojima se opisuju naredbe grananja kao i kvalitet obučenih modela mašinskog učenja u terminima odgovarajućih metrika, na primer, u slučaju modela za regresiju, srednjekvadratne greške, apsolutne greške i koeficijenta determinacije. Pored toga, evaluacija statičkog profajlera *GraalSP* uključila je i evaluaciju njegovog uticaja na performanse programa optimizovanih prema predviđenom profilu.

Najvažniji pokazatelj kvaliteta statičkog profajlera je njegov doprinos performansama izvršavanja programa. Pored toga, u evaluaciju statičkog profajlera *GraalSP* uključeni su i drugi relevantni aspekti, poput veličine generisanih izvršivih fajlova i vremena potrebnog za kompilaciju programa. Naime, poboljšanje performansi izvršavanja nije teško postići ako se ovi faktori zanemare. Na primer, jednostavna heuristika koja umeće pozive svih metoda može ubrzati program, ali često uz cenu značajnog povećanja veličine izvršivog fajla i produženja vremena kompilacije.

6.1 Programi za testiranje

Za evaluaciju statičkog profajlera *GraalSP* korišćen je skup standardnih programa koji se koriste za merenje performansi i ocenu kvaliteta kompilatora *GraalVM Native Image*:

- Skup programa *Renaissance* [241] predstavlja savremenu i raznovrsnu kolekciju aplikacija razvijenih u okviru različitih programskih paradigmi. U njemu se nalaze programi zasnovani na konkurentnom, funkcionalnom i objektno-orijentisanom programiranju, kao i aplikacije za obradu velikih količina podataka, mrežnu razmenu poruka, obradu tokova podataka (engl. *stream processing*) i mašinsko učenje. Kolekcija obuhvata raznovrsne primere, poput optimizacije genetskih algoritama korišćenjem biblioteke *Jenetics* [318], kao i simulacije intenzivnog serverskog opterećenja pomoću radnog okvira *Twitter Finagle* [300].

- Skup programa *DaCapo* [23] sastoji se od modernih i raznovrsnih programa koji se koriste u komercijalnoj i slobodnoj upotrebi. Na primer, obuhvata programe za simulaciju procesa na mreži AVR mikrokontrolera [160], kreiranje PDF dokumenata, indeksiranje i pretraživanje velikih kolekcija dokumenata, kao i alate za analizu izvornog koda.

- Skup programa *DaCapo con Scala* [267] predstavlja proširenu verziju *DaCapo* kolekcije, usmerenu na programe napisane u programskom jeziku Skala. Uključuje biblioteke za

obradu programskih jezika, kompilatore i dekodere klasa za programski jezik Skala, alat za formatiranje koda, kao i aplikacije za obradu i povezivanje sa XML podacima, takođe razvijene u programskom jeziku Skala.

Svi programi korišćeni za evaluaciju statičkog profajlera *GraalSP* prikazani su u tabeli 6.1. Ovi programi su različiti od onih koji su korišćeni za obučavanje modela i ova dva skupa programa se međusobno ne preklapaju. Skup podataka za testiranje modela mašinskog učenja formiran je na osnovu podataka dobijenih iz ovih programa i sadrži ukupno 198 157 naredbi grananja. Na ovom skupu podataka ocenjivan je kvalitet obučenih modela za predviđanje verovatnoća izvršavanja grana. Pored toga, na ovim test programima ocenjivan je i uticaj statičkog profajlera *GraalSP* na performanse optimizovanih programa.

6.2 Izvođenje eksperimenata

Da bi se obezbedila validnost evaluacije, eksperimenti su izvođeni u izolovanom okruženju, bez izvršavanja drugih aplikacija na serveru tokom izvršavanja test programa. Na taj način su eliminisani svi potencijalni spoljašnji faktori koji bi mogli uticati na rezultate.

Softverska konfiguracija sistema za evaluaciju. Svi eksperimenti sprovedeni tokom evaluacije statičkog profajlera *GraalSP* izvođeni su u istom okruženju. Korišćen je kompilator *Enterprise GraalVM Native Image*, verzija *23.0*, zasnovan na Javi 17. Eksperimenti su pokretani na sistemu sa operativnim sistemom *Oracle Linux Server* verzije *7.4* i kernelom *Linux 4.1.12-112.14.13.el7uek.x86_ 64*. Za upravljanje memorijom korišćen je sakupljač otpadaka *Serial Garbage Collector* iz kompilatora *GraalVM Native Image*.

Hardverska konfiguracija sistema za evaluaciju. Eksperimenti su izvođeni na računarskom klasteru čiji čvorovi opremljeni sa po dva *Intel E5-2699* procesora sa frekvencijom od *2.30* GHz, svaki sa *18* jezgara. Svaki čvor poseduje L1 keš memoriju od 64 KB, L2 keš memoriju od 256 KB i L3 keš memoriju od 46080 KB. Pored toga, sistem raspolaže sa 512 GB DDR4 radne memorije, *LSI MegaRAID SAS-3 3108* kontrolerom i *Mellanox Connect-X Infiniband* mrežnim karticama. Eksperimenti se mogu izvršavati i na laptop računarima sa znatno slabijim performansama od korišćenog servera.

Podešavanje okruženja. Prilikom evaluacije isključeno je ubrzanje procesora (engl. *turbo boost*), proces je izvršavan na prvom jezgru procesora, pri čemu je korišćena memorija samo prvog jezgra procesora kako bi se smanjila nestabilnost merenja izazvana promenljivom frekvencijom procesora i neuniformnim pristupom memoriji [149, 20]. Eksperimenti su pokretani sa uključenom tehnologijom hiperniti (engl. *hyperthreading*), a umesto standardnog diska korišćen je RAM disk veličine 40 GB, kako bi se izbegla nestabilnost rezultata uzrokovana ulazno-izlaznim operacijama. Procesori su postavljeni u nulto stanje (engl. *C-state 0*) kako bi se onemogućili režimi uštede energije koji mogu uticati na nestabilnost merenja rezultata.

Odabir ulaza za izvršavanje test programa. Prilikom pokretanja programa za testiranje korišćen je standardni skup ulaza koji simuliraju tipične scenarije upotrebe aplikacija iz skupova programa *Renaissance*, *DaCapo* i *DaCapo con Scala*. Ovi ulazi predstavljaju deo platforme *GraalVM Native Image*, koja ih već dugi niz godina koristi za pokretanje programa za testiranje radi što vernije i realističnije ocene kvaliteta i performansi kompilatora.

Tabela 6.1: Programi korišćeni za evaluaciju profajlera *GraalSP*. Kolona n prikazuje broj izvršavanja testa, dok kolona Vreme izvrš. (ms) prikazuje prosečno vreme i standardnu devijaciju.

Skup programa *DaCapo*, verzija 9.12

#	Program	Opis	n	Vreme izvrš. (ms)
1.	avrora	Simulacija izvršavanja programa na mreži mikrokontrolera [160].	20	6741.39 ± 18.24
2.	fop	Parsiranje i konvertovanje *XSL-FO* fajlova u PDF.	40	1206.28 ± 14.90
3.	h2	Simulacija bankarskog sistema korišćenjem relacionih baza podataka [129].	40	27112.14 ± 130.72
4.	luindex	Indeksiranje dela Šekspira i Biblije kralja Džejmsa pomoću sistema *Apache Lucene* [21, 186].	20	5679.21 ± 68.53
5.	lusearch	Pretraga ključnih reči u korpusu koji obuhvata dela Šekspira i Bibliju kralja Džejmsa pomoću sistema *Apache Lucene*.	40	438.01 ± 6.11
6.	pmd	Detektor programerskih grešaka [236] koji detektuje česte greške u programiranju poput neiskorišćenih promenljivih, praznih *catch* blokova i slično.	30	15947.94 ± 180.13
7.	sunflow	Obrada računski zahtevnih zadataka poput renderovanja korišćenjem sistema *Sunflow* [158, 108].	30	1938.48 ± 29.38

Skup programa *Renaissance*, verzija 1.14.0

#	Program	Opis	n	Vreme izvrš. (ms)
8.	akka-uct	Izvršava *Unbalanced Cobwebbed Tree* test program [344] korišćenjem radnog okvira *Akka* [224, 280].	24	51116.57 ± 337.46
9.	finagle-chirper	Simulira mikrobloging servis koristeći radni okvir *Twitter Finagle* [300] za izgradnju mrežnih aplikacija.	90	8483.10 ± 83.10
10.	finagle-http	Simulacija velikog opterećenja servera koristeći radni okvir *Twitter Finagle*.	12	9223.50 ± 51.95
11.	fj-kmeans	Izvršava algoritam k-sredina (engl. *k-means*) u konkurentnom okruženju.	30	5796.09 ± 100.18
12.	future-genetic	Optimizacija funkcija genetskim algoritmom korišćenjem biblioteke *Jenetics* [318].	50	14132.75 ± 78.24
13.	mnemonics	Rešavanje zadatka pamćenja brojeva telefona korišćenjem *JDK* strimova.	16	41063.38 ± 243.61
14.	par-mnemonics	Rešavanje zadatka pamćenja brojeva telefona korišćenjem paralelnih strimova.	16	33323.08 ± 179.94
15.	philosophers	Simulira problem filozofa koji večeraju [170] koristeći radni okvir *ScalaSTM* [33].	30	4640.16 ± 31.59
16.	reactors	Izvršava test sličan mikrobenchmarku *Savina* [134] sekvencijalno u radnom okviru za distribuirano programiranje *Reactors.IO* [262]	10	66946.44 ± 2485.89
17.	rx-scrabble	Rešava zagonetku *Scrabble* koristeći funkcionalno-reaktivne koncepte [214] i radni okvir *RxJava* [243].	80	1806.08 ± 5.86
18.	scala-doku	Rešava sudoku zagonetke korišćenjem jezika Skala.	20	23620.25 ± 99.52
19.	scala-kmeans	Koristi kolekcije u jeziku Skala za izvođenje algoritma k-sredina.	50	2375.70 ± 82.19
20.	scala-stm-bench7	Koristi radni okvir *ScalaSTM* za pokretanje test programa *STMBench7* [117].	60	7775.83 ± 65.02
21.	scrabble	Rešava zagonetku *Scrabble* koristeći Java strimove.	50	943.07 ± 9.00

Skup programa *DaCapo con Scala*, nije verzionisan

#	Program	Opis	n	Vreme izvrš. (ms)
22.	factorie	Koristi alat *Factorie* [229, 194] za probabilističko modelovanje i ekstrakciju tema iz teksta korišćenjem latentne Dirihleove alokacije [25, 26].	60	89362.71 ± 842.55
23.	kiama	Koristi biblioteku za obradu jezika u Skali, *Kiamu* [274], za kompajliranje i izvršavanje programa pisanih u jeziku *Oberon* [275] i jezicima koji proširuju jezik *ISWIM* [163].	40	2030.87 ± 59.83
24.	scalac	Koristi kompilator *Scalac* [304] za kompajliranje dekodera klasa *Scalap* [38].	12	7425.68 ± 143.23
25.	scaladoc	Koristi alat za generisanje dokumentacije u Skali [348] za izradu API dokumentacije za *Scalap*.	12	8685.27 ± 123.49
26.	scalap	Koristi Skala dekoder klasa za dekodiranje više klasa iz Skala biblioteke.	12	603.56 ± 9.82
27.	scalariform	Koristi alat *Scalariform* [348] za formatiranje izvornog koda Skala programa.	30	2533.38 ± 13.24
28.	scalaxb	Koristi XML alat *Scalaxb* [263] za kompilaciju XML šema [102] u Skala kôd.	60	2991.35 ± 9.33

Ponavljanje eksperimenata. Prilikom evaluacije uticaja statičkog profajlera *GraalSP* na performanse programa iz skupa za testiranje, svaki test program izvršavan je više puta, nakon

čega je izračunata aritmetička sredina izmerenih rezultata. Brojevi izvršavanja pojedinačnih test programa dati su uz opise test programa (sekcija 6.1). Na primer, kako bi se izmerilo vreme izvršavanja test programa *avrora* iz skupa programa *DaCapo*, program je izvršen 20 puta, a kao konačno vreme izvršavanja uzeta je srednja vrednost tih 20 merenja. Broj izvršavanja svakog test programa dat je u tabeli 6.1, u koloni označenoj sa n. Taj broj je odabran tako da programi koji pokazuju veću nestabilnost (na primer, usled nedeterminističkog ponašanja ili paralelnog izvršavanja) budu izvršavani više puta. Na ovaj način obezbeđena je veća pouzdanost i stabilnost dobijenih rezultata. Broj izvršavanja je prilagođen za svaki test program pojedinačno, pri čemu su u ovom radu korišćene podrazumevane vrednosti prilagođene kontinuiranom testiranju performansi kompilatora *GraalVM Native Image* u okviru platforme *GraalVM*. Iste vrednosti koriste se i u ranijim istraživanjima koja koriste ove skupove podataka [34].

Pored broja izvršavanja, u tabeli 6.1 prikazano je i izračunato prosečno vreme izvršavanja test programa u podrazumevanoj konfiguraciji kompilatora *Enterprise GraalVM Native Image*, kao i varijacija vremena izvršavanja. Ovi podaci nalaze se u koloni pod nazivom *Vreme izvrš. (ms)*.

Agregiranje rezultata. Prilikom agregiranja rezultata na nivou skupa programa za testiranje (*Renaissance*, *DaCapo* ili *DaCapo con Scala*), prikupljeni rezultati su skalirani u odnosu na podrazumevanu konfiguraciju kompilatora *Enterprise GraalVM Native Image*, nakon čega su skalirane vrednosti agregirane korišćenjem geometrijske sredine [191]. Za agregiranje rezultata (skaliranih vrednosti) korišćena je geometrijska sredina obzirom da je to preferiran metod za agregaciju skaliranih vrednosti i da korišćenje aritmetičke sredine za agregaciju skaliranih vrednosti može dovesti do pogrešnih zaključaka [96]. U nastavku teksta, kada se upotrebi termin *prosek* u kontekstu skaliranih vrednosti odnosiće se na agregiranje korišćenjem geometrijske sredine. Kako smo u ovoj disertaciji za agregiranje vrednosti skaliranih u odnosu na osnovnu konfiguraciju kompilatora koristili geometrijsku sredinu, za računanje odstupanja koristili smo geometrijsku standardnu devijaciju.

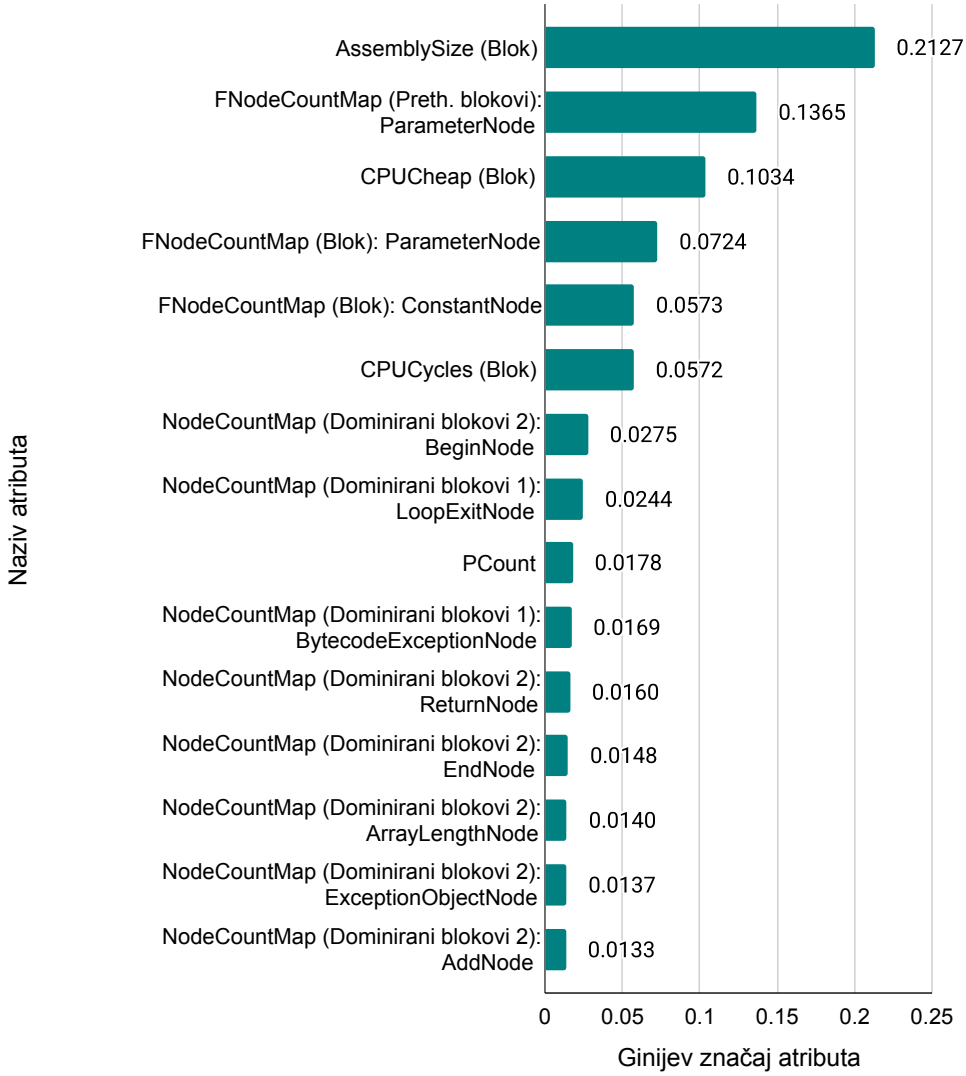
6.3 Analiza skupa atributa

U ovoj sekciji analizirani su atributi korišćeni za statičko predviđanje verovatnoća izvršavanja grana u programu. Najinformativniji atributi izdvojeni su primenom modela stabla odlučivanja. Pored toga, ispitivan je i uticaj broja atributa koji opisuju naredbe grananja na kvalitet modela. Za to je odabran model *XGBoost* kao kompromis između kvaliteta predviđanja i vremena potrebnog za obučavanje. Model stabla odlučivanja se lako obučava, ali je manje izražajan od modela *XGBoost*, dok je obučavanje dubokih neuronskih mreža previše zahtevno u pogledu računskih resursa da bi omogućilo izvođenje iterativnih eksperimenata, poput postepenog povećavanja broja atributa ili veličine skupa za obučavanje modela.

Najinformativniji atributi

Na slici 6.1 prikazano je 15 najinformativnijih atributa, identifikovanih na osnovu obučenog modela stabla odlučivanja i Ginijevog značaja atributa. Analiza pokazuje da su atributi koji su definisani u ovom radu a koji opisuju kôd niskog nivoa koji će biti generisan na osnovu *Graal IR* grafa veoma informativni i značajni za statičko predviđanje verovatnoća izvršavanja grana. Konkretno, procenjena veličina asemblerskog koda bloka koji sadrži čvor koji odgovara naredbi

grananja rangiran je kao najinformativniji atribut, dok su broj jeftinih CPU instrukcija i broj CPU ciklusa u tom bloku rangirani kao treći, odnosno šesti atribut po značaju.



Slika 6.1: Atributi kojima se opisuju naredbe grananja sortirani na osnovu Ginijevog značaja i obučenog modela stabla odlučivanja. Za atribute predstavljene rečnicima `NodeCountMap` i `FNodeCountMap` dat je i konkretan naziv čvora odnosno ključa rečnika.

Analiza značaja atributa identifikuje informativne komponente atributa predstavljenih rečnicima, `NodeCountMap` i `FNodeCountMap`. Na primer, stablo odlučivanja kao drugi najznačajniji atribut izdvaja broj *Graal IR* čvorova tipa `ParameterNode` u blokovima koji prethode bloku koji sadrži *cfs*-čvor naredbe grananja, dok je broj ovih čvorova u samom bloku koji sadrži *cfs*-čvor četvrti najznačajniji atribut. Čvorovi tipa `ParameterNode` predstavljaju parametre metoda i prilikom kreiranja grafa kontrole toka programa smeštaju se u prvi CFG blok metode. Značaj ovih atributa može se interpretirati kao činjenica da je važno identifikovati poziciju naredbe grananja u metodi, kao i to da li se ona pojavljuje na samom početku metode, ili kasnije.

Pored broja *Graal IR* čvorova tipa `ParameterNode`, analiza značaja atributa pokazala je da je broj *Graal IR* čvorova tipa `ConstantNode` u bloku u kome se nalazi *cfs*-čvor koji odgovara

naredbi grananja rangiran kao peti najznačajniji atribut. Ovaj rezultat ukazuje na to da prisustvo konstantnih vrednosti u relevantnim blokovima ima značajnu ulogu u modelovanju toka izvršavanja programa.

U okviru rečnika atributa `NodeCountMap`, broj čvorova tipa `LoopExitNode` u blokovima dominiranim prvom granom naredbe grananja identifikovan je kao osmi najinformativniji atribut po značaju. Takođe, broj *Graal IR* čvorova tipa `BeginNode` i `EndNode` u blokovima dominiranim drugom granom naredbe grananja rangirani su kao sedmi, odnosno dvanaesti najinformativniji atribut. Na osnovu ovih rezultata može se zaključiti da su čvorovi koji modeluju tok programa – poput početka ili kraja grane, kao i izlaska iz petlje – relevantni za precizno predviđanje verovatnoće izvršavanja pojedinačnih grana, odnosno za efikasno modelovanje toka izvršavanja programa.

Slično prethodnom, *Graal IR* čvorovi koji definišu prekide programa, kao što su broj *Graal IR* čvorova tipa `BytecodeExceptionNode` u blokovima dominiranim prvom granom naredbe grananja i broj *Graal IR* čvorova tipa `ReturnNode` i `ExceptionObjectsNode` u blokovima dominiranim drugom granom naredbe grananja, veoma su informativni atributi za model stabla odlučivanja. Ovo je u skladu sa ranijim istraživanjima u ovoj oblasti, koja su ručno definisala i izdvajala slične atribute radi opisivanja prekida izvršavanja unutar grana naredbi grananja [202, 254]. Na taj način, osim što omogućavaju otkrivanje novih atributa, atributi predstavljeni rečnicima mogu se koristiti i za potvrđivanje značaja ranije utvrđenih i korišćenih atributa za statičko predviđanje verovatnoća izvršavanja grana naredbi grananja.

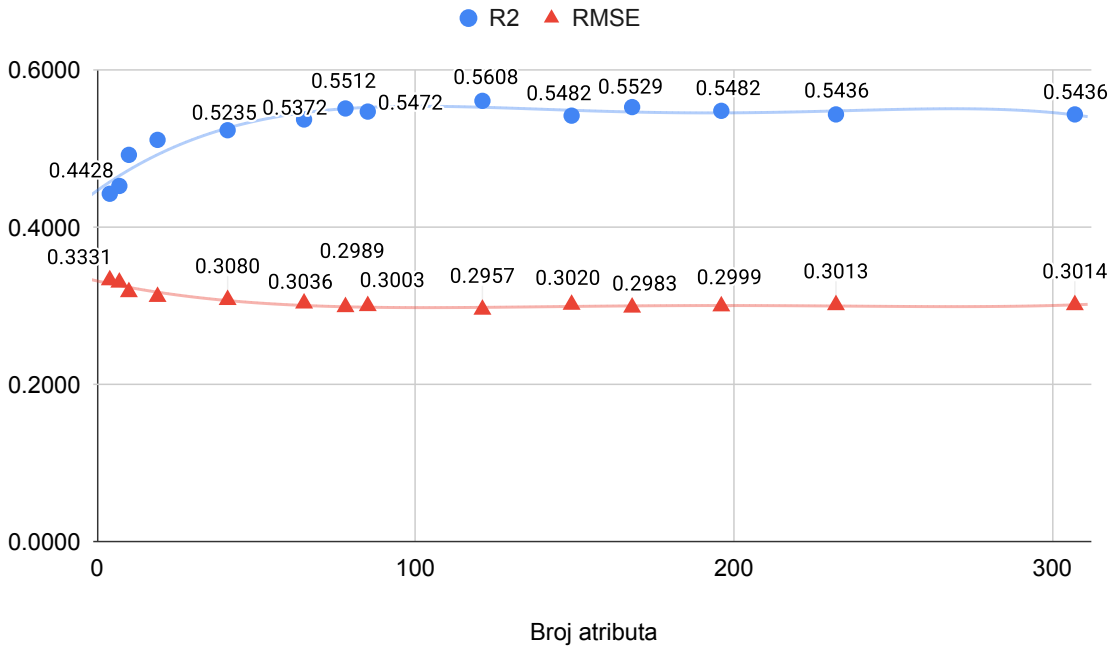
Odabir skupa atributa

Da bi se ispitao uticaj atributa kojima se opisuju naredbe grananja na kvalitet modela *XGBoost*, ovaj model je obučavan sa različitim brojem atributa, pri čemu je praćen kvalitet predviđanja modela na skupu za testiranje. Na slici 6.2 prikazane su promene srednjekvadratne greške i koeficijenta determinacije na test skupu u zavisnosti od broja atributa koji su korišćeni za obučavanje modela.

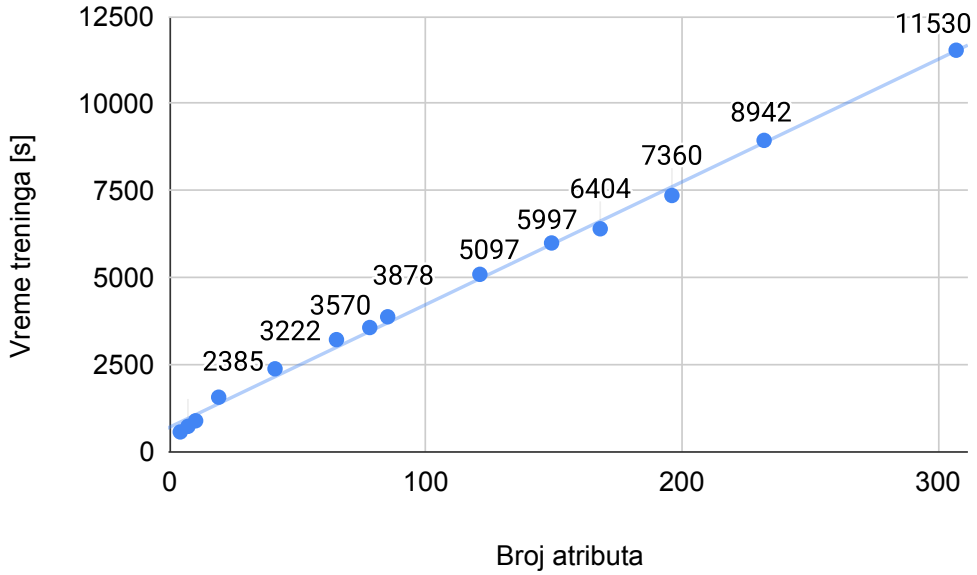
Prilikom odabira atributa za obučavanje modela uvek su korišćeni najinformativniji atributi, odabirom atributa na osnovu njihove varijanse. Korišćenje većeg broja atributa prilikom obučavanja modela poboljšava kvalitet modela, ali po cenu povećanja vremena obučavanja (prikazano na slici 6.3). Pored toga, korišćenje većeg broja atributa može dovesti do blagog preprilagođavanja modela, što se odražava kroz blago smanjenje koeficijenta determinacije i povećanja srednjekvadratne greške na skupu za testiranje modela u slučajevima kada model koristi više od 200 atributa.

6.4 Analiza skupa podataka za obučavanje modela

U ovoj sekciji izvršeno je ispitivanje uticaja veličine i sadržaja skupa za obučavanje modela na kvalitet modela. Pored toga, kvalitet modela analiziran je i u zavisnosti od težina instanci korišćenih prilikom obučavanja modela radi fokusiranja modela na značajnije instance podataka odnosno na frekventnije naredbe grananja. Kao i u prethodnoj sekciji, za izvođenje ovih eksperimenata korišćen je model *XGBoost*.



Slika 6.2: Uticaj veličine skupa atributa na kvalitet predviđanja modela na skupu za testiranje odnosno na vrednosti metrika srednjekvadratne greške i koeficijenta determinacije

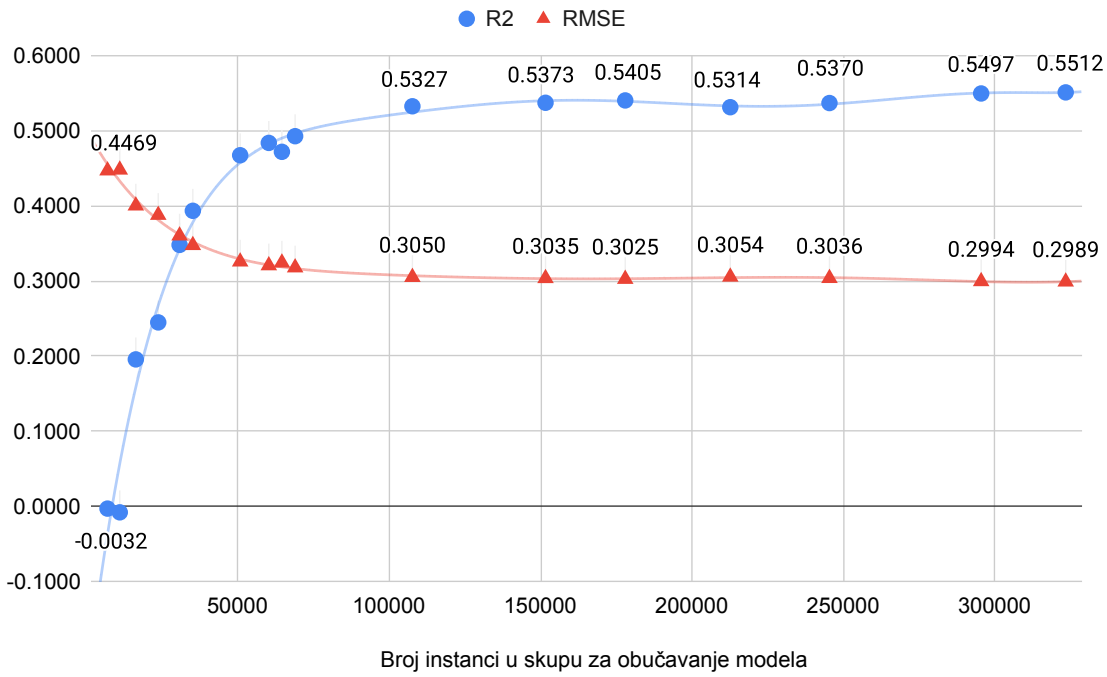


Slika 6.3: Uticaj veličine skupa atributa na trajanje obučavanja modela

Veličina skupa

Na slici 6.4 prikazan je uticaj postepenog dodavanja programa u skup za obučavanje modela na kvalitet predviđanja modela na skupu za testiranje odnosno na vrednosti metrika za ocenu kvaliteta regresionih modela na skupu za testiranje. Povećanje broja instanci u skupu za

obučavanje omogućava bolju generalizaciju modela, što dovodi do smanjenja vrednosti srednjekvadratne greške i povećanja vrednosti koeficijenta determinacije na test skupu. Ovi rezultati su u skladu sa sličnim eksperimentima sprovedenim na klasifikacionim modelima, gde je greška klasifikacije na test skupu opadala sa povećanjem veličine skupa za obučavanje modela [37].



Slika 6.4: Uticaj veličine skupa za obučavanje modela na kvalitet predviđanja modela na skupu za testiranje odnosno na vrednosti metrika srednjekvadratne greške i koeficijenta determinacije

Kada skup za obučavanje modela sadrži manje od 10 000 instanci, model ima poteškoća da generalizuje, što rezultira veoma visokom vrednošću srednjekvadratne greške od oko 0.45 i koeficijentom determinacije bliskom nuli. Kada se broj instanci kreće između 10 000 i 100 000, dodavanje novih instanci podataka u skup za obučavanje modela značajno doprinosi poboljšavanju kvaliteta modela. Nakon što skup za obučavanje modela premaši 100 000 instanci, dodatni podaci i dalje doprinose poboljšanju kvaliteta modela, ali u manjoj meri.

Sadržaj skupa

Skup za obučavanje modela sastoji se isključivo od programa pisanih u programskom jeziku Java. Kako bi se ocenio kvalitet modela i skupa atributa, pri radu sa programima pisanim u različitim programskim jezicima koji se kompajliraju na Java bajtkod, upoređena je greška predviđanja obučenog modela *XGBoost* na programima iz *DaCapo* skupa (koji sadrži isključivo programe pisane u programskom jeziku Java) i programima iz *DaCapo con Scala* skupa (koji se sastoji od programa pisanih u programskom jeziku Skala). Srednjekvadratna greška predviđanja modela *XGBoost* na programima iz *DaCapo* skupa iznosi 0.2950, dok za programe iz *DaCapo con Scala* skupa ona iznosi 0.3038. S druge strane, koreficijent determinacije računat na osnovu predviđanja modela na programima iz *DaCapo* skupa iznosi 0.5402, dok je koeficijent determinacije računat na osnovu predviđanja modela na programima iz *DaCapo con Scala* skupa 0.5664.

Zbog sličnih vrednosti srednjekvadratne greške i koeficijenta determinacije, teško je izvući opšte zaključke o kvalitetu predviđanja modela *XGBoost* na programe napisane u jezicima Java i Skala. Ovo je u skladu sa promenljivim performansama izvršavanja programa napisanim u programskom jeziku Skala u poređenju sa programima pisanim u programskom jeziku Java, budući da postoje primeri i boljih i lošijih ubrzanja kod Skala programa optimizovanih na osnovu profila predviđenih modelom *XGBoost* (videti odeljak 6.5). Zbog toga se može pretpostaviti da kvalitet predviđanja modela prvenstveno zavisi od karakteristika konkretnog programa, a ne od programskog jezika u kojem je napisan.

Težine instanci

S obzirom na to da je raspodela težina instanci izraženo pozitivno asimetrična, očekivano je da ona utiče kako na vreme obučavanja modela mašinskog učenja, tako i na kvalitet samog modela, fokusirajući model ka frekventnim instancama, odnosno instancama kojima su pridružene visoke vrednosti težina. Radi potvrde ove pretpostavke, obučen je model *XGBoost* sa istom konfiguracijom kao i originalni model, ali bez korišćenja težina instanci tokom obučavanja.

Model *XGBoost* obučen bez korišćenja težina instanci zauzima 6.9 MB, što je skoro 24 puta više u poređenju sa originalnim modelom *XGBoost* u čijem procesu obučavanja su korišćene težine instanci. Ova razlika nastaje zbog velikog broja instanci sa veoma malim težinama koje originalni model *XGBoost* u velikoj meri zanemaruje. Zbog toga što je model *XGBoost* obučen bez korišćenja težina instanci, on je znatno veći od originalnog modela, pa mu je i vreme predviđanja duže i u proseku iznosi 1.5 sekundi. Takođe, vreme njegovog obučavanja iznosi 4503 sekunde, što je 38% duže od vremena obučavanja modela *XGBoost* kada se koriste težine instanci.

Kada se model *XGBoost* obučen bez korišćenja težina instanci integriše u statički profajler, prosečno ubrzanje programa na programima skupa za testiranje iznosi 3.18%, u poređenju sa programima optimizovanim osnovnom verzijom kompilatora, kada se verovatnoće izvršavanja grana modeluju uniformnom raspodelom. Ovo je za 57% manje od prosečnog ubrzanja koje se postiže kada statički profajler za predviđanje profila koristi model *XGBoost* koji je obučen korišćenjem težina instanci i koji se stoga fokusira na frekventne naredbe grananja.

Model *XGBoost* obučen bez korišćenja težina instanci sve instance u skupu podataka za obučavanje tretira ravnopravno, i sa približno jednakom preciznošću predviđa oznake i za frekventne i za nefrekventne naredbe grananja. To rezultira prosečnom veličinom izvršivih fajlova koja je za 0.89% manja u odnosu na veličinu izvršivih fajlova programa optimizovanih osnovnom verzijom kompilatora.

Kako se prilikom obučavanja originalnog *XGBoost* modela koriste težine instanci, on ima tendenciju da zanemaruje naredbe grananja sa nižom frekvencijom i agresivno se fokusira na frekventne instance. To dovodi do promašaja prilikom predviđanja profila manje frekventnih instanci. Ti promašaji ne utiču na vreme izvršavanja optimizovanih programa jer ove instance ne utiču značajno na vreme izvršavanja programa. Ipak, greške u predviđanjima na ovakvim instancama mogu navesti optimizacije zasnovane na duplikacijama da agresivno dupliciraju na ovim mestima, tako da manje precizna predviđanja profila ovih instanci utiču na povećanje veličine izvršivih fajlova. Stoga *XGBoost* model obučavan bez korišćenja težina instanci ostvaruje bolje rezultate od originalnog modela kada je u pitanju veličina optimizovanih programa.

6.5 Evaluacija kvaliteta predviđanja modela

U ovoj sekciji prikazana je evaluacija modela stabla odlučivanja, modela *XGboost* i modela duboke neuronske mreže za predviđanje verovatnoća izvršavanja grana naredbi grananja. Prilikom razvoja statičkog profajlera *GraalSP* obučavani su i drugi modeli mašinskog učenja, poput modela linearne regresije i modela zasnovanih na širokom pojasu: metoda potpornih vektora za regresiju i algortima k najbližih suseda za regresiju. U ovom delu biće prikazana evaluacija samo onih modeli koji su ostvarili najbolje rezultate.

U tabeli 6.2 prikazane su vrednosti srednjekvadratne greške i koeficijenta determinacije koje su modeli stabla odlučivanja (engl. *decision tree*, skraćeno DT), duboke neuronske mreže (engl. *deep neural network*, skraćeno DNN) i gradijentnog pojačavanja (modela *XGBoost*) postigli na skupu podataka za testiranje. Pored toga, tabela sadrži i informacije o veličinama obučenih modela, vremenu potrebnom za njihovo obučavanje, kao i vremenu predviđanja modela (engl. *inference time*). Vreme obučavanja i vreme predviđanja izmereni su na laptopu koji je korišćen za razvoj statičkog profajlera *GraalSP*, čime se dobijaju realistične procene performansi koje mogu uočiti i krajnji korisnici.

Tabela 6.2: Poređenje različitih modela mašinskog učenja na osnovu vrednosti srednjekvadratne greške (RMSE) i koeficijenta determinacije (R^2) na skupu za testiranje modela, kao i veličine modela, vremena obučavanja i vremena predviđanja

Model	RMSE	R^2	Veličina (KB)	Vreme obučavanja (s)	Vreme predviđanja (s)
DT	0.3510	0.3812	19.9	243	0.1
DNN	0.3230	0.4758	3117.3	6459	3.0
XGBoost	0.2989	0.5512	289.5	3264	0.5

Model mašinskog učenja *XGBoost* nadmašuje modele stabla odlučivanja i duboke neuronske mreže, postižući najnižu srednjekvadratnu grešku i najviši koeficijent determinacije na test skupu. Pored toga, model *XGBoost* je više od deset puta manji od modela duboke neuronske mreže. Vremena obučavanja i predviđanja takođe značajno variraju. Kao što je i očekivano, vreme obučavanja modela stabla odlučivanja je najkraće, s vremenom obučavanja nešto dužim od 4 minuta. Nasuprot tome, obučavanje modela duboke neuronske mreže traje skoro dva sata, što je posledica mreže sa više od 800 000 parametara. Ansambl *XGBoost*, koji se sastoji od 1500 stabala, zahteva približno sat vremena za obučavanje. Vreme predviđanja modela *XGBoost* je oko šest puta kraće u poređenju sa vremenom predviđanja modela duboke neuronske mreže.

Da bi se ocenio uticaj statičkih profajlera zasnovanih na modelima mašinskog učenja na performanse programa optimizovanih prema predviđenom profilu, statički profajleri koji koriste model stabla odlučivanja, duboku neuronsku mrežu i model *XGBoost* integrisani su u kompilator *Enterprise GraalVM Native Image*. Evaluacija kvaliteta modela mašinskog učenja dobijena je upoređivanjem:

- kao osnovne verzije, podrazumevane konfiguracije kompilatora *Enterprise GraalVM Native Image*, koja koristi uniformnu raspodelu da modeluje verovatnoće izvršavanja grana naredbi grananja,
- kompilatora *Enterprise GraalVM Native Image* koji koristi statički profajler zasnovan na modelu stabla odlučivanja za predviđanje verovatnoća izvršavanja grana naredbi grananja,

- kompilatora *Enterprise GraalVM Native Image* koji koristi statički profajler zasnovan na modelu duboke neuronske mreže za predviđanje verovatnoća izvršavanja grana naredbi grananja i
- kompilatora *Enterprise GraalVM Native Image* koji koristi statički profajler zasnovan na modelu mašinskog učenja *XGBoost* za predviđanje verovatnoća izvršavanja grana naredbi grananja.

Rezultati poređenja prikazani su u tabeli 6.3. Konfiguracije kompilatora prikazane u narednim tabelama i slikama označene su prema modelu mašinskog učenja koji se koristi za predviđanje verovatnoće izvršavanja grana: DT označava stablo odlučivanja, DNN označava duboku neuronsku mrežu, a XGBoost označava model *XGBoost*. Prikazani rezultati su uvek dati u procentima u odnosu na osnovnu konfiguraciju kompilatora.

Tabela 6.3: Poređenje statičkih profajlera prema uticaju na vreme izvršavanja programa, veličinu generisanih izvršivih fajlova i vreme kompilacije programa. Vrednosti u tabeli izražene su kao procenti u odnosu na osnovnu (podrazumevanu) konfiguraciju kompilatora.

Model	Ubrzanje izvršavanja	Povećanje veličine izvršivih fajlova	Povećanje vremena kompilacije
DT	4.91	2.44	4.04
DNN	5.16	2.44	3.20
XGBoost	5.22	2.44	2.17

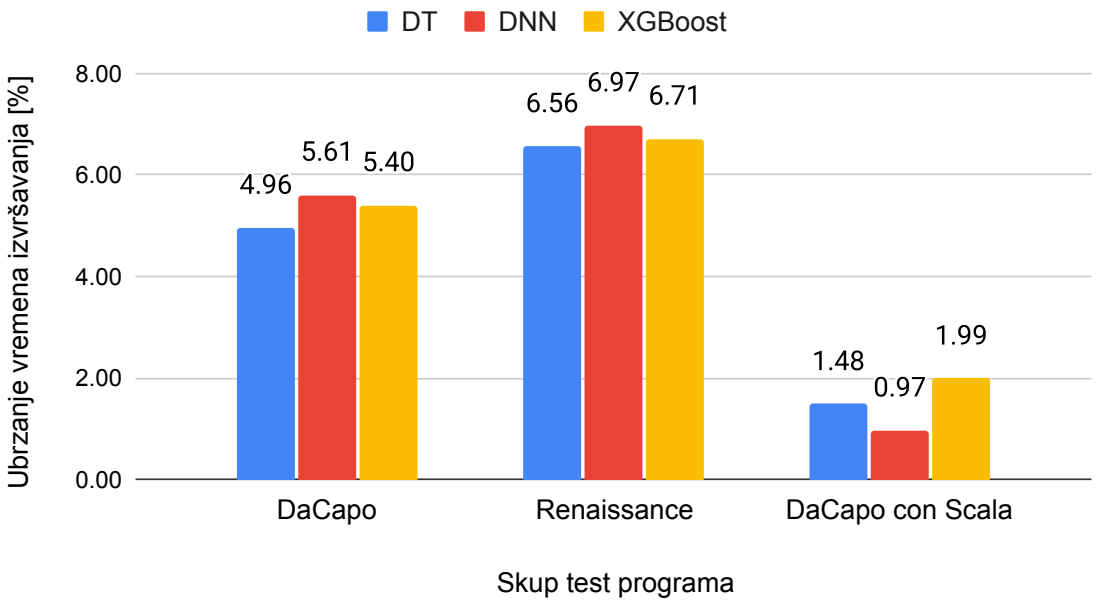
Trajanje izvršavanja programa

Najveće ubrzanje programa, od 5.22%, ostvareno je upotrebom statičkog profajlera koji koristi model *XGBoost* za predviđanje verovatnoća izvršavanja grana naredbi grananja (tabela 6.3). Na slici 6.5 prikazan je uticaj statičkih profajlera na vreme izvršavanja programa iz skupa programa za testiranje modela. U tabeli 6.4 prikazana je geometrijska standardna devijacija prilikom agregiranja skaliranih vremena izvršavanja programa kroz različite skupove test programa.

Programi napisani u jezicima Java i Skala prevode se u Java bajtkod i mogu se izvršavati na Java virtuelnoj mašini. Ipak, iako se izvršavaju na istoj platformi, ovi jezici se značajno razlikuju. Na primer, Skala je u većoj meri zasnovana na funkcionalnoj paradigmi, za razliku od klasičnih Java programa. Zbog toga nije iznenađujuće što modeli postižu znatno lošije rezultate na programima iz skupa *DaCapo con Scala*, budući da u skupu za obučavanje modela nema aplikacija napisanih u programskom jeziku Skala. Model *XGBoost* pokazuje bolju sposobnost generalizacije u poređenju sa modelom duboke neuronske mreže, pri čemu na programima iz skupa *DaCapo con Scala* ostvaruje približno dvostruko bolje rezultate od modela duboke neuronske mreže.

Tabela 6.4: Geometrijska standardna devijacija vremena izvršavanja programa

Skup programa	DT	DNN	XGBoost
DaCapo	1.08	1.04	1.04
Renaissance	1.12	1.10	1.10
DaCapo con Scala	1.06	1.05	1.07



Slika 6.5: Prosečno ubrzanje vremena izvršavanja programa ostvareno na osnovu predviđanja statičkih profajlera koji koriste modele mašinskog učenja za predviđanje verovatnoća izvršavanja grana

Uzimajući u obzir raspodelu podataka u skupu za obučavanje modela, programi za testiranje *scala-doku*, *scala-kmeans*, *scalap* i *scalaxb* predstavljaju odudarajuće podatke. Statički profajleri vođeni obučenim modelima mašinskog učenja vode do usporenja izvršavanja ovih test programa u rasponu od 0.27% do 30.48%. Iako su svi ovi programi pisani u programskom jeziku Skala, važno je naglasiti da modeli mogu pokazati dobre performanse i na programima pisanim u programskom jeziku Skala. Na primer, statički profajler koji koristi model *XGBoost* ubrzava izvršavanje test programa *kiama* (iz skupa test programa *DaCapo con Scala*), Skala biblioteke za obradu jezika [274] za 13.54%.

Veličina izvršivih fajlova

Svi modeli mašinskog učenja slično utiču na veličinu izvršivih fajlova optimizovanih programa, dovodeći do prosečnog povećanja veličine izvršivih fajlova od 2.44% (tabela 6.3). Prilikom integracije obučeni modeli postaju deo kompilatora *Enterprise GraalVM Native Image*, a ne deo optimizovanih izvršivih fajlova. Zbog toga veličina modela utiče na veličinu kompilatora, ali ne i na veličinu optimizovanih izvršivih fajlova.

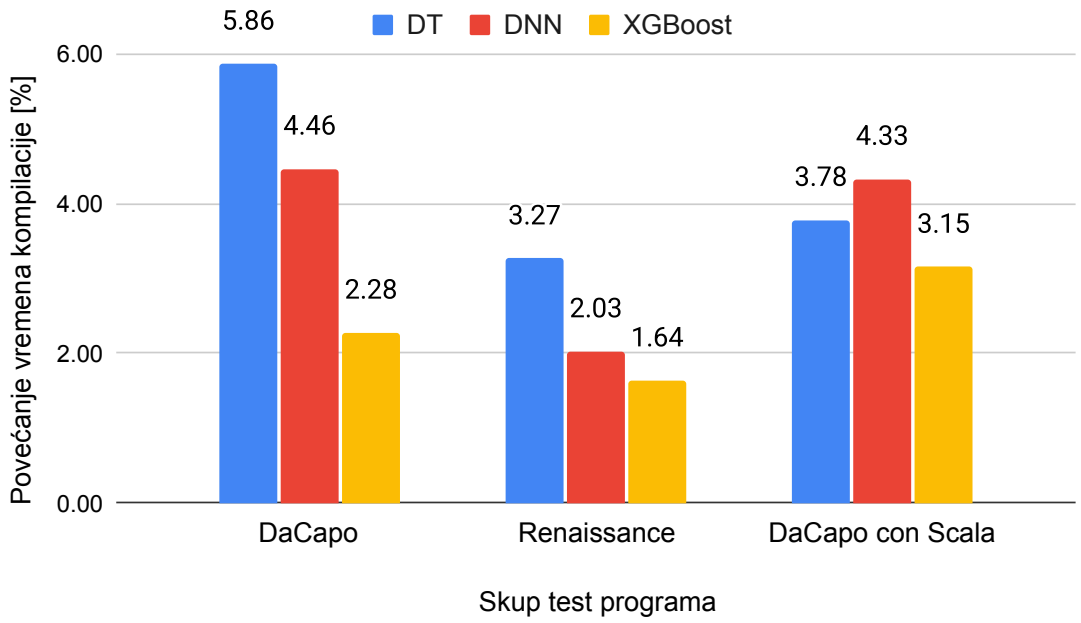
Statički profajler utiče na veličinu optimizovanih izvršivih fajlova jer se optimizacije (npr. duplikacije koda) vrše na osnovu predviđenih verovatnoća izvršavanja grana naredbi grananja. U podrazumevanom scenariju kompilacije, kompilator pretpostavlja uniformnu raspodelu verovatnoća izvršavanja grana naredbi grananja, što dovodi do manje agresivne duplikacije koda [176] i manjih izvršivih fajlova.

Razlike u veličini optimizovanih izvršivih fajlova za različite modele mašinskog učenja su manje od 0.01%. Prosečno povećanje veličine izvršivih fajlova iznosi 2.70% za programe iz skupa programa *DaCapo*, 2.59% za programe iz skupa *Renaissance* i 1.88% za programe iz skupa *DaCapo con Scala*. Iako modeli mašinskog učenja ostvaruju najmanje prosečno ubrzanje izvr-

šavanja programa na programima iz skupa *DaCapo con Scala*, oni takođe uzrokuju i najmanje povećanje veličine izvršivih fajlova na programima iz tog skupa.

Trajanje kompilacije programa

Trajanje procesa kompilacije uz korišćenje statičkog profajlera zasnovanog na modelima mašinskog učenja zavisi od trajanja izdvajanja atributa, trajanja predviđanja modela i kvaliteta predviđenih profila. Statički profajler koji verovatnoće izvršavanja grana predviđa korišćenjem modela *XGBoost* dovodi do najmanjeg povećanja vremena kompilacije programa (tabela 6.3). Na slici 6.6 prikazan je uticaj statičkih profajlera na vreme kompilacije programa agregiran prema skupu test programa, dok tabela 6.5 prikazuje geometrijsku standardnu devijaciju agregiranja ovih rezultata kroz različite skupove test programa.



Slika 6.6: Uticaj statičkih profajlera na vreme kompilacije programa, pri čemu se za predviđanje verovatnoća izvršavanja grana koriste modeli stabla odlučivanja (DT), duboke neuronske mreže (DNN) i model *XGBoost*

Tabela 6.5: Geometrijska standardna devijacija vremena kompilacije programa

Skup programa	DT	DNN	XGBoost
DaCapo	1.03	1.05	1.03
Renaissance	1.02	1.05	1.05
DaCapo con Scala	1.04	1.08	1.08

Model *XGBoost* dovodi do najmanjeg prosečnog povećanja vremena kompilacije na sva tri skupa test programa. Kod modela duboke neuronske mreže, glavni nedostatak u kontekstu povećanja vremena kompilacije jeste skupo izvođenje predviđanja modela, s obzirom na to da model sadrži više od 800 000 parametara. S druge strane, modeli zasnovani na stablima odlučivanja imaju niže vreme predviđanja zbog svoje jednostavne strukture. U slučaju modela stabla

odlučivanja, do povećanja vremena kompilacije dolazi zbog nešto lošijeg kvaliteta predviđenog profila u poređenju sa drugim modelima. Takav profil može usmeriti optimizacije ka drugačijim delovima programa nego što bi to učinio profil predviđen korišćenjem modela duboke neuronske mreže i modela *XGBoost*.

6.6 Evaluacija statičkog profajlera *GraalSP*

Evaluacija modela mašinskog učenja za statičko predviđanje verovatnoća izvršavanja grana naredbi grananja pokazala je da model *XGBoost* daje najbolje rezultate od svih obučavanih modela. Ovaj model ostvaruje najveće ubrzanje izvršavanja programa, pokazuje najbolju sposobnost generalizacije i najmanje povećava prosečno vreme kompilacije programa. Zbog toga ćemo u nastavku teksta na statički profajler *GraalSP* referisati kao na statički profajler koji koristi upravo ovaj model za predviđanje verovatnoća izvršavanja grana naredbi grananja i statičke heuristike opisane u odeljku 4.3 za korekcije predviđanja modela.

Kako bi se ocenio statički profajler *GraalSP* i uporedio sa relevantnim pristupima, u ovoj sekciji upoređeni su:

Programi optimizovani osnovnom verzijom kompilatora – programi optimizovani kompilatorom *Enterprise GraalVM Native Image*, koji koristi uniformnu raspodelu da modeluje verovatnoće izvršavanja grana naredbi grananja.

Vu-Larus optimizovani programi – programi optimizovani kompilatorom *Enterprise GraalVM Native Image*, koji koristi statički profajler zasnovan na heuristikama Vua i Larusa (implementacija heuristika opisana je u Dodatku A) za predviđanje verovatnoća izvršavanja grana naredbi grananja.

***XGBoost* optimizovani programi** – programi optimizovani kompilatorom *Enterprise GraalVM Native Image*, koji koristi model *XGBoost* za predviđanje verovatnoća izvršavanja grana naredbi grananja.

***GraalSP* optimizovani programi** – programi optimizovani kompilatorom *Enterprise GraalVM Native Image*, koji koristi statički profajler *GraalSP* za predviđanje verovatnoća izvršavanja grana naredbi grananja.

Dinamički optimizovani programi – programi optimizovani kompilatorom *Enterprise GraalVM Native Image*, koji koristi dinamički profajler zasnovan na instrumentaciji za modelovanje verovatnoća izvršavanja grana naredbi grananja.

U tabeli 6.6 prikazani su rezultati evaluacije uticaja ovih profajlera na trajanje izvršavanja optimizovanih programa, veličinu izvršivih fajlova i trajanje kompilacije programa iz skupa programa za testiranje. Konfiguracije kompilatora u narednim tabelama i slikama označene su prema korišćenom profajleru. Prikazani rezultati su izraženi u procentima u odnosu na osnovnu konfiguraciju kompilatora.

Trajanje izvršavanja programa

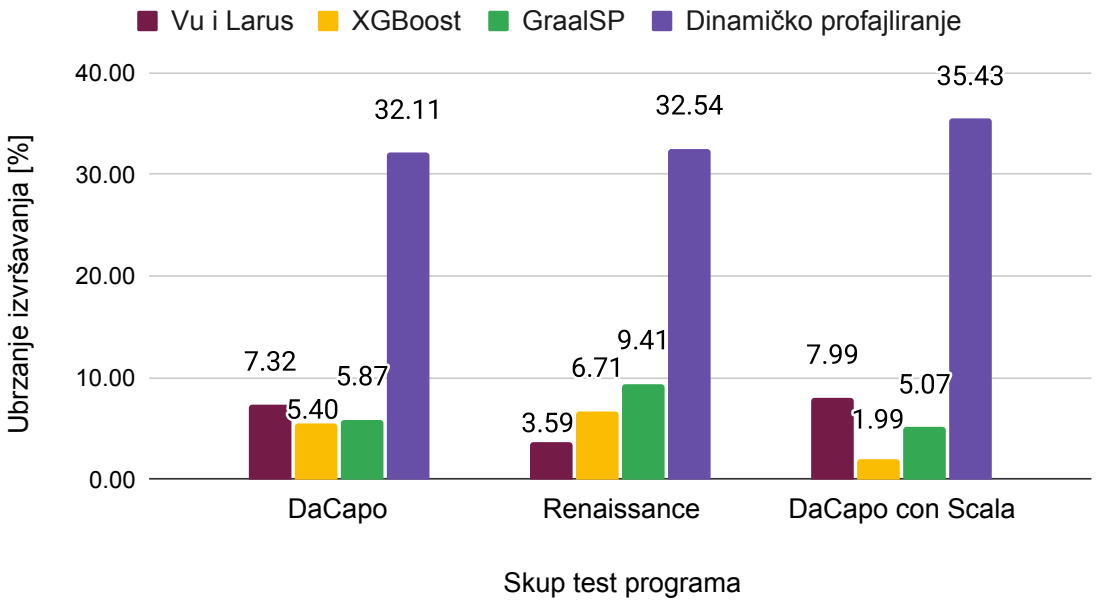
Vreme izvršavanja *GraalSP* optimizovanih programa u proseku je za 7.46% kraće od vremena izvršavanja programa optimizovanih osnovnom verzijom kompilatora (tabela 6.6). Ovo

Tabela 6.6: Poređenje uticaja statičkih profajlera na trajanje izvršavanja programa, veličinu izvršivih fajlova i trajanje kompilacije programa. Vrednosti u tabeli izražene su kao procenti u odnosu na osnovnu (podrazumevanu) konfiguraciju kompilatora.

Profajler	Ubrzanje izvršavanja	Povećanje veličine izvršivih fajlova	Povećanje vremena kompilacije
Vu i Larus	5.64	31.60	78.51
XGBoost	5.22	2.44	2.17
GraalSP	7.46	3.91	12.01
Dinamički profajler	33.17	-14.57	-10.98

ubrzanje je za 2.24% veće od ubrzanja koje ostvaruju *XGBoost* optimizovani programi u odnosu na programe optimizovane osnovnom verzijom kompilatora. Dodatnih 2.24% poboljšanja rezultat su heuristika za korekciju predviđanja modela mašinskog učenja, koje predstavljaju kompromis između performansi izvršavanja programa i povećanja veličine izvršivih fajlova i povećanja vremena kompilacije programa.

Na slici 6.7 prikazana su poboljšanja performansi koja se ostvaruju integracijom navedenih profajlera u kompilator *Enterprise GraalVM Native Image* na programima iz skupova programa za testiranje. Tabela 6.7 prikazuje geometrijsku standardnu devijaciju agregiranja vremena izvršavanja programa. Heuristike za korekciju predviđanja verovatnoća izvršavanja grana naredbi grananja poboljšavaju performanse modela *XGBoost* za 0.47%, 2.70% i 3.08% na skupovima programa za testiranje *DaCapo*, *Renaissance* i *DaCapo con Scala*, redom.



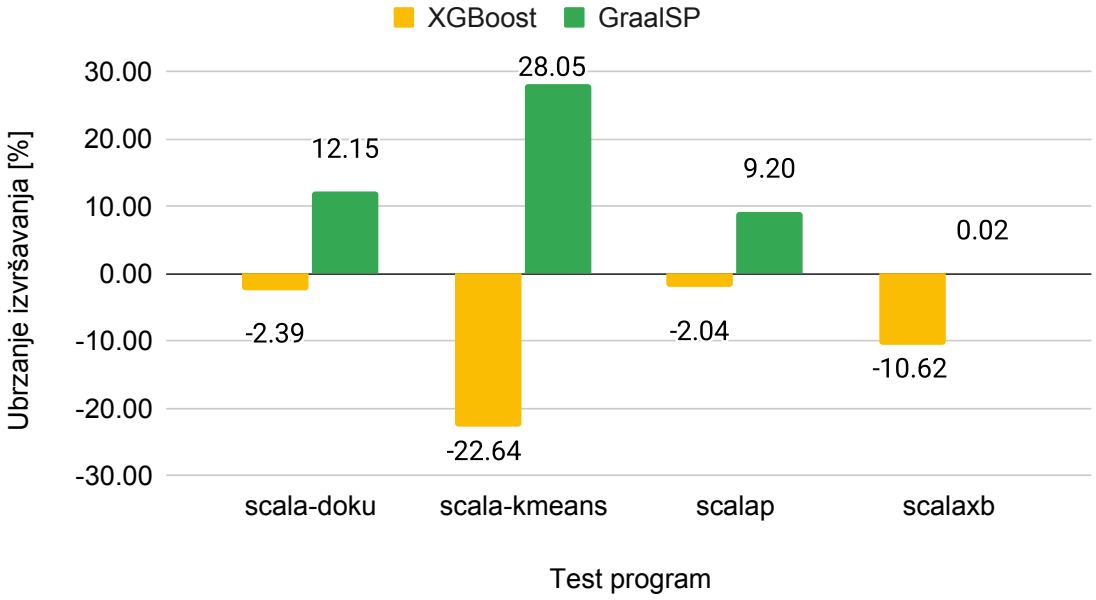
Slika 6.7: Prosečno ubrzanje izvršavanja optimizovanih programa

Slika 6.8 prikazuje uticaj statičkog profajlera koji koristi model mašinskog učenja *XGBoost*, kao i statičkog profajlera *GraalSP*, na performanse izvršavanja test programa `scala-doku`, `scala-kmeans`, `scalap` i `scalaxb`. Ovi test programi napisani su u programskom jeziku Skala, a statički profajler koji za predviđanje profila koristi model *XGBoost* dovodi do usporenja izvršavanja ovih programa u odnosu na to kada se ovi programi prevedu korišćenjem osnovne verzije

Tabela 6.7: Geometrijska standardna devijacija pri agregiranju rezultata prilikom računanja uticaja profajlera na trajanje izvršavanja programa

Skup test programa	Vu i Larus	XGBoost	GraalSP	Dinamički profajler
DaCapo	1.05	1.04	1.04	1.16
Renaissance	1.17	1.10	1.08	1.26
DaCapo con Scala	1.08	1.07	1.03	1.18

kompilatora. Profajler *GraalSP* uspešno prevazilazi ova usporenja korišćenjem statičkih heuristika koje služe za korekciju predviđanja modela. U zavisnosti od konkretnog programa i toga koliko pogrešno predviđene petlje i grane utiču na vreme njegovog izvršavanja, *GraalSP* uspeva da performanse vrati na nivo osnovne konfiguracije kompilatora (kao kod test programa `scalaxb`) ili čak da ih poboljša (kao u slučajevima test programa `scala-doku`, `scala-kmeans` i `scalap`).



Slika 6.8: Uticaj statičkog profajlera koji koristi model *XGBoost* za predviđanje profila i statičkog profajlera *GraalSP* na performanse izvršavanja test programa koji predstavljaju odudarajuće podatke

Vu-Larus optimizovani programu su za, u proseku, 5.64% brži u odnosu na programe optimizovane osnovnom verzijom kompilatora. Ovo ubrzanje je za 1.82% manje od ubrzanja koje ostvaruju *GraalSP* optimizovani programi. Iako na skupovima test programa *DaCapo* i *DaCapo con Scala* (Slika 6.7) Vu-Larus optimizovani programi imaju bolje performanse izvršavanja u odnosu na *GraalSP* optimizovane programe, ti rezultati dolaze po visokoj ceni: statički profajler zasnovan na heuristikama Vua i Larusa dovodi do generisanja znatno većih izvršivih fajlova (odeljak 6.6) i zahteva znatno više vremena za kompilaciju programa (tabela 6.6).

Kompilator *Enterprise GraalVM Native Image* koji koristi dinamički prikupljen profil izvršavanja grana naredbi grananja postavlja gornju granicu performansi koje se mogu postići korišćenjem statičkog profajliranja (poboljšanje od 33.17%, tabela 6.6). Iako ovakav vid dina-

mičkog profajliranja ne sadrži eksplicitne informacije o izvršavanim metodama, on ih implicitno uključuje jer dinamički profajler prikuplja profil izvršavanja grana naredbi grananja isključivo iz izvršenih metoda. Kao rezultat toga, kompilator primenjuje agresivnije optimizacije na tim metodama za koje je i prikupljen profil i postiže bolje rezultate.

Na slici 6.9 prikazan je uticaj ocenjivanih profajlera na performanse izvršavanja pojedinačnih programa iz skupa programa korišćenih za evaluaciju modela. Vu-Larus optimizovani programi, zbog agresivnosti heuristika, mogu ostvariti značajna ubrzanja. Na primer, ubrzanje od 30.11% zabeleženo je na test programu *scala-kmeans* iz skupa test programa *Renaissance*, dok je ubrzanje od 21.12% ostvareno na programu *scalaxb* iz skupa test programa *DaCapo con Scala*. S druge strane, moguća su i značajna usporenja poput usporenja od 49.30% na test programu *scala-doku*, takođe iz skupa test programa *DaCapo con Scala*. Iako je to jedini slučaj usporenja među testiranim programima, ono je drastično.

XGBoost optimizovani programi ostvaruju značajna ubrzanja programa, do 13.06% i 11.61% na test programima *fj-kmeans* i *future-genetic* iz skupa test programa *Renaissance*. *XGBoost* optimizovani programi mogu biti i sporiji od programa optimizovanih osnovnom verzijom kompilatora. Takva usporenja zabeležena su, na primer, kod test programa *scala-kmeans* i *scala-doku*, kao i kod test programa *scalaxb*.

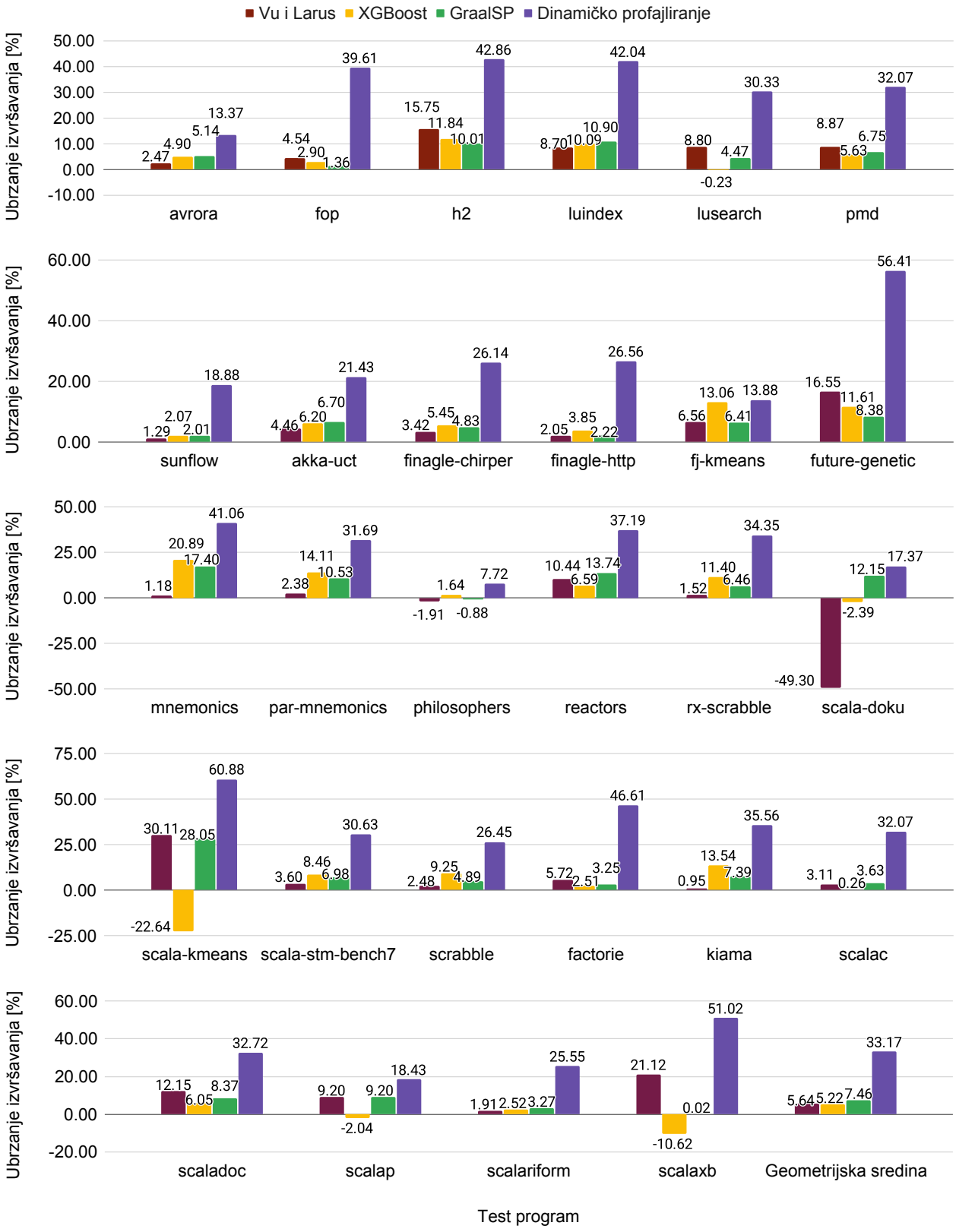
GraalSP optimizovani programi ostvaruju ubrzanja do 28.05% na test programu *scala-kmeans*, 13.74% na test programu *reactors* i 12.15% na test programu *scala-doku*, sva tri iz skupa test programa *Renaissance*. Takođe, *GraalSP* optimizovani programi ostvaruju značajna ubrzanja i na programima iz drugih test skupova. Na primer, ubrzanje od 9.20% na test programu *scalap* iz skupa test programa *DaCapo con Scala* ili 10.90% i 10.01% redom na test programima *luindex* i *h2*, oba iz skupa test programa *DaCapo*. Značajno je i to što *GraalSP* optimizovani programi ni na jednom od test programa ne unose značajno usporenje izvršavanja. Jedini program na kome je primećeno usporenje jeste test program *philosophers* iz skupa test programa *Renaissance* ali je ovo usporenje od 0.88% minimalno. Na svim drugim test programima performanse *GraalSP* optimizovanih programa bolje su od performansi programa optimizovanih osnovnom verzijom kompilatora.

Dinamički optimizovani programi ostvaruju najveća ubrzanja i ona dosežu do čak 60.88% na programu *scala-kmeans* i 56.41% na programu *future-genetic*, oba iz skupa test programa *Renaissance*. Na skupu test programa ubrzanja dinamički optimizovanih programa uglavnom su bila dvocifrena. Jedini dinamički optimizovan program na kom je izmereno jednocifreno ubrzanje jeste test program *philosphers* iz skupa test programa *Renaissance*. Ubrzanje dinamički optimizovane verzije ovog test programa iznosi 7.72%.

Veličina izvršivih fajlova

Veličina izvršivih fajlova *GraalSP* optimizovanih programa je za 3.91% veća od veličine izvršivih fajlova programa optimizovanih osnovnom verzijom kompilatora (tabela 6.6). Dodatno povećanje veličine izvršivih fajlova od 1.47% u poređenju sa *XGBoost* optimizovanim programima, posledica je heuristika za korekciju predviđanja modela. Ovo je očekivano ponašanje: heuristike koje koristi statički profajler *GraalSP* ne dozvoljavaju da verovatnoća izvršavanja tela petlje bude manja od 0.2, što navodi kompilator da izvrši dodatne optimizacije zasnovane na duplikaciji u telima petlji i time generiše veće programe.

Veličina izvršivih fajlova Vu-Larus optimizovanih programa je za, u proseku, 31.60% veća od veličine izvršivih fajlova programa optimizovanih osnovnom verzijom kompilatora. Do povećanja veličine izvršivih fajlova dolazi zbog optimizacija zasnovanih na duplikaciji i činjenice da heuristike za predviđanje verovatnoća izvršavanja grana agresivno favorizuju tela petlji: tri

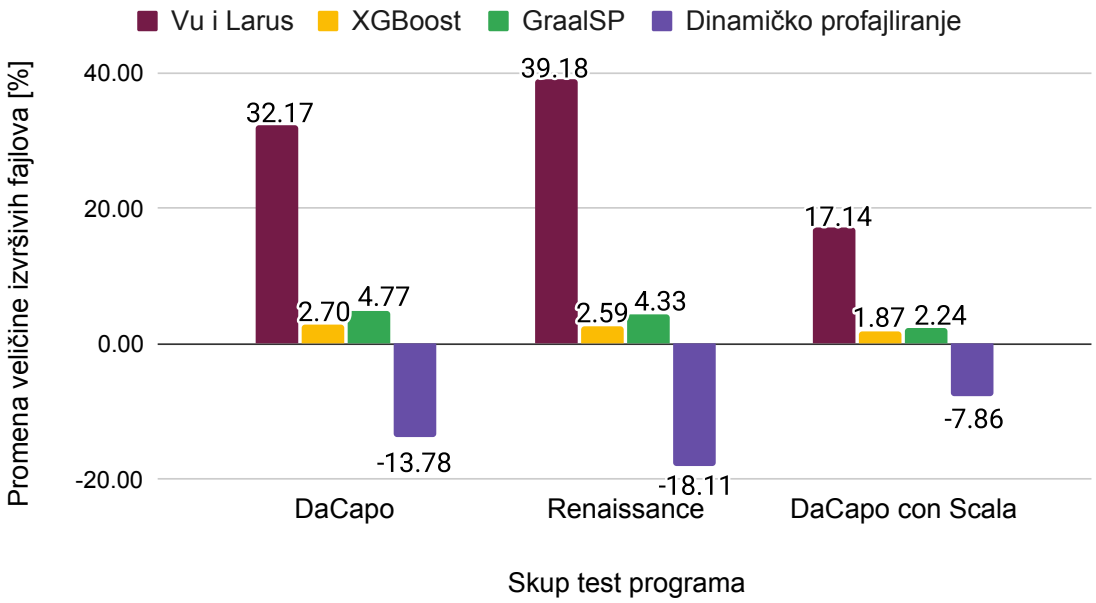


Slika 6.9: Uticaji profajlera na vreme izvršavanja optimizovanih programa

od devet heuristika koje su predložili Vu i Larus predviđaju izvršavanje tela petlje sa visokim verovatnoćama. Heuristika grananja, heuristika izlaza iz petlje i heuristika početka petlje predviđaju izvršavanje tela petlje sa verovatnoćama od 0.88, 0.80 i 0.75, redom. Heuristike Vua i Larusa dovode do znatno većeg povećanja veličine izvršivih fajlova optimizovanih programa u poređenju sa heuristikama koje koristi statički profajler *GraalSP*. Razlog tome je što heuristike Vua i Larusa dodeljuju mnogo veću verovatnoću izvršavanja telima petlji nego heuristika petlje koju koristi statčki profajler *GraalSP*.

Veličina izvršivih fajlova dinamički optimizovanih programa u proseku je manja za 14.57% u poređenju sa veličinom izvršivih fajlova programa optimizovanih osnovnom verzijom kompilatora. Do ovog smanjenja dolazi zato što kompilator vrši duplikacije koda isključivo u delovima programa koji će biti izvršeni prilikom izvršavanja programa, koristeći tačne, dinamički prikupljene frekvencije izvršavanja grana.

Na slici 6.10 prikazan je uticaj profajlera na veličinu izvršivih fajlova optimizovanih programa po skupovima programa za testiranje, dok tabela 6.8 prikazuje geometrijsku standardnu devijaciju izračunatu prilikom agregiranja rezultata na nivou skupova programa za testiranje modela. Dinamički profajler nadmašuje statičke profajlere i na svakom od skupova programa za testiranje dovodi do generisanja manjih izvršivih fajlova. Nasuprot tome, statički profajler koji koristi heuristike Vua i Larusa vodi do generisanja znatno većih izvršivih fajlova u poređenju sa ostalim profajlerima. Na skupu programa za testiranje *Renaissance* ovo povećanje veličine izvršivih fajlova iznosi čak 39.18%.



Slika 6.10: Prosečan uticaj na veličinu izvršivih fajlova optimizovanih programa

Na slici 6.11 prikazan je uticaj ocenjivanih profajlera na veličinu izvršivih fajlova pojedinačnih programa iz skupa programa korišćenih za evaluaciju modela. Vu-Larus optimizovani programi konzistentno su veći od programa optimizovanih osnovnom verzijom kompilatora. Oni unose značajna uvećanja veličine izvršivih fajlova optimizovanih programa. Ova uvećanja idu čak i do uvećanja od 47.93% na test programu *akka-uct* i 43.31% na test programu *finagle-chirper*, oba iz skupa test programa *Renaissance*. Ipak, uvećanje veličine izvršivih fajlova nešto

Tabela 6.8: Geometrijska standardna devijacija pri agregiranju rezultata prilikom računanja uticaja profajlera na veličinu izvršivih fajlova programa

	Vu i Larus	XGBoost	GraalSP	Dinamički profajler
DaCapo	1.03	1.01	1.02	1.20
Renaissance	1.04	1.01	1.02	1.26
DaCapo con Scala	1.19	1.01	1.01	1.11

je manje kod programa iz skupa test programa *DaCapo con Scala*. Na primer, to su uvećanja od 6.05% na test programu *factorie* i 2.50% na test programu *scalariform*.

XGBoost optimizovani programi konzistento su veći od programa optimizovanih osnovnom verzijom kompilatora. U većini slučajeva, to uvećanje iznosi između 1% i 3%. Najmanje uvećanje veličine optimizovanog programa koje ostvaruju *XGBoost* optimizovani programi jeste uvećanje od 1.03%, i to u slučaju test programa *factorie*, iz skupa test programa *Renaissance*. Poređenja radi, uvećanje veličine izvršivog fajla *GraalSP* optimizovane verzije programa *factorie* iznosi 1.53%. Prethodno uvećanje od pola procenta nešto je manje od uobičajenog uvećanja veličine *GraalSP* optimizovanih programa. *GraalSP* optimizovani programi su za u proseku jedan do dva procenta veći od *XGBoost* optimizovanih programa. Njihovo povećanje veličine izvršivih fajlova ulavnom se kreće između 3% i 5%, u odnosu na osnovnu verziju kompilatora. Ipak, uvećanje veličine *GraalSP* optimizovanih programa može ići i do 6.44%, koliko iznosi uvećanje na test programu *fop* iz skupa programa *DaCapo* i 6.31% koliko iznosi uvećanje na test programu *sunflow*, takođe iz skupa test programa *DaCapo*.

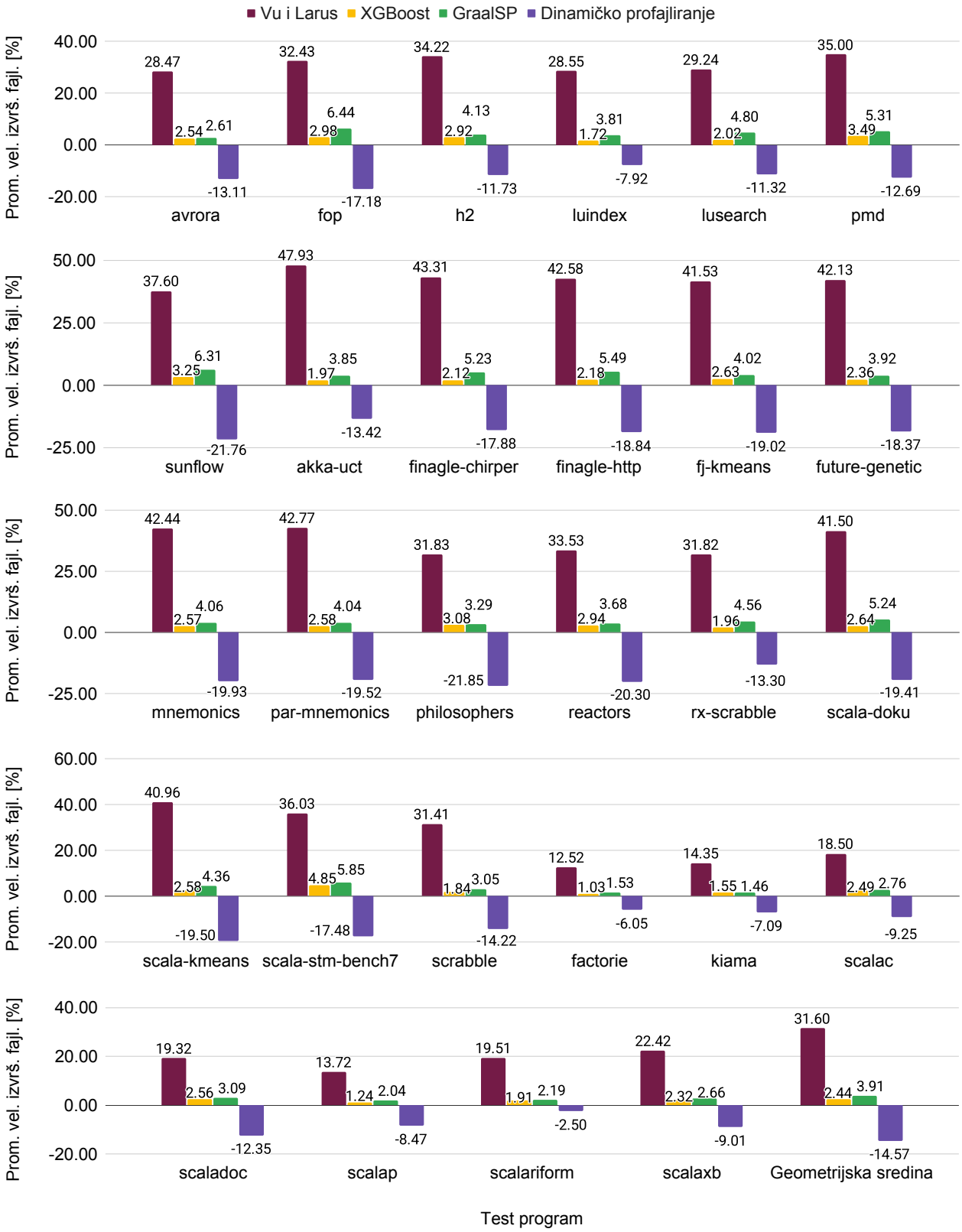
Dinamički optimizovani programi konzistetno su manji od programa optimizovanih osnovnom verzijom kompilatora. Na većini test programa ovo smanjenje veličine izvršivih fajova kreće se između 10% i 20%. Najveće smanjenje velićine izvršivog fajla optimizovanog programa ostvareno je na test programu *sunflow* iz skupa test programa *DaCapo* i ono iznosi 21.76%.

Trajanje kompilacije programa

U poređenju sa kompilacijom programa sa osnovnom verzijom kompilatora, upotreba statičkog profajlera *GraalSP* povećava vreme kompilacije za, u proseku, 12.01% (tabela 6.6). Statički profajler koji koristi model *XGBoost* za predviđanje profila povećava vreme kompilacije u proseku za 2.17%, takođe u odnosu na osnovnu verziju kompilatora. Kao što je već pomenuto, zbog heuristike petlje, izvršivi fajlovi *GraalSP* optimizovanih programa veći su od izvršivih fajlova *XGBoost* optimizovanih programa. Veći izvršivi fajlovi dodatno produžavaju vreme kompilacije.

Kako je veličina izvršivih fajlova Vu-Larus optimizovanih programa značajno veća od veličine izvršivih fajlova programa optimizovanih osnovnom verzijom kompilatora, to upotreba statičkog profajlera Vua i Larusa značajno povećava i vreme kompilacije programa. Statički profajler Vua i Larusa ne sprovodi izdvajanje atributa niti izvršava predviđanje modela mašinskog učenja prilikom predviđanja profila, ali generisanje znatno većih izvršivih fajlova poništava te prednosti i produžava vreme kompilacije programa. Kao rezultat toga, profajler Vua i Larusa u značajno većoj meri povećava vreme kompilacije u poređenju sa statičkim profajlerom *GraalSP*.

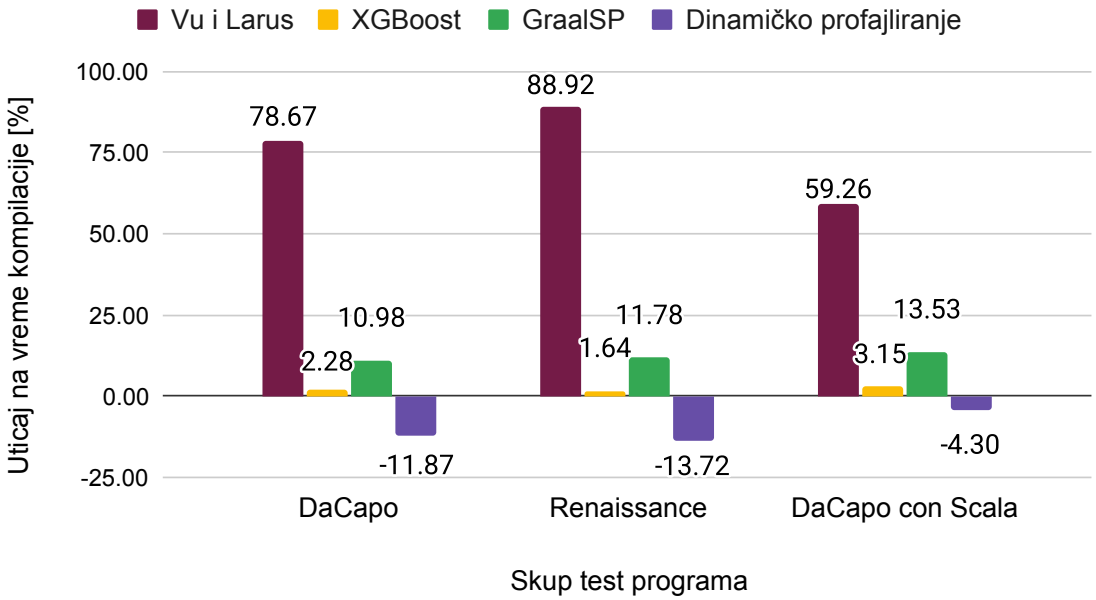
Upotreba dinamičkog profajliranja grana smanjuje ukupno vreme kompilacije programa za 10.98% u poređenju se kompilacijom u osnovnoj verziji kompilatora. Do toga dolazi jer kompilator *Enterprise GraalVM Native Image*, kada koristi dinamički prikupljene frekvencije izvršavanja grana, primenjuje vremenski zahtevne optimizacije samo na one delove programa



Slika 6.11: Uticaji profajlera na veličinu izvršivih fajlova optimizovanih programa

za koje postoji odgovarajući profil izvršavanja — odnosno na one delove koji će zaista biti izvršeni. Prilikom merenja vremena kompilacije za programe optimizovane na osnovu dinamički sakupljenih profila, ne uzima se u obzir vreme potrebno za instrumentaciju programa i prikupljanje profila. Kada bi se i to uključilo, vreme kompilacije korišćenjem dinamičkog profajlera bilo bi za red veličine duže u odnosu na vreme kompilacije kada se koristi neki od statičkih profajlera.

Na slici 6.12 prikazana je prosečna promena trajanja kompilacije programa prilikom korišćenja različitih profajlera. U tabeli 6.9 prikazana je geometrijska standardna devijacija agregiranja ovih rezultata. Uticaj upotrebe profajlera na vreme kompilacije programa korelisan je sa veličinom generisanih izvršivih fajlova.



Slika 6.12: Prosečan uticaj na vreme kompilacije optimizovanih programa

Tabela 6.9: Geometrijska standardna devijacija pri agregiranju rezultata kroz programe skupova programa za testiranje prilikom računanja uticaja profajlera na vreme kompilacije programa

	Vu i Larus	XGBoost	GraalSP	Dinamički profajler
DaCapo	1.07	1.03	1.05	1.06
Renaissance	1.08	1.05	1.04	1.06
DaCapo con Scala	1.11	1.08	1.04	1.07

Na slici 6.13 prikazan je uticaj ocenjivanih profajlera na vreme kompilacije pojedinačnih programa iz skupa programa korišćenih za evaluaciju modela. Vremena kompilacije Vu-Larus optimizovanih programa značajno su veća od vremena kompilacije programa optimizovanih osnovnom verzijom kompilatora. Uvećanje vremena kompilacije uglavnom je preko 70% i može ići čak do 122.20% kao što je slučaj sa vremenom kompilacije test programa *mnemonics* ili 112.31% kao što je slučaj sa vremenom kompilacije test programa *future-genetic* iz skupa test programa *Renaissance*.

Slika 6.13: Uticaji profajlera na vreme kompilacije optimizovanih programa

Vreme kompilacije *XGBoost* optimizovanih programa uglavnom je duže u odnosu na vreme kompilacija programa optimizovanih osnovnom verzijom kompilatora. Ipak, u pojedinim slučajevima ono može biti i kraće. Na primer, vreme kompilacije test programa *mnemonics* smanjeno je za 8.80%. Slično, kod programa *kiama* iz skupa test programa *DaCapo con Scala*, zabeleženo je smanjenje vremena kompilacije od 3.79%.

Vreme kompilacije *GraalSP* optimizovanih programa veće je od vremena kompilacije programa optimizovanih osnovnom verzijom kompilatora. U nekim slučajevima to povećanje iznosi svega 3.63%, kao što je slučaj sa test programom *scaladoc* iz skupa test programa *DaCapo con Scala*. Međutim, u drugim slučajevima, vreme kompilacije može biti znatno veće. Na primer, čak 21.01% na test programu *luindex* iz skupa test programa *DaCapo*.

Vreme kompilacije dinamički optimizovanih programa uglavnom je manje od vremena kompilacije porograma optimizovanih osnovnom verzijom kompilatora. Ušteda vremena kreće se do čak 21.21% i 20.61% na test programima *avrora* i *fj-kmeans*, prvi iz skupa test programa *DaCapo*, a drugi iz skupa test programa *DaCapo con Scala*.

6.7 Analiza predviđanja statičkog profajlera *GraalSP*

Najvažnija ocena kvaliteta predviđanja statičkog profajlera dobija se merenjem njegovog uticaja na efikasnost izvršavanja programa, veličinu izvršivih fajlova optimizovanih na osnovu predviđanja statičkog profajlera, kao i merenjem njegovog uticaja na vreme kompilacije programa. Pored toga, da bi se stekao bolju uvid u kvalitet predviđanja statičkog profajlera, potrebno je analizirati njegova predviđanja na konkretnim primerima.

Sortiranje pomoću hipa

U ovom delu analiziraju se predviđanja statičkog profajlera *GraalSP* na primeru programa za sortiranje pomoću hipa (listing 2, poglavlje 4). Pored ocene kvaliteta tih predviđanja kroz merenje njihovog uticaja na performanse izvršavanja programa i veličinu generisanog izvršivog fajla, izvedeno je i poređenje predviđenog profila sa profilom dobijenim dinamičkim profajliranjem, kao i semantička procena kvaliteta tih predviđanja (ljudska evaluacija, engl. *human evaluation*).

Predviđanje profila

Da bi se ocenio kvalitet predviđanja statičkog profajlera, najpre je potrebno pokrenuti taj statički profajler i zabeležiti predviđeni profil. Kako bi se predvidele verovatnoće izvršavanja grana naredbe grananja predstavljene čvorom *1. If* (slika 4.2, poglavlje 4), najpre je neophodno izvršiti konkatenaciju tri vektora atributa prikazana na slici 4.3. Nakon što se ti vektori spoje u jedan, sledi kodiranje složenih atributa, a zatim i preprocesiranje kroz odabir najinformativnijih atributa na osnovu varijanse. Na taj način izdvajaju se 78 najinformativnijih komponenti vektora atributa. Tako formiran vektor predstavlja ulaz u prethodno obučeni model mašinskog učenja *XGBoost*, koji služi za predviđanje verovatnoća izvršavanja grana u programu.

Naredni korak u procesu predviđanja profila jeste pokretanje modela i predviđanje verovatnoće izvršavanja prve grane naredbe grananja koja odgovara čvoru *1. If*. Model *XGBoost*, integrisan u statički profajler *GraalSP*, predviđa da je verovatnoća izvršavanja prve grane čvora *1. If* jednaka 0.62. Ova vrednost predstavlja verovatnoću izvršavanja tela `for` petlje. Verovatnoća izvršavanja druge grane, odnosno verovatnoća izlaska iz petlje, tada je implicitno određena i iznosi 0.38.

Kako predviđena verovatnoća izvršavanja tela petlje nije manja od 0.20, *GraalSP* ne primenjuje heuristiku petlje. Slično, kako ni jedna od grana čvora *1. If* ne vodi ka prekidu programa, *GraalSP* neće primeniti ni heuristiku zaustavljanja programa. Koristeći sličnu proceduru, *GraalSP* predviđa da je verovatnoća izvršavanja tela *while* petlje jednaka 0.94.

Uticaj na performanse programa

Precizno predviđanje verovatnoća izvršavanja grana od ključnog je značaja za agresivne optimizacije kompilatora, poput umetanja funkcija. Ukoliko statički profajler predvidi da će se tela `for` i *while* petlji često izvršavati, to će sugerisati kompilatoru da umetne pozive metode *pushDown*. Na taj način će se izbeći troškovi poziva metoda pri umetanju elemenata u hip i smanjiti ukupno vreme sortiranja. Ukoliko statički profajler predvidi niske vrednosti verovatnoća izvršavanja tela pomenutih petlji, to može sugerisati kompilatoru da ne umetne pozive metoda *pushDown*, obzirom da se u tom slučaju ne radi o frekventnoj putanji u programu. Na taj način će kompilator propustiti priliku za optimizaciju programa.

Na primer, u slučaju sortiranja nasumično generisanog niza od deset miliona celih brojeva, ukoliko kompilator umetne pozive metode *pushDown*, prosečno vreme sortiranja iznosi 1.80 sekundi[1]. U slučaju kada ti pozivi nisu umetnuti, prosečno vreme sortiranja iznosi 2.18 sekundi. Ovo je usporenje vremena sortiranja od 21.11%.

Predviđena verovatnoća izvršavanja tela `for` petlje korišćenjem statičkog profajlera *GraalSP* jeste 0.62. Kvalitet ovakvog predviđanja može se dovesti u pitanje, budući da veća verovatnoća izvršavanja odgovara frekvencijama izvršavanja petlje prilikom sortiranja većih nizova. U slučaju većih nizova i efikasnost sortiranja postaje znatno važnija. Na primer, za sve nizove duže od 20 elemenata, frekvencija izvršavanja prelazi 0.90, što implicira grešku u predviđanju od najmanje 0.28.

Međutim, predviđanje verovatnoće izvršavanja tela `for` petlje od 0.62 dovoljno je da dovede do umetanja poziva metode *pushDown* i time poboljša vreme izvršavanja sortiranja za više od 20%. Ovo poboljšanje je ostvareno u poređenju sa osnovnom verzijom kompilatora *Enterprise GraalVM Native Image*, koja koristi uniformnu raspodelu verovatnoća da modeluje verovatnoće izvršavanja grana naredbi grananja.

Pored značajnog pozitivnog uticaja na performanse izvršavanja optimizovanog programa, korišćenje statičkog profajlera *GraalSP* donosi minimalan negativan uticaj na veličinu optimizovanog programa i vreme kompilacije programa. Pri kompilaciji programa za sortiranje sa optimizacijama vođenim profilom kog predviđa statički profajler *GraalSP*, veličina generisanog izvršivog fajla povećava se za 3.12%, sa 6.4 MB na 6.6 MB. Takođe, vreme kompilacije povećava se za 16.44%, sa 22.5 sekundi na 26.2 sekunde.

Evaluacija korišćenjem dinamički sakupljenih profila

Na slici 6.14 prikazano je prvih 20 vrsta izveštaja koji alat *GraalSP-PLog* generiše pri pokretanju na primeru programa koji koristi metod sortiranja pomoću hipa (listing 2) za sortiranje niza od deset miliona nasumično generisanih celih brojeva. Na taj način alat *GraalSP-PLog* omogućava praćenje statički predviđenih verovatnoća izvršavanja grana i dinamički prikupljenih frekvencija izvršavanja tih grana. Važno je naglasiti da bi dinamičko profajliranje, u zavisnosti od veličine ulaznih nizova, prikupilo različite frekvencije izvršavanja grana naredbi grananja.

[1]Da bi se procenio stvarni uticaj na korisnike, merenja su izvršena korišćenjem kompilatora *Enterprise GraalVM Native Image* na razvojnom laptopu sa šestojezgarnim Intel i7 procesorom. Vreme sortiranja mereno je sa preciznošću u nanosekundama, a prosek izračunat na osnovu deset nezavisnih izvršavanja programa.

Declaring Class	Method Name	Node	NodeSource Position	Block	Block Frequency	Profiled Probability	Predicted Probability	Guarded Probability	Global Execution Frequency	Execution Count	Absolute Error	Weighted Absolute Error
SortExample	pushDown	14|If	13	B1	1.3333	0.9487	0.6316		0.232643	175836759	0.31711	0.073773
SortExample	pushDown	25|If	25	B2	0.6667	1	0.8873		0.220708	166816004	0.11274	0.024883
SortExample	pushDown	69|If	44	B9	0.3333	0.4916	0.5382		0.220708	166815980	0.04666	0.010298
java.nio.Buffer	position	29|If	37	B1	1	0	0.8747		0.003626	2740756	0.87465	0.003172
SortExample	heapSort	14|If	8	B1	2	0	0.3809		0.006615	5000002	0.38092	0.00252
SortExample	pushDown	113|If	61	B20	0.6667	0.9642	0.9539		0.220708	166816004	0.01027	0.002267
sun.nio.cs.StreamEncoder	implWrite	6|If	4	B0	1	1	0.0035		0.001376	1039907	0.99648	0.001371
java.util.Random	next	33|If	35	B4	2	0	0.0707	0.2	0.013231	10000000	0.07069	0.000935
SortExample	heapSort	50|If	32	B10	2	0	0.0642		0.013231	10000000	0.06424	0.00085
java.nio.charset.CharsetEncoder	encode	99|If	97	B17	0.2832	1	0.4283		0.001378	1041299	0.57172	0.000788
SortExample	generateRandomArray	25|If	21	B4	2	1	0.9632		0.013231	10000001	0.03684	0.000487
sun.nio.cs.StreamEncoder	implWrite	52|If	25	B8	1.3333	0	0.1848		0.001385	1046473	0.18482	0.000256
sun.nio.cs.UTF_8$Encoder	encodeArrayLoop	73|If	103	B7	0.9999	0.0063	0.1623		0.001397	1055831	0.156	0.000218
sun.nio.cs.StreamEncoder	implWrite	85|If	46	B13	0.6666	0.0064	0.1175	0.2	0.001385	1046495	0.11115	0.000154
jdk.internal.util.Preconditions	checkIndex	11|If	6	B2	1	1	0.9402		0.002197	1660455	0.0598	0.000131
java.nio.Buffer	limit	31|If	42	B1	1	0	0.1229		0.000979	740030	0.12294	0.00012
java.nio.charset.CharsetEncoder	encode	78|If	82	B13	1.1327	0.9936	0.9074		0.001386	1047910	0.08624	0.00012
sun.nio.cs.StreamEncoder	write	44|If	49	B14	1	0	0.2728		0.000403	304624	0.27276	0.00011
sun.nio.cs.StreamEncoder	implWrite	141|If	80	B35	0.3333	0	0.0522		0.001376	1039891	0.05218	7.18E-05
jdk.internal.util.DecimalDigits	stringSize	29|If	24	B5	0.6667	0.3113	0.3877		0.000801	605148	0.07636	6.11E-05
sun.nio.cs.UTF_8$Encoder	encodeLoop	38|If	11	B8	0.5	0	0.0285		0.001397	1055809	0.02847	3.98E-05

Slika 6.14: Izveštaj alata *GraalSP-PLog* generisan za analizu predviđanja statičkog profajlera *GraalSP* na primeru programa koji sortira deset miliona slučajno generisanih celih brojeva metodom sortiranja hipom. Vrste koje se odnose na naredbe grananja u metodi *pushDown* obojene su svetlo narandžastom bojom, dok su vrste koje se odnose na naredbe grananja u metodi *heapSort* obojene svetlo zelenom bojom.

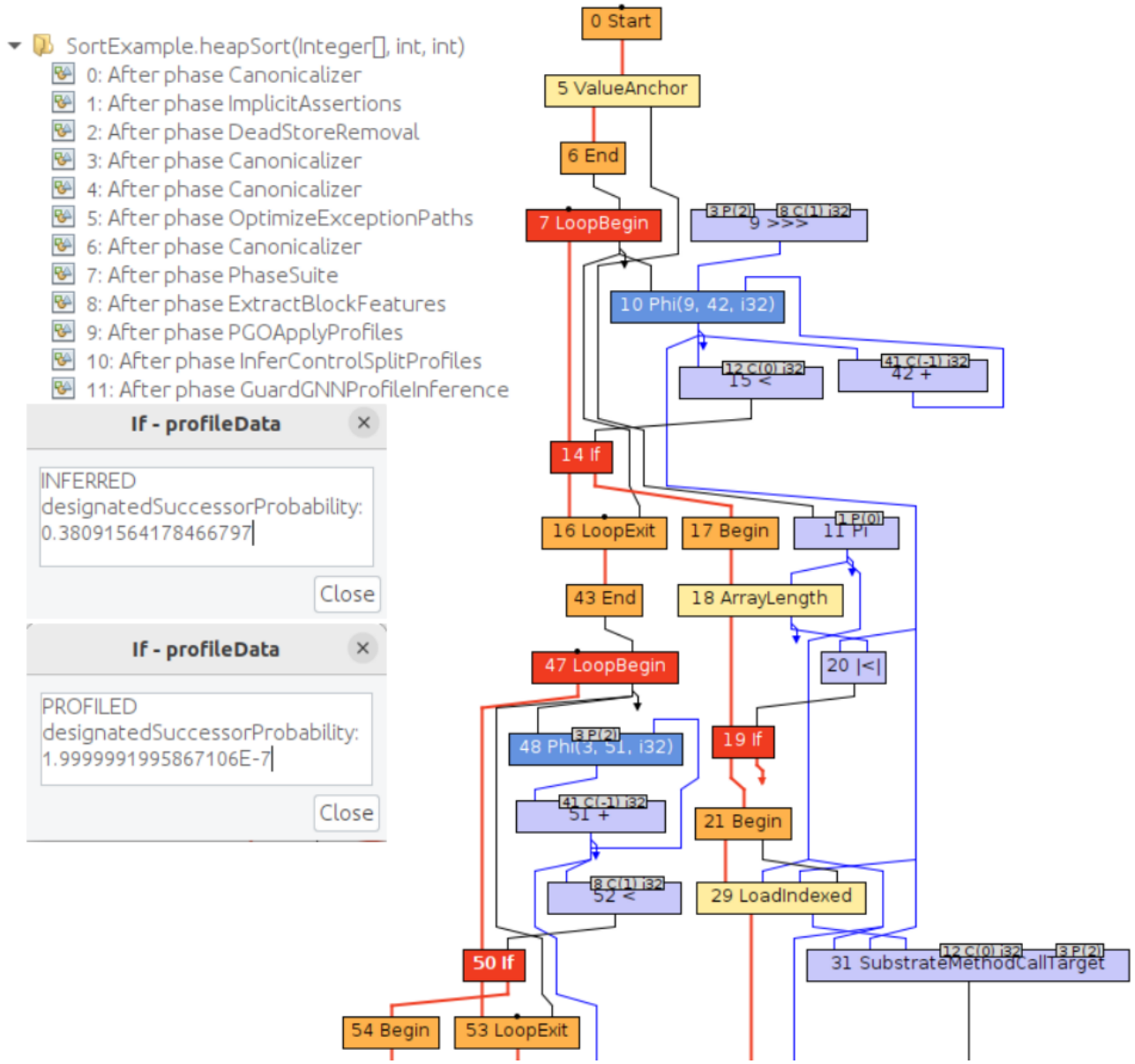
Normalizovane frekvencije izvršavanja naredbi grananja iz metoda *heapSort* i *pushDown* iznose ukupno 0.9146. Ovo ukazuje na to da performanse izvršavanja programa za sortiranje u velikoj meri zavise upravo od ovih dvaju funkcija.

Alat *GraalSP-PLog* sortira vrste u izveštaju po opadajućem redosledu prema vrednosti težinske apsolutne greške. S obzirom na to da se u test programu najviše vremena provodi u samom procesu sortiranja nizova, u prvih deset vrsta izveštaja pojavljuju se čak četiri *cfs*-čvora iz metode *pushDown* i dva *cfs*-čvora iz metode *heapSort*. Jedan od njih odgovara `for` petlji, a drugi *while* petlji. Alat *GraalSP* predviđa verovatnoće izvršavanja grana ovih naredbi grananja sa prosečnom apsolutnom greškom koja je jednaka 0.1553, kao i prosečnom srednjekvadratnom greškom koja je jednaka 0.2100.

Na slici 6.15 prikazane su i predviđena verovatnoća izvršavanja i prikupljena frekvencija izvršavanja za čvor koji odgovara naredbi grananja for petlje u metodi *heapSort*. Čvor označen kao *14. If* na toj slici odgovara čvoru *1. If* sa slike 4.2[2] Čvor označen kao *50. If* predstavlja proveru uslova za ulazak u petlju *while* na liniji broj 12 u listingu 2. Korisnici alata *GraalSP-PLog* mogu klikom miša na te čvorove videti verovatnoće izvršavanja prve grane u različitim fazama kompilacije.

Leva strana slike 6.15 prikazuje kako alat IGV predstavlja različite faze kompilacije. Tu su prikazane i: verovatnoća izlaska iz `for` petlje na osnovu statičkog predviđanja, kao i frekvencija izvršavanja te grane dobijena dinamičkim profajliranjem. Iste brojke nalaze se i u izveštaju sa slike 6.14.

[2]Na slici 4.2 koristi se pojednostavljena numeracija čvorova *Graal IR* grafa. Takođe, radi preglednosti i jasnije intuicije, kao prva grana prikazana je ona koja vodi ka telu petlje, dok druga grana vodi van petlje. Nasuprot tome, na *Graal IR* grafu sa slike 6.15, prva grana vodi van petlje. Redosled grana u okviru naredbe grananja definiše se nakon faze parsiranja Java bajtkoda i kreiranja *Graal IR* grafa i može da varira u zavisnosti od primenjenih optimizacija.



Slika 6.15: Deo *Graal IR* grafa koji odgovara početku `for` petlje metoda *heapSort*. Sa leve strane prikazan je padajuća lista u kome se mogu videti *Graal IR* grafovi metode u različitim fazama kompilacije, dok je u donjem levom uglu dat prikaz predviđene verovatnoće izvršavanja i dinamički prikupljene frekvencije izvršavanja prve grane čvora *14. If*. Kako prva grana ove naredbe grananja predstavlja izlazak iz petlje, *GraalSP* predviđa verovatnoću izvršavanja tela petlje jednaku *0.62*. Dinamički sakupljena frekvencija izvršavanja tela `for` jednaka je $0.9999998 = 1 - 1.9999991995867106 \cdot 10^{-7}$.

Procena eksperta

Ocena tačnosti predviđanja verovatnoća izvršavanja grana od strane ljudskih eksperata oslanja se na razumevanje semantike programa, kao i na pretpostavke o raspodeli ulaznih podataka. Drugim rečima, ekspert analizira logiku programa, uključujući kontrolne strukture kao što su naredbe grananja i petlje, bacanje i obradu izuzetaka, alokacije i kreiranje objekata i slično kako bi procenio koje se grane verovatnije izvršavaju pri tipičnim ulazima. Ovakva

procena često uključuje i uvid u namenu algoritma i karakteristike podataka koje se u praksi očekuju. Time se omogućava formulacija informisanih pretpostavki o ponašanju programa bez njegovog stvarnog izvršavanja.

Na primer, `for` petlja sa linije broj 9 (listing 2) smešta polovinu elemenata iz ulaznog niza a u hip. Dakle, telo ove petlje biće izvršeno $\frac{n}{2}$ puta, gde je n broj elemenata niza a. Osim toga, petlja se prekida samo jednom, kada su ovi elementi smešteni u hip. Ovo dovodi do frekvencije izvršavanja jednake $\frac{\frac{n}{2}}{\frac{n}{2}+1}$. U slučajevima kada se ova petlja izvršava[3], njena najmanja moguća frekvencija izvršavanja prikuplja se pri sortiranju niza koji se sastoji od samo dva elementa i iznosi 0.5. Frekvencija izvršavanja tela `for` petlje raste sa povećanjem broja elemenata niza a. Zbog toga se svako predviđanje verovatnoće izvršavanja tela `for` petlje koje je veće od 0.5 može smatrati ispravnim, dok se više vrednosti predviđene verovatnoće poklapaju sa frekvencijama izvršavanja kod sortiranja nizova veće dužine.

Primeri iz skupa programa za testiranje

U ovom delu analizirana su predviđanja statičkog profajlera *GraalSP* na konkretnim primerima programa iz skupa programa za testiranje. Kako bi se prikazale različite programske tehnike kao što su rekurzija, iteracija i koncepti objektno orijentisanog programiranja, odabrani su raznovrsni primeri programa kao i različiti programski jezici – Java i Skala.

Ispravna predviđanja

Na listingu 4 prikazana je `for` petlja unutar metode `indexDocs` u okviru test programa *luindex* iz skupa test programa *DaCapo*[4]. U ovoj petlji metoda indeksira dokumenate iz ulaznog direktorijuma. Petlja na liniji broj 4 predstavlja tipičnu strukturu petlje koja poziva metod uz obavljanje dodatnih zadataka kao što su kreiranje objekata, pristup elementima niza i slično.

GraalSP predviđa verovatnoću izvršavanja tela petlje od 0.9873, dok dinamički profajler prikuplja frekvenciju izvršavanja tela petlje od 0.9477. Predviđena verovatnoća izvršavanja tela petlje može se smatrati ispravnom, jer je veoma bliska dinamički sakupljenoj frekvenciji izvršavanja petlje. Dinamički sakupljen profil ukazuju na to da se telo petlje prilikom pokretanja programa izvršava veliki broj puta. Verovatnoća koju je predvideo statički profajler *GraalSP* takođe ukazuje na učestalo izvršavanje tela petlje. Zbog toga će optimizacija ovog dela programa zasnovana na statički predviđenom profilu biti identična optimizaciji zasnovanoj na dinamički prikupljenom profilu. Shodno tome, predviđanja statičkog profajlera će omogućiti poboljšanje performansi programa kroz efikasnije optimizovanje tela petlje u odnosu na osnovnu konfiguraciju kompilatora koja koristi uniformnu verovatnoću da modeluje verovatnoće izvršavanja grana naredbi grananja.

Greške u predviđanjima

U listingu 5 prikazana je `for` petlja iz metode `filter` biblioteke *Jenetics*, koju test program *feature-genetic* iz skupa test programa *Renaissance* koristi za filtriranje starih i nevažećih jedinki iz populacije[5]. *GraalSP* predviđa verovatnoću izvršavanja tela petlje od 0.7839, dok

[3]Petlja se izvršava za sve nizove koji imaju bar dva elementa. Niz koji se sastoji samo od jednog elementa je već sortiran i zato ova petlja neće ni biti izvršena.

[4]Izvorni kôd (napisan u programskom jeziku Java) test programa *luindex* javno je dostupan i može se pronaći na servisu GitHub [107].

[5]Izvorni kôd (napisan u programskom jeziku Skala) test programa *feature-genetic* javno je dostupan i može se pronaći na servisu GitHub [106].

```
1 if (files != null) {
2   System.out.print(file.getName()+" ("+files.length+") ");
3   Arrays.sort(files);
4   for (int i = 0; i < files.length; i++) {
5     indexDocs(writer, new File(file, files[I]), prefix);
6   }
7 }
```

Listing 4: Indeksiranje dokumenata u test programu *luindex* iz skupa test programa *DaCapo*

dinamički profajler sakuplja frekvenciju izvršavanja tela te petlje od 0.9615. Razlika predviđene verovatnoće izvršavanja tela petlje i dinamički sakupljene frekvencije izvršavanja tela petlje iznosi 0.1776. Greška u predviđanjima od 0.1776 može se tumačiti kao nezanemarljiva. Ipak, u slučaju programa *future-genetic* predviđena verovatnoća izvršavanja tela petlje od 0.7839 dovodi do izvođenja istih optimizacija programa kao i kada se optimizacije izvode na osnovu dinamički sakupljenih profila. Stoga se greška od 0.1776 u ovom konkretnom slučaju ne smatra značajnom.

GraalSP predviđa da je verovatnoća izvršavanja prve grane `if` naredbe grananja unutar `for` petlje (`if` naredba koja se nalazi na liniji broj 4) jednaka 0.6147. Za drugu `if` naredbu, koja se nalazi na liniji broj 7, predviđa da je verovatnoća izvršavanja prve grane jednaka 0.8284. Međutim, dinamički profajlirane frekvencije izvršavanja ovih grana su 0 jer prilikom prikupljanja profila algoritam nije filtrirao stare i nevažeće jedinke iz populacije (na osnovu ulaznog primera za prikupljanje profila). Ove dve naredbe grananja predstavljaju primere netačnih predviđana statičkog profajlera *GraalSP*. Ipak, da su ulazni podaci (ili parametri i ograničenja u test programu) bili drugačiji, metoda bi možda izvršavala navedene grane naredbi grananja, što bi rezultovalo manjim greškama u predviđanjima.

```
1 final MSeq<Phenotype<G, C>> pop = MSeq.of(population);
2 for (int i = 0, n = pop.size(); i < n; ++i) {
3   final Phenotype<G, C> individual = pop.get(i);
4   if (!_constraint.test(individual)) {
5     pop.set(i, _constraint.repair(individual, generation));
6     ++invalidCount;
7   } else if (individual.age(generation) > _evolutionParams.maximalPhenotypeAge) {
8     pop.set(i, Phenotype.of(_genotypeFactory.newInstance(), generation));
9     ++killCount;
10  }
11 }
```

Listing 5: Iteracija kroz jedinke populacije u biblioteci *Jenetics*

Uticaj heuristika za korekciju predviđanja modela mašinskog učenja

Kako bi se analizirao i bolje razumeo uticaj heuristika za korekciju predviđanja modela mašinskog učenja na rezultate statičkog profajlera *GraalSP*, alat *GraalSP-PLog* pokrenut je nad test programom *scala-kmeans* iz skupa test programa *Renaissance*. Program *scala-kmeans* izabran je zato što ima znatno kraće vreme izvršavanja u poređenju sa ostalim programima iz

skupa. Ovaj izbor je napravljen i jer alat *GraalSP-PLog* paralelno pokreće i dinamički profajler, koji izvršava vremenski i memorijski zahtevnu instrumentaciju i prikupljanje profila.

U ovom test programu ukupno je izvršeno 4030 naredbi grananja. Heuristike za korekciju predviđanja modela korigovale su predviđanja modela *XGBoost* na 70 instanci. Konkretno, heuristika petlje je korigovala predviđanja modela 28 puta, dok je heuristika zaustavljanja programa korigovala predviđanja modela 42 puta.

Iako su heuristike primenjene na manje od 2% instanci, njihov uticaj na performanse programa veoma je značajan. Na test programu *scala-kmeans*, heuristika petlje aktivirana je dva puta među deset najfrekventnijih naredbi grananja. U tim slučajevima, model *XGBoost* predvideo je verovatnoće izvršavanja grana koje vode u tela petlji kao 0.18 i 0.05, dok su dinamički prikupljene frekvencije izvršavanja tih grana iznosile 0.92 i 0.85. Heuristika za petlje korigovala je ove verovatnoće na 0.2, što je bilo dovoljno za optimizovanje ovih petlji i poboljšanje performansi programa.

6.8 Diskusija

Izvedeni eksperimenti se mogu reprodukovati jer su svi potrebni resursi javno dostupni, uključujući i skupove programa koji su iskorišćeni za kreiranje skupa podataka za obučavanje modela [222] dok je proces obučavanja modela detaljno opisan u ovoj disertaciji i u relevantnim naučnim radovima [68]. Prilikom evaluacije različitih profajlera korišćena je identična verzija kompilatora *Enterprise GraalVM Native Image*, čime je osigurano da jezgro kompilatora bude isto u svim eksperimentima. Samim tim, sve razlike u rezultatima potiču isključivo od različitih profila koji su primenjeni a na osnovu kojih su izvođene optimizacije programa.

Skup za obučavanje modela i skup za testiranje se ne preklapaju [222, 239, 23, 267], stoga je evaluacija izvršena na nezavisnom skupu podataka. Zbog toga se prijavljeni rezultati na test skupu mogu smatrati relevantnim.

Merenja vremena kompilacije programa vršena su na laptopu korišćenom za razvoj statičkog profajlera, kako bi se procenio uticaj koji krajnji korisnici statičkog profajlera mogu osetiti. Na taj način dobijeni su rezultati koji su reprezentativni za sve korisnike koji će kompilirati programe pomoću kompilatora *Enterprise GraalVM Native Image* na ličnim računarima.

Glava 7

Poređenje sa relevantnim statičkim profajlerima

U ovom poglavlju uporedićemo statički profajler *GraalSP* sa najnaprednijim postojećim rešenjima. Najbolji do sada razvijeni statički profajleri su profajler Vua i Larusa [326], zasnovan na statičkim heuristikama za predviđanje profila, kao i tri profajlera zasnovana na modelima mašinskog učenja: profajler Rotema i Kuminsa [254], profajler Ramana i Lija [242] i profajler *VESPA* [202].

7.1 Kvantitativno poređenje

Statički profajleri razvijani su za različite platforme i programske jezike, te se shodno tome i ocenjuju pomoću različitih referentnih testova (engl. *benchmarks*). Zbog toga se rezultati prikazani u relevantnim radovima ne mogu direktno upoređivati, već je prilikom poređenja neophodno uzeti u obzir sve navedene razlike.

Tabela 7.1 prikazuje uticaj razmatranih statičkih profajlera na ubrzavanje programa, veličinu optimizovanih programa i vreme potrebno za generisanje optimizovanih programa. Pored toga, u tabeli je prikazan i broj referentnih test programa korišćenih prilikom evaluacije statičkih profajlera kao i najbolji i najlošiji rezultat koji su statički profajleri ostvarili na nekom od test programa. Dodatno, u tabeli je naznačen i tip agregacije rezultata, odnosno da li su prikazani podaci dobijeni korišćenjem aritmetičke ili geometrijske sredine pojedinačnih rezultata test programa.

U tabeli 7.1 prikazane su vrednosti prijavljene u referentnim radovima [326, 254, 242, 202] i rezultati dobijeni ocenjivanjem statičkog profajlera *GraalSP* i ocenjivanjem implementiranog profajlera Vua i Larusa (Dodatak A) za kompilator *Enterprise GraalVM Native Image*. Vrednosti u tabeli izražene su u procentima u odnosu na osnovnu (engl. *baseline*) konfiguraciju kompilatora. Osnovna konfiguracija označava podrazumevana podešavanja kompilatora, bez korišćenja statičkog profajliranja.

Uticaj statičkog profajlera na performanse optimizovanih programa u velikoj meri zavisi od referentnih testova koji se koriste tokom evaluacije. Što je veći broj različitih programa na kojima se vrši testiranje, to je izraženiji raspon između najlošijeg i najboljeg uticaja profajlera na performanse optimizovanih programa. To je posledica činjenice da performanse izvršavanja – poput vremena izvršavanja programa – zavise ne samo od tačnosti predviđanja statičkog profajlera na frekventnim petljama, već i od optimizacija koje ta predviđanja omogućavaju.

Tabela 7.1: Kvantitativna analiza relevantnih statičkih profajlera. Vrednosti u tabeli (osim u prve dve kolone) su izražene u procentima u poređenju sa osnovnom konfiguracijom. Prazne ćelije odgovaraju vrednostima koje nisu prijavljene.

		Vu i Larus [326]				
	BOLT [228, 202]	Enterprise GraalVM Native Image	Rotem i Kumins [254]	Raman i Li [242]	VESPA [202]	GraalSP
Broj test programa	4	28	10	40	4	28
Agregacija	Aritm. sred.	Geom. sred.	Geom. sred.	Geom. sred.	Aritm. sred.	Geom. sred.
Ubrzanje prog. (%)	2.77	5.64	1.60	1.00	5.47	7.46
Najmanje, Najveće	1.11, 4.42	−49.30, 20.11	−7.00, 16.00	−6.50[1], 8.10	2.28, 8.84	−0.88, 28.05
Povećanje veličine prog. (%)		31.60				3.91
Najmanje, Najveće		12.52, 47.93				1.46, 6.44
Povećanje vremena generisanja prog. (%)	178.72	78.51			1296.07	12.01
Najmanje, Najveće	138.56, 213.24	31.70, 122.20			1089.13, 1629.82	3.37, 21.01

Pored toga, aplikacije se razlikuju i po svojoj prirodi – neke su veoma osetljive na tačnost predviđanja, dok druge pokazuju malu ili nikakvu zavisnost od njih.

Relevantni radovi ne razmatraju uticaj statičkih profajlera na veličinu optimizovanih programa, iako je ova metrika od velikog značaja. Na primer, u kontekstu aplikacija koje se izvršavaju u oblaku (engl. *cloud applications*), gde se izvršivi fajlovi skladište na serverima, a korisnici plaćaju prema količini zauzetog prostora i protoka podataka, manja veličina izvršivih fajlova može direktno doprineti smanjenju troškova i poboljšanju efikasnosti sistema.

Kada je reč o vremenu potrebnom za generisanje optimizovanih programa, *GraalSP* postiže za red veličine bolje rezultate u poređenju sa relevantnim statičkim profajlerima. Moreira[2] i ostali [202] navode samo vremena potrebna za generisanje izvršivih fajlova, ne i odgovarajuće vrednosti izražene u procentima, koje bi omogućile jasnije poređenje statičkih progajlera. U tabeli 7.1 prikazan je odnos vremena potrebnog za generisanje izvršivih fajlova kada optimizator *BOLT* koristi statički profajler *VESPA* za predviđanje profila, u poređenju sa vremenom generisanja izvršifih fajlova u podrazumevanoj konfiguraciji optimizatora. Na osnovu tih vrednosti izračunali smo prosečan odnos i prikazali ga u procentima.

7.2 Kvalitativno poređenje

Tabela 7.2 prikazuje kvalitativno poređenje statičkih profajlera, koje obuhvata poređenje atributa kojima se opisuju naredbe grananja, korišćenih modela i tehnika mašinskog učenja, načina obučavanja modela, korišćenih heuristika za statičko predviđanje profila kao i platformi na kojima su profajleri razvijani.

Skup atributa. Statički profajler *GraalSP* izdvaja atribute iz grafovske međureprezentacije kompilatora. Ni jedan drugi relevantan statički profajler ne operiše nad internom reprezentacijom zasnovanom na grafu. Vu i Larus [326] razvijaju heuristike za predviđanje verovatnoća izvršavanja grana naredbi grananja na osnovu binarne reprezentacije programa (engl. *program binary*) odnosno na nivou izvršivih fajlova. Slično, Moreira i ostali [202] izdvajaju atribute iz binarne reprezentacije programa i integrišu statički profajler *VESPA* u binarni optimizator *BOLT* [228]. Rotem i Kumins [254] i Raman i Li [242] izdvajaju atribute iz LLVM interne reprezentacije.

[2]Angélica Aparecida Moreira

Tabela 7.2: Glavne karakteristike statičkog profajlera *GraalSP* i relevantnih statičkih profajlera

	Vu i Larus [326]	Rotem i Kumins [254]	Raman i Li [242]	VESPA [202]	*GraalSP*
Atributi	Binarna reprez.	LLVM IR	LLVM IR	Binarna reprez.	*Graal IR*
Koristi mašinsko učenje	Ne	Da	Da	Da	Da
Model		XGBoost	DNN	DNN	XGBoost
Tehnika		Višeklasna klasifikacija	Regresija	Regresija	Regresija
Obučavanje		Ignorisanje retko izvršenih naredbi		Normalizovane težine instanci	Normalizovane težine instanci
Heuristike (Broj)	Da (9)	Ne	Ne	Ne	Da (2)
Platforma	DYNIX/ptx	LLVM [167]	Clang [165]	*BOLT* [228]	GraalVM [320]

Tok podataka unutar programa nosi veoma važne informacije za uspešno izvođenje optimizacija [29]. Iako se informacije o toku podataka u programu mogu dobiti i u drugim međureprezentacijama, u *Graal IR* pokretni čvorovi koji predstavljaju tok podataka dostupni su podrazumevano i njihovo korišćenje ne zahteva dodatni trud. Analiza značaja atributa sa slike 6.1 potvrđuje važnost informacija o toku podataka u programu i pokazuje da su tri od pet najznačajnijih atributa korišćenih za obučavanje modela mašinskog učenja za potrebe razvoja statičkog progajlera *GraalSP* upravo pokretni čvorovi iz atributa `FNodeCountMap`.

Graal IR je interna međureprezentacija visokog nivoa kompilatora *GraalVM Native Image* koja se generiše odmah nakon parsiranja bajtkoda. Predviđanjem profila na osnovu Graal IR reprezentacije i to odmah nakon parsiranja bajtkoda omogućava se većem broju faza i optimizacija da iskoriste predviđeni profil. Izazov pri definisanju statičkog profajlera nad reprezentacijom *Graal IR* leži u odsustvu karakteristika programa niskog nivoa poput veličine asemblerskog koda i broja CPU ciklusa potrebnog za izvršavanje delova programa. Ovakve informacije podrazumevano su dostupne relevantnim statičkim profajlerima dizajniranim nad binarnim reprezentacijama programa [326, 202] ili nad internim međureprezentacijama programa niskog nivoa [254]. Prilikom razvoja statičkog profajlera *GraalSP* problem nedostatka informacijama o karakteristikama programa niskog nivoa prevazilazi se definisanjem atributa visokog nivoa koji aproksimiraju osobine delova programa niskog nivoa. Ovo se odnosi na atribute 13.–16. iz tabele 4.1.

Model mašinskog učenja. Kalder i ostali [37] obučavali su model stabla odlučivanja za predviđanje izvršavanja grana naredbi grananja. Kako bi izbegli preprilagođavanje modela, primenili su tehniku odsecanje stabla sa minimalnom složenošću (engl. *minimal-complexity pruning*). Prilikom razvoja statičkog profajlera *GraalSP* korišćen je i model stabla odlučivanja, pri čemu je primenjivan isti pristup odsecanja stabla. Sa druge strane, prilikom obučavanja modela *XGBoost*, ograničavana je dubina svakog stabla unutar ansambla jer je ograničavanje dubine efikasnije od naknadnog odsecanja stabla (engl. *post-pruning*). Ovo je posebno važno jer *XGBoost* ansambl u statičkom profajleru *GraalSP* sadrži 1500 stabala. Ograničavanjem dubine stabala postižu se rezultati uporedivi sa onima dobijenim naknadnim odsecanjima stabala.

Moreira i ostali [202] navode da njihovi pokušaji obučavanja ansambla stabala odlučivanja dovode do prekomerne potrošnje radne memorije. Rotem i Kumins [254] koriste model *XGBoost* za višeklasnu klasifikaciju, dok *GraalSP* koristi *XGBoost* model za regresiju. Na taj način *GraalSP* preciznije predviđa profil i izbegava gubitak informacija koji nastaje prilikom zaokruživanja i klasifikovanja verovatnoća u diskretne kategorije. S druge strane, statički profajler

Ramana i Lija [242] i statički profajler *VESPA* koriste regresioni model duboke neuronske mreže za predviđanje verovatnoća izvršavanja grana naredbi grananja.

Statički profajler *GraalSP* koristi model *XGBoost* mašinskog učenja jer na taj način postiže najbolje performanse optimizovanih programa, uz manju potrošnju resursa tokom kompilacije, u poređenju sa modelom duboke neuronske mreže. Pored toga, veličina obučenog modela, vreme predviđanja i vreme obučavanja su značajno manji u poređenju sa modelom duboke neuronske mreže. Iako postoji potencijal za dodatna poboljšanja modela duboke neuronske mreže, njegova veličina, vreme predviđanja i vreme obučavanja teško je značajno smanjiti, te to ostaju ključne prepreke za njihovu efikasnu primenu u produkcionom okruženju.

Obučavanje modela mašinskog učenja. Relevantni statički profajleri i statički prediktori grana naredbi grananja koriste različite pristupe za definisanje težina instanci prilikom obučavanja modela mašinskog učenja. Kalder i ostali [37] definišu *normalizovane težine naredbi grananja* (engl. *normalized branch weight*) koje se računaju tako što se broj izvršavanja naredbe grananja podeli sa ukupnim brojem izvršavanja svih naredbi grananja u programu. Time je normalizovana težina grane definisana kao realan broj između nula i jedan. Kako bi fokusirali modele mašinskog učenja na frekventnije naredbe grananja, Kalderi ostali dupliraju naredbe grananja proporcionalno njihovim normalizovanim težinama. Desmet i ostali [81] dodaju atribut koji beleži frekvenciju izvršavanja naredbi grananja. Rotem i Kumins [254] ignorišu naredbe grananja koje su se izvršile mali broj puta. Statički profajleri *GraalSP* i *VESPA* [202] koriste pristup predložen u radu Kaldera i ostalih i koriste normalizovane frekvencije izvršavanja naredbi grananja kao težine instanci kako bi fokusirali modele mašinskog učenja na frekventnije naredbe grananja. Raman i Li [242] ne koriste težine instanci prilikom obučavanja modela.

Heuristike. Vu i Larus [326] koriste devet statičkih heuristika za predviđanje verovatnoća izvršavanja grana. S druge strane, statički profajler *GraalSP* koristi dve statičke heuristike. Važna razlika je to što statički profajler *GraalSP* ne koristi heuristike za predviđanje profila, već za korekcije predviđanja modela mašinskog učenja. Ostali relevantni statički profajleri ne koriste heuristike.

Platforma. Statički profajler *GraalSP* integrisan je u kompilator *Enterprise GraalVM Native Image* koji je deo platforme *GraalVM*. Vu i Larus [326] razvili su statički profajler za kompilator za programski jezik C, *Sequent DYNIX/ptx*, verziju 2.1. Većinu heuristika autori su prilagodili iz rada Bala i Larusa [13]. Heuristike iz rada Bala i Larusa integrisane su u kompilator *Clang* [202, 165] i podrazumevano uključene. Rotem i Kumins [254] svoj statički profajler razvijali koristeći kompilatorsku infrastrukturu LLVM, dok su Raman i Li [242] integrisali razvijeni statički profajler u kompilator Clang [165]. Statički profajler *VESPA* [202] razvijen je za potrebe binarnog optimizatora *BOLT* [228].

Glava 8

Zaključci i pravci daljeg razvoja

Statički profajler *GraalSP* je višejezičan, efikasan i robustan statički profajler. Višejezičnost se ogleda u tome što podržava predviđanje profila u svim programima koji su napisani u jezicima koji se prevode na Java bajtkod. Efikasnost se postiže korišćenjem *XGBoost* modela mašinskog učenja zasnovanog na stablima odlučivanja, dok robusnost obezbeđuju heuristike koje koriguju predviđanja modela mašinskog učenja i omogućavaju stabilne performanse čak i u prisustvu odudarajućih podataka.

8.1 Glavni doprinosi disertacije

Za potrebe statičkog profajlera *GraalSP* razvijen je novi skup atributa za opisivanje naredbi grananja, zasnovan na internoj grafovskoj reprezentaciji visokog nivoa koju koristi kompilator *GraalVM Native Image*. Pored standardnih atributa za opis kontrole toka programa, preuzetih iz relevantne literature, prilagođeni su i atributi koji se zasnivaju na brojanju različitih instrukcija i različitih čvorova unutar blokova grafa kontrole toka programa. Zbog toga je proces definisanja atributa kojima se opisuju grane naredbi grananja delimično automatizovan, što poboljšava proces definisanja atributa i čini ga manje zavisnim od ljudskog faktora. Uz to, uvedeni su i novi atributi koji, koristeći internu reprezentaciju programa na visokom nivou aproksimiraju karakteristike odgovarajućeg programa niskog nivoa. U ove atribute se ubrajaju, na primer, broj ciklusa procesora potrebnih za izvršavanje dela programa i veličina generisanog asemblerskog koda.

Kreiran je obiman skup podataka za obučavanje modela mašinskog učenja, instrumentacijom četrdeset četiri moderne Java aplikacije koje su u aktivnoj upotrebi. Sprovedena je detaljna analiza skupa podataka koja je pokazala da su težine instanci koje se računaju na osnovu broja izvršavanja naredbi grananja od izuzetne važnosti za obučavanje kvalitetnih modela. Kako je raspodela ovako izračunatih težina izrazito pozitivno asimetrična, njihovo korišćenje omogućava obučavanje manjih modela sa smanjenim vremenom predviđanja, a koji su fokusirani na često izvršavane naredbe grananja, što rezultira boljim performansama optimizovanih programa.

Prilikom razvoja statičkog profajlera *GraalSP* za statičko predviđanje verovatnoća izvršavanja grana odabran je model *XGBoost*. Eksperimentisano je sa različitim modelima, ali je model *XGBoost* izabran jer se pokazao kao najprecizniji i najefikasniji. Optimizacije vođene profilom predviđenim ovim modelom rezultuju programima bržim za u proseku 5.22% u poređenju sa programima optimizovanim bez korišćenja statičkog profajlera. Efikasnost *XGboost* modela ogleda se u vremenu predviđanja koje je manje od 0.5 milisekundi. Pored toga, model *XGBoost* zauzima svega 290 KB, što značajno olakšava njegovu praktičnu upotrebu i isporučivanje

krajnjim korisnicima.

Razvijene su heuristike za korekciju predviđanja modela mašinskog učenja, koje doprinose većoj robusnosti statičkog profajlera *GraalSP*. Time se obezbeđuju stabilne performanse programa optimizovanih na osnovu predviđanja statičkog profajlera *GraalSP*, čak i u slučaju odudarajućih podataka.

Statički profajler *GraalSP* razvijen je kao robustno i jezički nezavisno rešenje koje može da se primeni na bilo koji programski jezik koji se kompilira u Java bajtkod. Njegovom integracijom u kompilator *Enterprise GraalVM Native Image* kreirano je kompletno rešenje (engl. *end-to-end solution*), koje obuhvata infrastrukturu za prikupljanje podataka, izdvajanje atributa za opisivanje naredbi grananja, obučavanje modela i njegovu primenu. Na ovaj način postignuto je poboljšanje performansi pri izvršavanju aplikacija koje se kompiliraju pomoću kompilatora *Enterprise GraalVM Native Image*.

Statički profajler *GraalSP* ostvaruje prosečno ubrzanje aplikacija od 7.46%, uz minimalno povećanje veličine izvršivih fajlova od 3.91% i vremena kompilacije od 12.01%. Ostvareno ubrzanje aplikacija od 7.46% je za 2% veće u poređenju sa dosad najefikasnijim statičkim profajlerom *VESPA*, koji dostiže ubrzanje od 5.47% kada se koristi u okviru binarnog optimizatora *BOLT*.

Za potrebe statičkog profajlera *GraalSP* razvijen je alat *GraalSP-PLog* za analizu i otkrivanje grešaka u predviđanjima verovatnoća izvršavanja grana. Alat je integrisan u kompilator *Enterprise GraalVM Native Image* i tokom kompilacije automatski pokreće statički i dinamički profajler, beleži predviđanja modela, korekcije heuristika i dinamički prikupljene frekvencije izvršavanja grana. Pored toga, alat *GraalSP-PLog* generiše izveštaj za korisnika i pokreće alat *Ideal Graph Visualizer*, koji omogućava prikaz predviđenih verovatnoća direktno na *Graal IR* grafovima metoda. Na taj način, alat *GraalSP-PLog* olakšava evaluaciju i procenu kvaliteta predviđanja statičkog profajlera *GraalSP*, kao i identifikaciju grešaka, čime podržava njegovo dalje unapređenje i razvoj.

8.2 Objavljeni rezultati

Statički profajler *GraalSP* integrisan je u kompilator *Enterprise GraalVM Native Image*, verziju 23.0, objavljenu u junu 2023. godine. Pri tome, statički profajler *GraalSP* podrazumevano je uključen, te se pri standardnoj kompilaciji pomoću kompilatora *Enterprise GraalVM Native Image* automatski pokreće radi predviđanja verovatnoća izvršavanja grana i usmeravanja optimizacija vođenih profilom. Detaljan opis i evaluacija profajlera *GraalSP* objavljeni su u julu 2024. godine [68]. Takođe, *GraalSP* je prijavljen kao patent u Sjedinjenim Američkim Državama pod brojem *ORC24137793-US-NPR* [351]. Alat *GraalSP-PLog* predstavljen je 2024. godine na konferenciji Veštačka inteligencija Srpske akademije nauka i umetnosti (engl. *SANU Artificial intelligence conference*) [67].

Nakon razvoja statičkog profajlera *GraalSP* nastavljen je rad na razvoju statičkih profajlera razvojem statičkog profajlera zasnovanog na grafovskim neuronskim mrežama. Grafovske neuronske mreže, kao izražajniji model u odnosu na modele zasnovane na stablima odlučivanja, sposobne su da preciznije predviđaju profile izvršavanja programa, što direktno doprinosi boljim performansama optimizovanih programa. Veća izražajna moć grafovskih neuronskih mreža takođe omogućava modelima da obrate pažnju i na instance sa manjim vrednostima težina, što doprinosi smanjenju veličine izvršivih fajlova. Na Međunarodnom simpozijumu o generisanju i optimizaciji koda (engl. *The International Symposium on Code Generation and Optimization, CGO*) održanom u martu 2025. godine u Sjedinjenim Američkim Državama, predstavljen

je statički profajler *GraalNN* [199]. U tu svrhu dizajnirana je i obučena grafovska neuronska mreža za predviđanje verovatnoća izvršavanja grana.

Statički profajler *GraalNN* implementira kontekstno osetljivo predviđanje verovatnoća izvršavanja grana naredbi grananja. U tom slučaju, atributi koji opisuju grane ne izdvajaju se samo iz grafa pozvane metode, već iz proširenog grafa koji uključuje i grafove metoda koje pozivaju datu metodu. Statićki profajler *GraalNN*, pored toga što koristi grafovsku neuronsku mrežu za predviđanje verovatnoća izvršavanja grana, koristi i kontekstno osetljivo predviđanje profila, profinjujući kontekstno neosetljiv profil i donoseći dodatnih od 2.5% do 3.7% performansi programa, u poređenju sa statičkim profajlerom *GraalSP*. Pored toga, statički profajler *GraalNN* doprinosi smanjenju veličine izvršivih fajlova od 1% geometrijske sredine, takođe poređeno sa statičkim profajlerom *GraalSP*. Statički profajler *GraalNN* prijavljen je kao patent u Sjedinjenim Američkim Državama pod brojem *ORC25139309-US-NPR* [140].

Kao deo razvoja i proširenja domena statičkog profajlera *GraalSP*, 2023. godine kreiran je skup podataka za predviđanje verovatnoća izvršavanja virtuelnih metoda u objektno-orijentisanim jezicima i izvršena je analiza kreiranog skupa podataka [70]. Kao deo bočnih istraživanja sprovođenih tokom izrade ove disertacije, ispitivana su svojstva grafova kontrole toka programa [249, 349]. Dodatno, kreiran je i skup od 200 000 grafova kontrole toka modernih Java aplikacija, a zatim su obučavani modeli mašinskog učenja za izbor najefikasnijeg načina njihovog obilaska [350]. Takođe, izvršen je opsežan pregled i sistematizacija moderne literature na temu dinamičkog i statičkog profajliranja u kontekstu kompilatorskih optimizacija [69]. Pored toga, u okviru bočnih istraživanja realizovani su i različiti projekti na kompilatoru *GraalVM Native Image*, uključujući ispitivanje i unapređenje sistema za evaluaciju performansi kompilatora [279].

8.3 Pravci daljeg razvoja

Postoji više različitih mogućnosti za dalje istraživanje u oblasti statičkog predviđanja profila izvršavanja programa u kontekstu statičkog profajlera *GraalSP*. One uključuju poboljšanja skupa podataka za obučavanje modela, poboljšanja modela mašinskog učenja koji se koristi za predviđanje verovatnoća izvršavanja grana naredbi grananja i proširivanje domena samog statičkog profajlera.

Jedan od najvažnijih pravaca daljeg razvoja jeste proširivanje skupa podataka za obučavanje modela programima pisanim u drugim jezicima koji se izvršavaju na Java virtuelnoj mašini, poput Skale i Kotlina. Programi napisani u ovim jezicima po svojoj prirodi se znatno razlikuju od onih napisanih u Javi. Zbog toga i ne iznenađuju nešto slabije performanse optimizovanih programa pisanih u ovim jezicima kada se za predviđanje profila izvršavanja koriste statički profajleri zasnovani na modelima mašinskog učenja obučavanim isključivo na Java programima. Na taj način se donekle odstupa od osnovne pretpostavke mašinskog učenja, a to je da podaci korišćeni za obučavanje modela moraju verno da odražavaju skup podataka nad kojim će se model primenjivati. Proširivanjem skupa za obučavanje modela poboljšaće se tačnost statičkog profajlera na programima koji su pisani u jezicima drugačijim od Jave. Pored toga, važan pravac budućih istraživanja jeste i razvoj modela specijalizovanih za pojedinačne jezike koji se izvršavaju na Java virtuelnoj mašini.

Ručno definisanje atributa za opis delova programa predstavlja složen i zahtevan posao koji se teško može precizno obaviti jer u velikoj meri zavisi od intuicije i iskustva inženjera koji te atribute definišu. Jedan od pravaca daljeg razvoja ovog projekta je automatizacija procesa definisanja atributa za statičko opisivanje programa. S obzirom na to da kompilator *GraalVM Native Image* koristi internu grafovsku reprezentaciju, potpunu automatizaciju procesa defi-

nisanja atributa moguće je uraditi korišćenjem grafovskih neuronskih mreža, koje po svom dizajnu operišu nad grafovima. Kako su se već pokazale uspešnim u razvoju statičkih profajlera, grafovske neuronske mreže mogu se iskoristiti za u potpunosti autonomno predviđanje verovatnoća izvršavanja grana, na osnovu atrubuta predstavljenih rečnicima *Graal IR* čvorova unutar blokova grafova kontrole toka programa.

Još jedan važan pravac daljeg razvoja predstavlja proširivanje domena statičkog profajlera *GraalSP* tako da, osim predviđanja verovatnoća izvršavanja grana naredbi grananja, može da predviđa i druge tipove profila. Na osnovu rezultata statičke analize moguće je kreirati graf poziva metoda u programu (engl. *call graph*), na osnovu kog bi bilo moguće predviđanje verovatnoća poziva metoda u programu. Ovaj pristup omogućava efikasnije korišćenje budžeta za optimizaciju, što bi rezultovalo bržim i manjim programima. Slično tome, *GraalSP* može biti proširen kako bi predviđao virtuelne pozive, čime bi se omogućio razvoj u punom smislu kontekstno osetljivog statičkog profajlera koji bi predviđao iste tipove profila kao i dinamički profajler.

Kao jedan od značajnih pravaca daljeg razvoja izdvaja se mogućnost generalizacije opisanih tehnika statičkog profajliranja na heterogene računarske sisteme. U slučaju heterogenog hardvera, gde se aplikacije izvršavaju na različitim vrstama procesorskih jedinica poput *CPU*, *GPU*, *TPU* ili *FPGA* čipova, statički profajleri morali bi da u obzir uzimaju specifične karakteristike hardvera. To podrazumeva proširivanje skupa atributa novim opisima karakteristika hardvera i obučavanje posebnih modela mašinskog učenja za različite vrste hardverskih platformi. Takođe, prilagođavanje statičkog profajlera heterogenom hardverskom sistemu uključuje i razvoj dodatnih heuristika koje bi omogućile korekciju i prilagođavanje predviđanja u skladu sa konkretnim osobinama ciljnog hardvera.

U heterogenom softverskom okruženju, gde komponente sistema obuhvataju tradicionalni aplikativni kôd, ali i savremene softverske module poput modela mašinskog učenja, kontrola toka u programu može biti manje važna ili manje predvidljiva u zavisnosti od komponente sistema. Kako bi se statički profajler uspešno prilagodio takvom okruženju potrebno je obučavanje različitih modela mašinskog učenja za specifične tipove softvera, ili razvoj jednog univerzalnog modela sposobnog da na osnovu proširenog skupa atributa razlikuje i adekvatno modeluje ponašanje različitih komponenti sistema. Na taj način otvara se prostor za razvoj robusnijih i fleksibilnijih alata za statičko profajliranje, prilagođenih zahtevima savremenih, sve složenijih softversko-hardverskih sistema.

Svi predstavljeni pravci poboljšanja i proširenja statičkog profajlera *GraalSP* doveli bi do značajnih naučnih rezultata i razvoja kvalitetnih praktičnih rešenja. Na predstavljenim pravcima poboljšanja i unapređenja statičkog profajlera *GraalSP* aktivno se radi. Na primer, u vreme pisanje ove disertacije završava se razvoj prve verzije statičkog profajlera za predviđanje izvršavanja metoda. U planu je da on bude integrisan u kompilator *Enterprise GraalVM Native Image* i to verziju 25.0, koja će biti objavljena u septembru 2025. godine.

Dodatak A

Implementacija heuristika Vua i Larusa

U tabeli 3.1 (sekcija 3.2) prikazane su heuristike Vua i Larusa [326] za predviđanje verovatnoća izvršavanja grana u programu. Detaljan opis ovih heuristika dat je u sekcijama 3.1 i 3.2, kao i u relevantnoj literaturi [326, 13]. U ovom dodatku biće opisan način na koji su navedene heuristike prilagođene i implementirane u kompilatoru *Enterprise GraalVM Native Image*.

Heuristika petlje Prilikom parsiranja Java bajtkoda kompilator *Enterpise GraalVM Native Image* uslov ulaska u petlju preslikava u čvor `If` u *Graal IR* grafu čija jedna grana vodi ka telu petlje, a druga ka izlasku iz nje. Heuristika petlje implementirana je tako da u ovakvim naredbama grananja dodeljuje verovatnoću izvršavanja od 0.88 grani koja vodi ka telu petlje.

Heuristika izlaska iz programa Implementacija ove heuristike oslanja se na vrednost atributa `NodeCountMap`. Ukoliko je za neku od grana izdvojen rečnik atributa `NodeCountMap` koji sadrži čvorove tipa `Return` ili `FarReturn`, za tu granu smatra se da je grana koja vodi izlasku iz petlje i dodeljuje joj se verovatnoća izvršavanja od 0.28.

Heuristika čuvanja u memoriji Implementacija ove heuristike oslanja se na vrednost atributa `NodeCountMap`. Ukoliko je za neku od grana izdvojen rečnik atributa `NodeCountMap` koji sadrži čvorove tipa `StoreField`, `RawStore` ili `StoreIndexed`, za tu granu smatra se da je grana koja sadrži operaciju čuvanja u memoriji i dodeljuje joj se verovatnoća izvršavanja od 0.45.

Heuristika zaštite registara Implementacija ove heuristike oslanja se na vrednost atributa `FNodeCountMap` obzirom da pokretni čvorovi u *Graal IR* grafu modeluju tok podataka u programu na sličan način na koji to registri čine u binarnom fajlu programa. Ukoliko je za neku od grana izdvojen rečnik atributa `FNodeCountMap` koji sadrži čvorove tipa `Unary`, `Binary`, `LogicNegation`, `UnaryOpLogic` ili `BinaryOpLogic`, za tu granu predviđa se da će biti izvršena sa verovatnoćom 0.62.

Heuristika poziva Implementacija ove heuristike oslanja se na vrednost atributa `NodeCountMap`. Ukoliko je za neku od grana izdvojen rečnik atributa `NodeCountMap` koji sadrži čvorove tipa `Invoke` ili `InvokeWithException`, za tu granu smatra se da je grana koja neće biti izvršena i dodeljuje joj se verovatnoća izvršavanja od 0.22.

Heuristika koda operacije Kompilator *Enterprise GraalVM Native Image* parsira poređenja celobrojnih vrednosti u *Graal IR* čvorove tipa `IntegerLowerThan` ili `IntegerEquals`. Ovi čvorovi koriste se da bi se predvidela verovatnoća od 0.84 granama koje odgovaraju

neuspešnom poređenju celih brojeva po uslovima: manje od nule, manje ili jednako nuli, ili jednako nekoj konstantnoj vrednosti. Kako kompilator *Enterprise GraalVM Native Image* predstavlja konstante pomoću čvora tipa *Constant* u Graal IR grafu, to se upravo taj tip čvora koristi za identifikaciju konstantnih vrednosti.

Heuristika poređenja pokazivača U *Graal IR* grafu svaki čvor naredbe grananja kao ulaz ima odgovarajuću vrednost pokretnog čvora tipa `Logic` koji proverava da li je ispunjen uslov grananja. Naredbe grananja koje odgovaraju poređenju pokazivača definisane su kao naredbe grananja kod kojih je pokretni čvor koji određuje uslov grananja *Graal IR* čvor tipa `PointerEquals`. Ova heuristika predviđa verovatnoću izvršavanja 0.60 za grane koje će biti izvršene u slučaju neuspešne provere uslova u čvoru `PointerEquals`.

Heuristika početka petlje Ova heuristika oslanja se na činjenicu da se početak petlje u *Graal IR* grafu obeležava čvorom `LoopBegin` (na primer, blok *B1* sa slike 4.2). Na taj način jednostavno se identifikuju grane koje pokazuju na početak petlje (engl. *loop header*) i predviđa njihovo izvršavanje sa verovatnoćom 0.75.

Heuristika izlaska iz petlje Ova heuristika identifikuje naredbe grananja koje se nalaze u petljama kao čvorove u *Graal IR* grafu tipa `If` kod kojih jedna grana vodi ka izlasku iz petlje (pokazuje na čvor tipa `LoopExit`) a druga grana nastavlja izvršavanje petlje i ne pokazuje na početak petlje. U tom slučaju, ova heuristika dodeljuje verovatnoću izvršavanja 0.80 grani koja nastavlja izvršavanje petlje.

Važno je napomenuti da se heuristika izlaska iz petlje i heuristika petlje odnose na različite naredbe grananja. Heuristika izlaska iz petlje usmerena je na naredbe grananja unutar tela petlje, dok je heuristika petlje usmerena na naredbe grananja koje odgovaraju petljama iz izvornog koda programa (na primer čvor `1. If` sa slike 4.2 odgovara `for` petlji iz funkcije `heapSort`).

Kao što je predloženo u radu [326], prilikom implementacije je korišćena Dempster–Šafer teorija dokaza za kombinovanje predviđanja heuristika [111]. Statički profajler koji koristi heuristike Vua i Larusa implementiran je kao posebna faza unutar kompilatora *Enterprise GraalVM Native Image*.

Bibliografija

[1] Hervé Abdi and Lynne J Williams. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, 2(4):433–459, 2010.

[2] Jeanne C Adams, Walter S Brainerd, Jeanne T Martin, Brian T Smith, and Jerrold L Wagener. *Fortran 90 Handbook*, volume 32. McGraw-Hill, New York, 1992.

[3] Charu C Aggarwal and CRD Clustering. Algorithms and applications, 2014.

[4] Amer FAH Alnuaimi and Tasnim HK Albaldawi. An overview of machine learning classification techniques. In *BIO Web of Conferences*, volume 97, page 00133. EDP Sciences, 2024.

[5] Pietro Alovisi. Static Branch Prediction through Representation Learning, 2020.

[6] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2020.

[7] Charles J Alpert, Te C Hu, Jen-Hsin Huang, Andrew B Kahng, and David Karger. Prim-Dijkstra tradeoffs for improved performance-driven routing tree design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(7):890–896, 1995.

[8] Shunichi Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5(4-5):185–196, 1993.

[9] Ken Arnold, James Gosling, David Holmes, and David Holmes. *The Java programming language*, volume 2. Addison-wesley Reading, 2000.

[10] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. *ACM SIGPLAN Notices*, 32(5):134–145, 1997.

[11] John Backus. The history of Fortran I, II, and III. *ACM Sigplan Notices*, 13(8):165–180, 1978.

[12] David F Bacon, Susan L Graham, and Oliver J Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994.

[13] Thomas Ball and James R Larus. Branch prediction for free. *ACM SIGPLAN Notices*, 28(6):300–313, 1993.

[14] Thomas Ball and James R Larus. Efficient path profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 46–57. IEEE, 1996.

[15] Antoine Barbez, Foutse Khomh, and Yann-Gaël Guéhéneuc. A machine-learning based ensemble method for anti-patterns detection. *Journal of Systems and Software*, 161:110486, 2020. `https://www.sciencedirect.com/science/article/pii/S0164121219302602`.

[16] Kenneth Barclay and John Savage. *Groovy programming: an introduction for Java developers*. Elsevier, 2010.

[17] Horace B Barlow. Unsupervised learning. *Neural computation*, 1(3):295–311, 1989.

[18] Candice Bentéjac, Anna Csörgő, and Gonzalo Martínez-Muñoz. A comparative analysis of gradient boosting algorithms. *Artificial Intelligence Review*, 54:1937–1967, 2021.

[19] Daniel Berrar et al. Cross-Validation, 2019.

[20] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: Requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1):1–29, 2019.

[21] Andrzej Białecki, Robert Muir, Grant Ingersoll, and Lucid Imagination. Apache lucene 4. In *SIGIR 2012 workshop on open source information retrieval*, page 17, 2012.

[22] Gérard Biau and Erwan Scornet. A random forest guided tour. *Test*, 25:197–227, 2016.

[23] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, 2006.

[24] Bruno Blanchet. Escape analysis for object-oriented languages: application to Java. *Acm Sigplan Notices*, 34(10):20–34, 1999.

[25] David Blei, Andrew Ng, and Michael Jordan. Latent dirichlet allocation. *Advances in neural information processing systems*, 14, 2001.

[26] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[27] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM review*, 60(2):223–311, 2018.

[28] Oliver Bračevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. Graph IRS for impure higher-order languages: making aggressive optimizations affordable with precise effect dependencies. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):400–430, 2023.

[29] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction*, pages 201–211, 2020.

[30] Leo Breiman. *Classification and regression trees*. Routledge, 2017.

[31] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.

[32] Mr Brijain, R Patel, Mr Kushik, and K Rana. A survey on decision tree algorithm for classification. 2014.

[33] Nathan Bronson and the Scala STM Expert Group. Library-Based Software Transactional Memory for Scala, 2023. `https://nbronson.github.io/scala-stm/faq.html`.

[34] Rodrigo Bruno, Vojin Jovanovic, Christian Wimmer, and Gustavo Alonso. Compiler-assisted object inlining with value fields. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 128–141, 2021.

[35] Michael Burke and Linda Torczon. Interprocedural optimization: eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(3):367–399, 1993.

[36] Raymond PL Buse and Westley Weimer. The road not taken: Estimating path execution frequency statically. In *2009 IEEE 31st International Conference on Software Engineering*, pages 144–154. IEEE, 2009.

[37] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):188–222, 1997.

[38] Canonical Ltd. Ubuntu Manpages, 2023. `https://manpages.ubuntu.com/`.

[39] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning*, pages 161–168, 2006.

[40] Brian Case. Spec95 retires spec92. *Microprocessor Report*, 9(11):11–14, 1995.

[41] Girish Chandrashekar and Ferat Sahin. A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1):16–28, 2014.

[42] Pohua P Chang, Scott A Mahlke, William Y Chen, and Wen-Mei W Hwu. Profile-guided automatic inline expansion for c programs. *Software: Practice and Experience*, 22(5):349–369, 1992.

[43] Pohua P Chang, Scott A Mahlke, William Y Chen, Nancy J Warter, and Wen-mei W Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. *ACM SIGARCH Computer Architecture News*, 19(3):266–275, 1991.

[44] Pohua P Chang, Scott A Mahlke, and Wen-Mei W Hwu. Using profile information to assist classic code optimizations. *Software: Practice and Experience*, 21(12):1301–1321, 1991.

[45] Bernard Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM (JACM)*, 47(6):1028–1047, 2000.

[46] Dehao Chen, David Xinliang Li, and Tipp Moseley. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 12–23, 2016.

[47] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for fdo compilation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 42–52, 2010.

[48] Lianping Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE software*, 32(2):50–54, 2015.

[49] Lianping Chen. Continuous delivery: overcoming adoption challenges. *Journal of Systems and Software*, 128:72–86, 2017.

[50] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM. `http://doi.acm.org/10.1145/2939672.2939785`.

[51] Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, Kailong Chen, Rory Mitchell, Ignacio Cano, Tianyi Zhou, et al. XGBoost: extreme gradient boosting. *R package version 0.4-2*, 1(4):1–4, 2015.

[52] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *11th IEEE Symposium on Computers and Communications (ISCC'06)*, pages 749–754. IEEE, 2006.

[53] Mitch Cherniack, Eduardo F Galvez, Michael J Franklin, and Stan Zdonik. Profile-driven cache management. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*, pages 645–656. IEEE, 2003.

[54] Ian D Chivers and Jane Sleightholme. *Introduction to programming with Fortran*, volume 2. Springer, 2018.

[55] Eun-young Cho. Jprofiler: Code coverage analysis tool for OMP project. *Technical Report: CMU 17–654 & 17, Tech. Rep.*, 2006.

[56] Hyoun Kyu Cho, Tipp Moseley, Richard Hank, Derek Bruening, and Scott Mahlke. Instant profiling: Instrumentation sampling for profiling datacenter applications. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10. IEEE, 2013.

[57] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C Sreedhar, and Sam Midkiff. Escape analysis for Java. *Acm Sigplan Notices*, 34(10):1–19, 1999.

[58] Cliff Click. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 246–257, 1995.

[59] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre Van Schaik. EMNIST: Extending MNIST to handwritten letters. In *2017 international joint conference on neural networks (IJCNN)*, pages 2921–2926. IEEE, 2017.

[60] Oracle Company. GraalVM, 2024. Accessed: 2024-11-30.

[61] Thomas M Conte, Burzin A Patel, Kishore N Menezes, and J Stan Cox. Hardware-based profiling: An effective technique for profile-driven optimization. *International Journal of Parallel Programming*, 24:187–206, 1996.

[62] Keith D Cooper, Mary Hall, Ken Kennedy, and Linda Torczon. Interprocedural analysis and optimization. *Communications on Pure and Applied Mathematics*, 48(9):947–1003, 1995.

[63] Oracle Corporation. GraalVM: Run Programs Faster Anywhere, 2024. Accessed: 2024-12-07.

[64] Oracle Corporation. Ideal Graph visualizer, 2024. `https://www.graalvm.org/latest/tools/igv/`.

[65] Michael J Crawley. *The R book*. John Wiley & Sons, 2012.

[66] Douglas Crockford. *JavaScript: The Good Parts*. „ O'Reilly Media, Inc.", 2008.

[67] Milan Čugurović and Milena Vujošević Janičić. GraalSP Profiles Logger: A Tool for Analyzing and Interpreting Predictions of the ML-Based Static Profilers. In *ARTIFICAL INTELLIGENCE CONFERENCE*, page 15, 2024.

[68] Milan Čugurović, Milena Vujošević Janičić, Vojin Jovanović, and Thomas Würthinger. GraalSP: Polyglot, efficient, and robust machine learning-based static profiler. *Journal of Systems and Software*, 213:112058, 2024.

[69] Milan Čugurović. Dinamičko prikupljanje i statičko predviđanje profila izvršavanja programa u kontekstu kompajlerskih optimizacija. *InfoM-Časopis za informacione tehnologije i multimedijalne sisteme*, 23(78):17–27, 2023.

[70] Milan Čugurović and Milena Vujošević Janičić. Kreiranje i analiza skupa podataka za predviđanje verovatnoća izvršavanja virtualnih metoda u objektno-orijentisanim programima. In *Artificial Intelligence*, Serbia, December 2023.

[71] Chris Cummins, Zacharias V Fisches, Tal Ben-Nun, Torsten Hoefler, Michael FP O'Boyle, and Hugh Leather. PROGRAML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *International Conference on Machine Learning*, pages 2244–2253. PMLR, 2021.

[72] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[73] Mwamba Kasongo Dahouda and Inwhee Joe. A deep-learned embedding technique for categorical features encoding. *IEEE Access*, 9:114381–114391, 2021.

[74] Sivarama P Dandamudi. MIPS Processor. *Introduction to Assembly Language Programming: For Pentium and RISC Processors*, pages 347–360, 2005.

[75] Yann Dauphin, Harm De Vries, and Yoshua Bengio. Equilibrated adaptive learning rates for non-convex optimization. *Advances in neural information processing systems*, 28, 2015.

[76] Jack W Davidson and Sanjay Jinturkar. An aggressive approach to loop unrolling. Technical report, Citeseer, 1995.

[77] Jeffrey Dean and Craig Chambers. Towards better inlining decisions using inlining trials. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 273–282, 1994.

[78] Brian L Deitrich, Ben Chung Chen, and WW Hwu. Improving static branch prediction in a compiler. In *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*, pages 214–221. IEEE, 1998.

[79] Delphine Demange, Yon Fernández de Retana, and David Pichardie. Semantic reasoning about the sea of nodes. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 163–173, 2018.

[80] Li Deng. The MNIST database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[81] Veerle Desmet, Lieven Eeckhout, and Koen De Bosschere. Using decision trees to improve program-based and profile-based static branch prediction. In *Asia-Pacific Conference on Advances in Computer Systems Architecture*, pages 336–352. Springer, 2005.

[82] David Detlefs and Ole Agesen. Inlining of virtual methods. In *European Conference on Object-Oriented Programming*, pages 258–277. Springer, 1999.

[83] ONNX Runtime developers. ONNX Runtime. `https://onnxruntime.ai/`, 2021. Version: x.y.z.

[84] Pradip Dhal and Chandrashekhar Azad. A comprehensive survey on feature selection in the various fields of machine learning. *Applied Intelligence*, pages 1–39, 2022.

[85] Tom Dietterich. Overfitting and undercomputing in machine learning. *ACM computing surveys (CSUR)*, 27(3):326–327, 1995.

[86] Kaivalya M Dixit. Overview of the SPEC Benchmarks. *The Benchmark Handbook*, 7, 1993.

[87] Paul DuBois. *MySQL*. Addison-Wesley Professional, 2013.

[88] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal IR: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[89] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*, pages 1–10, 2013.

[90] Yoshiyuki Endo. Estimate of confidence intervals for geometric mean diameter and geometric standard deviation of lognormal size distribution. *Powder technology*, 193(2):154–161, 2009.

[91] Epic-Games-Inc. Unreal Engine 5.0 Documentation, 2022. `https://docs.unrealengine.com/5.0/en-US/`.

[92] Michael Fagan. Design and code inspections to reduce errors in program development. In *Software pioneers: contributions to software engineering*, pages 575–607. Springer, 2011.

[93] Matthias Feurer and Frank Hutter. Hyperparameter optimization. *Automated machine learning: Methods, systems, challenges*, pages 3–33, 2019.

[94] Joseph A Fisher and Stefan M Freudenberger. Predicting conditional branch directions from previous runs of a program. *ACM SIGPLAN Notices*, 27(9):85–95, 1992.

[95] David Flanagan and Yukihiro Matsumoto. *The Ruby programming language*. „ O'Reilly Media, Inc.", 2008.

[96] Philip J Fleming and John J Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, 1986.

[97] Free Software Foundation. GNU gcc, 2013. on-line at: `http://gcc.gnu.org/`.

[98] Martin Fowler and Matthew Foemmel. Continuous integration, 2006.

[99] Casper B Freksen, Lior Kamma, and Kasper Green Larsen. Fully understanding the hashing trick. *Advances in Neural Information Processing Systems*, 31, 2018.

[100] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[101] Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12:381–391, 1999.

[102] Shudi Sandy Gao, C Michael Sperberg-McQueen, and Henry Thompson. W3C XML schema definition language (XSD) 1.1 part 1: Structures. 2012.

[103] GCC Developers. Branch prediction definitions – file predict.def. `https://github.com/gcc-mirror/gcc/blob/master/gcc/predict.def`, 2024. Accessed: 2025-06-28.

[104] Zoubin Ghahramani. Unsupervised learning. In *Summer school on machine learning*, pages 72–112. Springer, 2003.

[105] Dhiren Ghosh and Andrew Vogt. Outliers: An evaluation of methodologies. In *Joint statistical meetings*, volume 12, pages 3455–3460, 2012.

[106] GithHub. Source Code of the feature-genetic benchmark from the Renaissance suite, 2024. `https://github.com/renaissance-benchmarks/renaissance/blob/487cb2072afb37661563e5102559303b5ba26b4c/benchmarks/jdk-concurrent/src/main/scala/org/renaissance/jdk/concurrent/FutureGenetic.scala#L36`.

[107] GithHub. Source Code of the indexDocs function from the Dacapo Luindex Benchmark, 2024. `https://github.com/dacapobench/dacapobench/blob/fd292e92f8c40495a6ca05ff3b8a77c6c4265606/benchmarks/bms/luindex/src/org/dacapo/luindex/Index.java#L177`.

[108] Andrew S Glassner. *An introduction to ray tracing*. Morgan Kaufmann, 1989.

[109] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[110] Google-Corporation. Chromium Documentation. 2023.

[111] Jean Gordon and Edward H Shortliffe. The Dempster-Shafer theory of evidence. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, 3:832–838, 1984.

[112] Cyril Goutte and Eric Gaussier. A probabilistic interpretation of precision, recall and F-score, with implication for evaluation. In *European conference on information retrieval*, pages 345–359. Springer, 2005.

[113] Ronald L Graham and Pavol Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43–57, 1985.

[114] Daniel Granot and Gur Huberman. Minimum cost spanning tree games. *Mathematical programming*, 21:1–18, 1981.

[115] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. LSTM: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2016.

[116] Patrick J Grother and KK Hanaoka. NIST special database 19. *Handprinted forms and characters database, National Institute of Standards and Technology*, 10:69, 1995.

[117] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: a benchmark for software transactional memory. Technical report, 2006.

[118] Rajiv Gupta, Eduard Mehofer, and Youtao Zhang. Profile guided compiler optimizations. 2002.

[119] Gerald J Hahn. The coefficient of determination exposed. *Chemtech*, 3(10):609–612, 1973.

[120] Mohamad H Hassoun et al. *Fundamentals of artificial neural networks*. MIT press, 1995.

[121] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.

[122] Douglas M Hawkins. The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12, 2004.

[123] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[124] Wenlei He, Julián Mestre, Sergey Pupyrev, Lei Wang, and Hongtao Yu. Profile inference revisited. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–24, 2022.

[125] Marti A. Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. Support vector machines. *IEEE Intelligent Systems and their applications*, 13(4):18–28, 1998.

[126] Steve Heath. *Embedded systems design*. Elsevier, 2002.

[127] John L Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[128] Thomas A Henzinger and Joseph Sifakis. The embedded systems design challenge. In *International Symposium on Formal Methods*, pages 1–15. Springer, 2006.

[129] HSQLDB Group. H2 database engine, 2023. `https://www.h2database.com/html/main.html`.

[130] Jung-Chang Huang and Tau Leng. Generalized loop-unrolling: a method for program speedup. In *Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No. PR00122)*, pages 244–248. IEEE, 1999.

[131] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and J Doug Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 43–58, 2011.

[132] John Hunt. *Beginner's Guide to Kotlin Programming*. Springer, 2021.

[133] Ross Ihaka and Robert Gentleman. R: a language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[134] Shams M Imam and Vivek Sarkar. Savina-an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, pages 67–80, 2014.

[135] SPARC International Inc and David L Weaver. *The SPARC architecture manual*. Prentice-Hall Englewood Cliffs, NJ, USA, 1994.

[136] LLVM Compiler Infrastructure. LLVM language reference manual, 2022.

[137] Anil K Jain, M Narasimha Murty, and Patrick J Flynn. Data clustering: a review. *ACM computing surveys (CSUR)*, 31(3):264–323, 1999.

[138] Sandra Johnson and S Valli. Hot method prediction using support vector machines. *Ubiquitous Computing and Communication Journal*, 3(4):75–81, 2008.

[139] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[140] Vojin Jovanović, Milan Čugurović, and Lazar Milikić. GraalNN: Context-Sensitive Static Profiling with Graph Neural Networks, 2024. Patent Application.

[141] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[142] David R Karger, Philip N Klein, and Robert E Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM (JACM)*, 42(2):321–328, 1995.

[143] Patrick Keegan, Ludovic Champenois, Gregory Crawley, Charlie Hunt, and Christopher Webster. *Netbeans™ ide field guide: developing desktop, web, enterprise, and mobile applications*. Prentice Hall Press, 2005.

[144] James M Keller, Michael R Gray, and James A Givens. A fuzzy k-nearest neighbor algorithm. *IEEE transactions on systems, man, and cybernetics*, (4):580–585, 1985.

[145] Brian W Kernighan and Dennis M Ritchie. The C programming language. 2002.

[146] Aaron Kershenbaum and Richard Van Slyke. Computing minimum spanning trees efficiently. In *Proceedings of the ACM annual conference-Volume 1*, pages 518–527, 1972.

[147] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[148] Kingsoft Office Software. WPS Office. `https://www.wps.com/`, 2025. Accessed: 2025-04-12.

[149] Andi Kleen. A NUMA API for Linux. *Novel Inc*, 2005.

[150] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

[151] Donald E Knuth and Francis R Stevenson. Optimal measurement points for program frequency counts. *BIT Numerical Mathematics*, 13(3):313–322, 1973.

[152] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in action*. Manning Publications Co., 2007.

[153] Stephen Kokoska and Daniel Zwillinger. *CRC standard probability and statistics tables and formulae*. Crc Press, 2000.

[154] Sotiris B Kotsiantis, Dimitris Kanellopoulos, and Panagiotis E Pintelas. Data preprocessing for supervised learning. *International Journal of Computer Science*, 1(2):111–117, 2006.

[155] Sotiris B Kotsiantis, Ioannis Zaharakis, P Pintelas, et al. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160(1):3–24, 2007.

[156] Jarosław Krochmalski. *IntelliJ IDEA Essentials*. Packt Publishing Ltd, 2014.

[157] Prasad A Kulkarni. JIT compilation policy for modern machines. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 773–788, 2011.

[158] Christopher Kulla. Sunflow Render System Documentation, 2023. `https://sunflow.sourceforge.net/`.

[159] Vipin Kumar and Sonajharia Minz. Feature selection: a literature review. *SmartCR*, 4(3):211–229, 2014.

[160] Wojciech Kunikowski, Ernest Czerwiński, Paweł Olejnik, and Jan Awrejcewicz. An overview of ATmega AVR microcontrollers used in scientific research and industrial applications. *Pomiary Automatyka Robotyka*, 19(1):15–19, 2015.

[161] Alexey Kurakin, Ian J Goodfellow, and Samy Bengio. Adversarial Machine Learning at Scale. In *International Conference on Learning Representations*, 2016.

[162] Takio Kurita. Principal component analysis (PCA). *Computer Vision: A Reference Guide*, pages 1–4, 2019.

[163] Peter J Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[164] Michael Larionov. Sampling techniques in Bayesian target encoding. *arXiv preprint arXiv:2006.01317*, 2020.

[165] Chris Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, volume 5, pages 1–20, 2008.

[166] Chris Lattner. LLVM and API reference documentation: How To Build Clang and LLVM with Profile-Guided Optimizations, 2020. `https://llvm.org/docs/HowToBuildWithPGO.html`.

[167] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[168] Alvin R Lebeck and David A Wood. Cache profiling and the SPEC benchmarks: A case study. *Computer*, 27(10):15–26, 1994.

[169] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[170] Daniel Lehmann and Michael O Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 133–138, 1981.

[171] David Leopoldseder. Simulation-based code duplication for enhancing compiler optimizations. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 10–12, 2017.

[172] David Leopoldseder. Simulation-Based Code Duplication in a Dynamic Compiler. 2019.

[173] David Leopoldseder, Roland Schatz, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. Fast-path loop unrolling of non-counted loops to enable subsequent compiler optimizations. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, pages 1–13, 2018.

[174] David Leopoldseder, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. A Cost Model for a Graph-based Intermediate-representation in a Dynamic Compiler. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, pages 26–35, 2018.

[175] David Leopoldseder, Lukas Stadler, Christian Wimmer, and Hanspeter Mössenböck. Java-to-JavaScript translation via structured control flow reconstruction of compiler IR. *ACM SIGPLAN Notices*, 51(2):91–103, 2015.

[176] David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. Dominance-based duplication simulation (DBDS): code duplication to enable compiler optimizations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 126–137, 2018.

[177] Robert I Lerman and Shlomo Yitzhaki. A note on the calculation and interpretation of the Gini index. *Economics Letters*, 15(3-4):363–368, 1984.

[178] David Xinliang Li, Raksit Ashok, and Robert Hundt. Lightweight feedback-directed cross-module optimization. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 53–61, 2010.

[179] Jundong Li, Kewei Cheng, Suhang Wang, Fred Morstatter, Robert P Trevino, Jiliang Tang, and Huan Liu. Feature selection: A data perspective. *ACM Computing Surveys (CSUR)*, 50(6):1–45, 2017.

[180] Hai Liu, Neal Glew, Leaf Petersen, and Todd A Anderson. The Intel labs Haskell research compiler. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, pages 105–116, 2013.

[181] Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. On the Variance of the Adaptive Learning Rate and Beyond. In *International Conference on Learning Representations*, 2019.

[182] Yanli Liu, Yuan Gao, and Wotao Yin. An improved analysis of stochastic gradient descent with momentum. *Advances in Neural Information Processing Systems*, 33:18261–18271, 2020.

[183] Stuart Lloyd. Least squares quantization in PCM. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[184] Ricardo Llugsi, Samira El Yacoubi, Allyx Fontaine, and Pablo Lupera. Comparison between Adam, AdaMax and Adam W optimizers to implement a Weather Forecast based on Neural Networks for the Andean city of Quito. In *2021 IEEE Fifth Ecuador Technical Chapters Meeting (ETCM)*, pages 1–6. IEEE, 2021.

[185] LLVM Project. How To Build Clang and LLVM with Profile-Guided Optimizations. `https://llvm.org/docs/HowToBuildWithPGO.html`. Accessed: 2024-11-29.

[186] Lucene, Apache. Apache lucene-overview, 2010. `http://lucene.apache.org/iava/docs/`.

[187] C-K Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. Ispike: a post-link optimizer for the Intel/spl reg/Itanium/spl reg/architecture. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 15–26. IEEE, 2004.

[188] Andrzej Maćkiewicz and Waldemar Ratajczak. Principal components analysis (PCA). *Computers & Geosciences*, 19(3):303–342, 1993.

[189] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[190] Amitav Mahapatra and Prashanta Kumar Patra. Support Vector Machine for Frequently Executed Method Prediction. In *2018 International Conference on Information Technology (ICIT)*, pages 193–198. IEEE, 2018.

[191] S Manikandan. Measures of central tendency: The mean. *Journal of Pharmacology and Pharmacotherapeutics*, 2(2):140, 2011.

[192] Abu Naser Masud and Federico Ciccozzi. More precise construction of static single assignment programs using reaching definitions. *Journal of Systems and Software*, 166:110590, 2020. `https://www.sciencedirect.com/science/article/pii/S0164121220300704`.

[193] Dastan Maulud and Adnan M Abdulazeez. A review on linear regression comprehensive in machine learning. *Journal of Applied Science and Technology Trends*, 1(2):140–147, 2020.

[194] Andrew McCallum, Karl Schultz, and Sameer Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. *Advances in Neural Information Processing Systems*, 22, 2009.

[195] Larry Medsker and Lakhmi C Jain. *Recurrent neural networks: design and applications*. CRC press, 1999.

[196] Edu Metz and Raimondas Lencevicius. Efficient instrumentation for performance profiling. *arXiv preprint cs/0307058*, 2003.

[197] Mathias Meyer. Continuous integration and its tools. *IEEE software*, 31(3):14–16, 2014.

[198] Tommi Mikkonen and Antero Taivalsaari. Using JavaScript as a real programming language, 2007.

[199] Lazar Milikic, Milan Cugurovic, and Vojin Jovanovic. GraalNN: Context-Sensitive Static Profiling with Graph Neural Networks. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, pages 123–136, 2025.

[200] Michael Minella. State of spring survey 2024 results. VMWare Tanzu *Spring* blog, June 2024. Accessed: 2025-07-07.

[201] Tom Michael Mitchell et al. *Machine learning*, volume 1. McGraw-hill New York, 2007.

[202] Angélica Aparecida Moreira, Guilherme Ottoni, and Fernando Magno Quintão Pereira. VESPA: static profiling for binary optimization. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–28, 2021.

[203] Mozilla-Foundation. Firefox Source Tree Documentation. 2023.

[204] Frank Mueller and David B Whalley. Avoiding unconditional jumps by code replication. *ACM SIGPLAN Notices*, 27(7):322–330, 1992.

[205] Frank Mueller and David B Whalley. Avoiding conditional branches by code replication. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 56–66, 1995.

[206] Adam Myatt. *Pro NetBeans IDE 6 Rich Client Platform Edition*. Apress, 2008.

[207] Nico JD Nagelkerke et al. A note on a general definition of the coefficient of determination. *biometrika*, 78(3):691–692, 1991.

[208] Stefano Nembrini, Inke R König, and Marvin N Wright. The revival of the Gini importance? *Bioinformatics*, 34(21):3711–3718, 2018.

[209] IDE NetBeans. NetBeans IDE. *Obtenido de https://netbeans. org*, 2013.

[210] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 65–74, 2007.

[211] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[212] Feedforward Neural Network. Optimal Unsupervised Learning in a Single-Layer Linear. *Neural Networks*, 2:459–473, 1989.

[213] Mladen Nikolić. Mašinsko učenje - skripta. `https://ml.matf.bg.ac.rs/readings/ml.pdf`, 2024. Accessed: 2024-12-12.

[214] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 51–64, 2002.

[215] William S Noble. What is a support vector machine? *Nature biotechnology*, 24(12):1565–1567, 2006.

[216] Diego Novillo. SamplePGO-the power of profile guided optimizations without the usability burden. In *2014 LLVM Compiler Infrastructure in HPC*, pages 22–28. IEEE, 2014.

[217] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. 2004.

[218] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc, 2008.

[219] University of Toronto. The Boston Housing Dataset. `https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html`. Accessed: 2024-12-12.

[220] Tiit Oja. Optimizing JVM profiling performance for Honest Profiler.

[221] Lubos Omelina, Jozef Goga, Jarmila Pavlovicova, Milos Oravec, and Bart Jansen. A survey of iris datasets. *Image and Vision Computing*, 108:104109, 2021.

[222] Oracle. GraalVM Reachability Metadata Repository. https://github.com/oracle/graalvm-reachability-metadata, 2023. 2023-07-27.

[223] Oracle (GraalVM Team). Github page: graalvm/setup-graalvm. `https://github.com/graalvm/setup-graalvm/network/dependents`, 2025. Accessed: 2025-07-07.

[224] Hector Veiga Ortiz and Piyush Mishra. *Akka Cookbook*. Packt Publishing Ltd, 2017.

[225] FY Osisanwo, JET Akinsola, O Awodele, JO Hinmikaiye, O Olakanmi, J Akinjobi, et al. Supervised machine learning algorithms: classification and comparison. *International Journal of Computer Trends and Technology (IJCTT)*, 48(3):128–138, 2017.

[226] Guilherme Ottoni and Bertrand Maher. Optimizing function placement for large-scale data-center applications. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 233–244. IEEE, 2017.

[227] Michael Paleczny, Christopher Vick, and Cliff Click. The Java {HotSpot™} server compiler. In *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*, 2001.

[228] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2019.

[229] Alexandre Passos, Luke Vilnis, and Andrew McCallum. Optimization and learning in factorie. In *NIPS Workshop on Optimization for Machine Learning (OPT)*. Citeseer, 2013.

[230] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[231] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[232] perf contributors. Linux perf Wiki. `https://perfwiki.github.io/main/`, 2024. Accessed: 2024-12-07.

[233] Ramesh V Peri, Sanjay Jinturkar, and Lincoln Fajardo. A novel technique for profiling programs in embedded systems. In *ACM Workshop on Feedback-Directed and Dynamic Optimization*, 1999.

[234] Leif E Peterson. K-nearest neighbor. *Scholarpedia*, 4(2):1883, 2009.

[235] Karl Pettis and Robert C. Hansen. Profile Guided Code Positioning. *SIGPLAN Not.*, 25(6):16–27, June 1990. `https://doi.org/10.1145/93548.93550`.

[236] PMD Open Source Project. Pmd documentation, 2023. `https://docs.pmd-code.org/latest/index.html`.

[237] Ekaterina Poslavskaya and Alexey Korolev. Encoding categorical data: Is there yet anything'hotter'than one-hot encoding? *arXiv preprint arXiv:2312.16930*, 2023.

[238] GNU Project. *GNU gprof: a Call Graph Execution Profiler*, 1998. Accessed: 2024-12-07.

[239] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Wïrthinger. An optimization-driven incremental inline substitution algorithm for just-in-time compilers. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 164–179. IEEE, 2019.

[240] Aleksandar Prokopec, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger. Making collection operations optimal with aggressive JIT compilation. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, pages 29–40, 2017.

[241] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tuma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, et al. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–47, 2019.

[242] Easwaran Raman and Xinliang David Li. Learning branch probabilities in compiler from datacenter workloads. *arXiv preprint arXiv:2202.06728*, 2022.

[243] ReactiveX. ReactiveX Documentation, 2023. `https://reactivex.io/documentation/observable.html`.

[244] James Reinders. *VTune performance analyzer essentials*, volume 9. Intel Press Santa Clara, 2005.

[245] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE micro*, 30(4):65–79, 2010.

[246] Colin Renouf. The IBM J9 Java Virtual Machine for Java 6. *Pro IBM® WebSphere® Application Server 7 Internals*, pages 15–34, 2009.

[247] Stephen Richardson and Mahadevan Ganapathi. Interprocedural optimization: Experimental results. *Software: Practice and Experience*, 19(2):149–169, 1989.

[248] Manuel Rigger, Matthias Grimmer, and Hanspeter Mössenböck. Sulong-execution of LLVM-based languages on the JVM: Position paper. In *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, pages 1–4, 2016.

[249] Ivan Ristović, Milan Čugurović, Strahinja Stanojević, Marko Spasić, Vesna Marinković, and Milena Vujošević Janičić. Efikasan obilazak grafova kontrole toka. In *YuInfo 2024*, pages 89–94, Kopaonik, March 2024.

[250] Pau Rodríguez, Miguel A Bautista, Jordi Gonzalez, and Sergio Escalera. Beyond one-hot encoding: Lower dimensional target embedding. *Image and Vision Computing*, 75:21–31, 2018.

[251] Lior Rokach and Oded Maimon. Clustering methods. *Data mining and knowledge discovery handbook*, pages 321–352, 2005.

[252] Lior Rokach and Oded Z Maimon. *Data mining with decision trees: theory and applications*, volume 69. World scientific, 2007.

[253] Shen Rong and Zhang Bao-Wen. The research of regression model in machine learning field. In *MATEC Web of Conferences*, volume 176, page 01033. EDP Sciences, 2018.

[254] Nadav Rotem and Chris Cummins. Profile guided optimization without profiles: A machine learning approach. *arXiv preprint arXiv:2112.14679*, 2021.

[255] S Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics*, 21(3):660–674, 1991.

[256] Omer Sagi and Lior Rokach. Ensemble learning: A survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(4):e1249, 2018.

[257] Stephen Samuel and Stefan Bocutiu. *Programming Kotlin*. Packt Publishing Ltd, 2017.

[258] Michel F Sanner et al. Python: a programming language for software integration and development. *J Mol Graph Model*, 17(1):57–61, 1999.

[259] Saptarsi Sanyal, Saroj Kumar Biswas, Dolly Das, Manomita Chakraborty, and Biswajit Purkayastha. Boston house price prediction using regression models. In *2022 2nd International Conference on Intelligent Technologies (CONIT)*, pages 1–6. IEEE, 2022.

[260] S Subramanya Sastry, Rastislav Bodik, and James E Smith. Rapid profiling via stratified sampling. *ACM SIGARCH Computer Architecture News*, 29(2):278–289, 2001.

[261] Sunil Saxena. Profile guided TLB and cache optimization, December 16 1997. US Patent 5,699,543.

[262] Scala Community. Reactors Documentation, 2023. `http://reactors.io/learn/`.

[263] ScalaXB org. ScalaXB, 2023. `https://scalaxb.org/`.

[264] Robert W Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9):647–654, 1977.

[265] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[266] Raimund Seidel. Understanding the inverse Ackermann function. In *Twenty-second European Workshop on Computational Geometry Delphi, Greece March 27–29, 2006*, page 37, 2006.

[267] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da Capo con Scala: design and analysis of a Scala benchmark suite for the Java virtual machine. In *Proceedings of the 2011 ACM international conference on Object-oriented programming systems languages and applications*, pages 657–676, 2011.

[268] Hezbullah Shah and Tariq Rahim Soomro. Node. js challenges in implementation. *Global Journal of Computer Science and Technology*, 17(2):73–83, 2017.

[269] Mohsen Shahhosseini, Guiping Hu, and Hieu Pham. Optimizing ensemble weights and hyperparameters of machine learning models for regression problems. *Machine Learning with Applications*, 7:100251, 2022.

[270] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE access*, 5:3909–3943, 2017.

[271] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[272] Ching-Yen Shih, Drake Svoboda, Siao-Jie Su, and Wei-Chung Liao. Static branch prediction for LLVM IRs Using Machine Learning. 2021.

[273] RA Singh et al. Analysis of SPEC CPUINT2006 benchmarksČ performance and classification. *Computer Science & Telecommunications*, 32(3), 2011.

[274] Anthony M Sloane. Lightweight language processing in Kiama. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 408–425. Springer, 2009.

[275] Anthony M Sloane and Matthew Roberts. Oberon-0 in kiama. *Science of Computer Programming*, 114:20–32, 2015.

[276] Gregory Smith. *PostgreSQL 9.0: High Performance*. Packt Publishing Ltd, 2010.

[277] Yan-Yan Song and LU Ying. Decision tree methods: applications for classification and prediction. *Shanghai archives of psychiatry*, 27(2):130, 2015.

[278] Aized Amin Soofi and Arshad Awan. Classification techniques in machine learning: applications and issues. *Journal of Basic & Applied Sciences*, 13:459–465, 2017.

[279] Marko Spasić, Ivan Ristović, Strahinja Stanojević, Milan Čugurović, Milica Karličić, and Milena Vujošević Janičić. Evaluacija performansi kompilatora GraalVM na distribuiranom računarskom klasteru. In *XV Srpski matematički kongres*, page 114, 2024.

[280] Satish Narayana Srirama, Freddy Marcelo Surriabre Dick, and Mainak Adhikari. Akka framework based on the Actor model for executing distributed Fog Computing applications. *Future Generation Computer Systems*, 117:439–452, 2021.

[281] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[282] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, and Thomas Würthinger. Compilation queuing and graph caching for dynamic compilers. In *Proceedings of the sixth ACM workshop on Virtual machines and intermediate languages*, pages 49–58, 2012.

[283] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. An experimental study of the influence of dynamic compiler optimizations on Scala performance. In *Proceedings of the 4th Workshop on Scala*, pages 1–8, 2013.

[284] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial escape analysis and scalar replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 165–174, 2014.

[285] Richard M Stallman et al. Using the GNU Compiler Collection. *Free Software Foundation*, 4(02), 2003.

[286] Richard M Stallman et al. Using the GNU Compiler Collection. *Free Software Foundation*, 2012.

[287] Ingo Steinwart and Andreas Christmann. *Support vector machines*. Springer Science & Business Media, 2008.

[288] Mark Stephenson and Saman Amarasinghe. Predicting unroll factors using nearest neighbors. 2004.

[289] Mervyn Stone. Cross-validation: A review. *Statistics: A Journal of Theoretical and Applied Statistics*, 9(1):127–139, 1978.

[290] Bjarne Stroustrup. The C++ Programming Language, 2013.

[291] Qiang Sun, Jianjun Zhao, and Yuting Chen. Probabilistic points-to analysis for Java. In *Compiler Construction: 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 20*, pages 62–81. Springer, 2011.

[292] Daniel Svozil, Vladimir Kvasnicka, and Jiri Pospichal. Introduction to multi-layer feed-forward neural networks. *Chemometrics and intelligent laboratory systems*, 39(1):43–62, 1997.

[293] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[294] The Document Foundation. Libreoffice, 2024. Accessed: 2025-04-11.

[295] Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. Understanding and exploiting optimal function inlining. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 977–989, 2022.

[296] David Thomas, Chad Fowler, and Andrew Hunt. *Programming Ruby*, volume 238. Pragmatic, 2004.

[297] tooomm. Github release stats for graalvmgraalvm-ce-builds. `https://tooomm.github.io/github-release-stats/?username=GraalVM&repository=graalvm-ce-builds`, 2025. Accessed: 2025-07-07.

[298] Vladimir Tsymbal and Alexandr Kurylev. Profiling Heterogeneous Computing Performance with VTune Profiler. In *Proceedings of the 9th International Workshop on OpenCL*, pages 1–1, 2021.

[299] Michael Tuchler, Andrew C Singer, and Ralf Koetter. Minimum mean squared error equalization using a priori information. *IEEE Transactions on Signal processing*, 50(3):673–683, 2002.

[300] Twiter Inc. Twiter Finagle, 2023. `https://twitter.github.io/finagle/`.

[301] Valgrind Developers. *Cachegrind: a Cache and Branch-Prediction Profiler*. Valgrind, 2024. Part of the Valgrind Documentation.

[302] Valgrind Developers. *Callgrind: a Call-Graph Generating Cache and Branch Prediction Profiler*. Valgrind, 2024. Part of the Valgrind Documentation.

[303] Valgrind Developers. *Memcheck: a Memory Error Detector*. Valgrind, 2024. Version 3.21.0.

[304] Vincent Van Der Leun. *Introduction to JVM Languages*. Packt Publishing Ltd, 2017.

[305] Laurens Van Der Maaten, Eric O Postma, H Jaap Van Den Herik, et al. Dimensionality reduction: A comparative review. *Journal of machine learning research*, 10(66-71):13, 2009.

[306] David A Van Dyk and Xiao-Li Meng. The art of data augmentation. *Journal of Computational and Graphical Statistics*, 10(1):1–50, 2001.

[307] Guido VanRossum and Fred L Drake. *The Python language reference*, volume 561. Python Software Foundation Amsterdam, The Netherlands, 2010.

[308] Richard M Vogel. The geometric mean? *Communications in Statistics-Theory and Methods*, 51(1):82–94, 2022.

[309] April W Wade, Prasad A Kulkarni, and Michael R Jantz. AOT vs. JIT: impact of profile data on code quality. In *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 1–10, 2017.

[310] Steven Wallace and Kim Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 209–220. IEEE, 2007.

[311] Craig Walls. *Spring in action*. Simon and Schuster, 2022.

[312] Zhou Wang and Alan C Bovik. Mean squared error: Love it or leave it? A new look at signal fidelity measures. *IEEE signal processing magazine*, 26(1):98–117, 2009.

[313] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 1113–1120, 2009.

[314] Matthew Edwin Weingarten, Theodoros Theodoridis, and Aleksandar Prokopec. Inlining-benefit prediction with interprocedural partial escape analysis. In *Proceedings of the 14th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, pages 13–24, 2022.

[315] Jason Wexbridge and Walter Nyland. Netbeans platform for beginners. *This book is for sale at http://leanpub. com/nbp4beginners*, 2014.

[316] John Whaley. A portable sampling-based profiler for Java virtual machines. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 78–87, 2000.

[317] Baptiste Wicht, Roberto A Vitillo, Dehao Chen, and David Levinthal. Hardware counted profile-guided optimization. *arXiv preprint arXiv:1411.6361*, 2014.

[318] Franz Wilhelmstötter. Jenetics. *URL: http://jenetics. io*, 2021.

[319] Kevin Wilson. Microsoft office 365. In *Using Office 365: With Windows 8*, pages 1–14. Springer, 2014.

[320] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B Kessler, Oleg Pliss, and Thomas Würthinger. Initialize once, start fast: application initialization at build time. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[321] Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, and Laurent Vanbever. P2GO: P4 profile-guided optimizations. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 146–152, 2020.

[322] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[323] Weng-Fai Wong. Source level static branch prediction. *The Computer Journal*, 42(2):142–149, 1999.

[324] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. An object storage model for the truffle language implementation framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 133–144, 2014.

[325] Raymond E Wright. Logistic regression. 1995.

[326] Youfeng Wu and James R Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 1–11, 1994.

[327] Zifeng Wu, Chunhua Shen, and Anton Van Den Hengel. Wider or deeper: Revisiting the resnet model for visual recognition. *Pattern Recognition*, 90:119–133, 2019.

[328] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on neural networks and learning systems*, 32(1):4–24, 2020.

[329] Thomas Würthinger. Extending the Graal compiler to optimize libraries. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 41–42, 2011.

[330] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 662–676, 2017.

[331] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 187–204, 2013.

[332] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, pages 73–82, 2012.

[333] Reda Yacouby and Dustin Axman. Probabilistic extension of precision, recall, and f1 score for more thorough evaluation of classification models. In *Proceedings of the first workshop on evaluation and comparison of NLP systems*, pages 79–91, 2020.

[334] Li Yang and Abdallah Shami. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, 415:295–316, 2020.

[335] Frank Yellin and Tim Lindholm. The Java virtual machine specification, 1996.

[336] YourKit. YourKit Java Profiler. `https://www.yourkit.com/`, 2024. Accessed: 2024-12-07.

[337] Shiwen Yu, Ting Wang, and Ji Wang. Data Augmentation by Program Transformation. *Journal of Systems and Software*, 190:111304, 2022. `https://www.sciencedirect.com/science/article/pii/S0164121222000541`.

[338] Ye Yuan, Liji Wu, and Xiangmin Zhang. Gini-impurity index analysis. *IEEE Transactions on Information Forensics and Security*, 16:3154–3169, 2021.

[339] Alina Yurenko. Graalvm community survey 2022 results. `https://medium.com/graalvm/graalvm-community-survey-2022-results-328d0404d36e`, 2022. Accessed: 2025-07-07.

[340] Stephen Zekany, Daniel Rings, Nathan Harada, Michael A Laurenzano, Lingjia Tang, and Jason Mars. CrystalBall: Statically analyzing runtime behavior via deep sequence learning. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[341] Cheng Zhang, Hao Xu, Sai Zhang, Jianjun Zhao, and Yuting Chen. Frequency estimation of virtual call targets for object-oriented programs. In *European Conference on Object-Oriented Programming*, pages 510–532. Springer, 2011.

[342] Zhongheng Zhang. Introduction to machine learning: k-nearest neighbors. *Annals of translational medicine*, 4(11):218, 2016.

[343] Zijun Zhang. Improved adam optimizer for deep neural networks. In *2018 IEEE/ACM 26th international symposium on quality of service (IWQoS)*, pages 1–2. Ieee, 2018.

[344] Xinghui Zhao and Nadeem Jamali. Load balancing non-uniform parallel computations. In *Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control*, pages 97–108, 2013.

[345] Yudi Zheng, Lubomír Bulej, and Walter Binder. Accurate profiling in the presence of dynamic compilation. *ACM SIGPLAN Notices*, 50(10):433–450, 2015.

[346] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI open*, 1:57–81, 2020.

[347] Patrick Ziegler. WebAssembly Code Generation from Java Bytecode. Master's thesis, ETH Zurich, 2023.

[348] École Polytechnique Fédérale Lausanne (EPFL) Lausanne, Switzerland. Scala Documentation, 2023. `https://www.scala-lang.org/`.

[349] Milan Čugurović, Ivan Ristović, Strahinja Stanojević, Marko Spasić, Vesna Marinković, and Milena Vujošević Janičić. Komparativna analiza algoritama obilaska grafova kontrole toka programa. In *Simpozijum MATEMATIKA I PRIMENE*, volume XIII, 2023.

[350] Milan Čugurović, Ivan Ristović, Strahinja Stanojević, Marko Spasić, Vesna Marinković, and Milena Vujošević Janičić. ML-Driven Prediction of Optimal Control Flow Graph Traversal Algorithm in Modern Applications. In *12th International Conference on Electrical, Electronics and Computer Engineering (IcETRAN)*, Čačak, June 2025.

[351] Milan Čugurović, Milena Vujošević Janičić, Vojin Jovanović, and Thomas Wuerthinger. Machine Learning Based Static Profiling, 2024. Patent Application.

Biografija autora

Milan Čugurović rođen je 30. 08. 1995. godine u Loznici. Gimnaziju Vuk Karadžić u Loznici završio je 2014. godine kao nosilac Vukove diplome. Osnovne studije na smeru Računarstvo i informatika na Matematičkom fakultetu završio je jula 2018. godine kao jedini student u generaciji sa prosečnom ocenom 10. Dobitnik je nagrade **„Student generacije"**. Master studije na Matematičkom fakultetu završio je 2019. godine sa prosečnom ocenom 10. Doktorske studije na Matematičkom fakultetu upisao je u oktobru 2019. godine. Sve ispite na doktorskim studijama položio je sa prosečnom ocenom 10.

Od 2018. godine zaposlen je na Matematičkom fakultetu i to u zvanjima demonstrator (februar 2018. godine – oktobar 2018. godine), saradnik u nastavi (oktobar 2018. godine – oktobar 2019. godine) i asistent (od oktobra 2019. godine do danas). Od 2019. godine bio je učesnik projekta Ministarstva prosvete, nauke i tehnološkog razvoja Republike Srbije „Automatsko rezonovanje i istraživanje podataka" pod brojem 174021. U periodu od februara 2020. do jula 2024. godine radio je kao istraživač na projektu saradnje Matematičkog fakulteta i kompanije *Oracle*. Od jula 2024. godine, pored zaposlenja na Matematičkom fakultetu angažovan je i kao istraživač-senior u kompaniji *Oracle*.

Прилог 1.

Изјава о ауторству

Потписани-а Милан Чугуровић

број индекса 2006/2019

Изјављујем

да је докторска дисертација под насловом

Предвиђање профила извршавања програма техникама машинског учења

- резултат сопственог истраживачког рада,
- да предложена дисертација у целини ни у деловима није била предложена за добијање било које дипломе према студијским програмима других високошколских установа,
- да су резултати коректно наведени и
- да нисам кршио/ла ауторска права и користио интелектуалну својину других лица.

Потпис докторанда

У Београду, 30. 09. 2025.

Милан Чугуровић

Прилог 2.

Изјава о истоветности штампане и електронске верзије докторског рада

Име и презиме аутора Милан Чугуровић

Број индекса 2006/2019

Студијски програм Информатика

Наслов рада Предвиђање профила извршавања програма техникама машинског учења

Ментор проф. др Милена Вујошевић Јаничић

Потписани/а Милан Чугуровић

Изјављујем да је штампана верзија мог докторског рада истоветна електронској верзији коју сам предао/ла за објављивање на порталу **Дигиталног репозиторијума Универзитета у Београду.**

Дозвољавам да се објаве моји лични подаци везани за добијање академског звања доктора наука, као што су име и презиме, година и место рођења и датум одбране рада.

Ови лични подаци могу се објавити на мрежним страницама дигиталне библиотеке, у електронском каталогу и у публикацијама Универзитета у Београду.

Потпис докторанда

У Београду, 30. 09. 2025.

Милан Чугуровић

Прилог 3.

Изјава о коришћењу

Овлашћујем Универзитетску библиотеку „Светозар Марковић“ да у Дигитални репозиторијум Универзитета у Београду унесе моју докторску дисертацију под насловом:

Предвиђање профила извршавања програма техникама машинског учења

која је моје ауторско дело.

Дисертацију са свим прилозима предао/ла сам у електронском формату погодном за трајно архивирање.

Моју докторску дисертацију похрањену у Дигитални репозиторијум Универзитета у Београду могу да користе сви који поштују одредбе садржане у одабраном типу лиценце Креативне заједнице (Creative Commons) за коју сам се одлучио/ла.

1. Ауторство

2. Ауторство - некомерцијално

3. Ауторство – некомерцијално – без прераде

4. Ауторство – некомерцијално – делити под истим условима

5. Ауторство – без прераде

6. Ауторство – делити под истим условима

(Молимо да заокружите само једну од шест понуђених лиценци, кратак опис лиценци дат је на полеђини листа).

Потпис докторанда

У Београду, 30. 09. 2025.

Милан Чугуровић

1. Ауторство - Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце, чак и у комерцијалне сврхе. Ово је најслободнија од свих лиценци.

2. Ауторство – некомерцијално. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела.

3. Ауторство - некомерцијално – без прераде. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела. У односу на све остале лиценце, овом лиценцом се ограничава највећи обим права коришћења дела.

4. Ауторство - некомерцијално – делити под истим условима. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца не дозвољава комерцијалну употребу дела и прерада.

5. Ауторство – без прераде. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца дозвољава комерцијалну употребу дела.

6. Ауторство - делити под истим условима. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца дозвољава комерцијалну употребу дела и прерада. Слична је софтверским лиценцама, односно лиценцама отвореног кода.