UDC: 004.85 Info M str. 17-27

ДИНАМИЧКО ПРИКУПЉАЊЕ И СТАТИЧКО ПРЕДВИЂАЊЕ ПРОФИЛА ИЗВРШАВАЊА ПРОГРАМА У КОНТЕКСТУ КОМПАЈЛЕРСКИХ ОПТИМИЗАЦИЈА
DYNAMIC PROFILING AND STATIC PREDICTION OF THE PROGRAM EXECUTION PROFILES IN THE CONTEXT OF COMPILER OPTIMIZATIONS

Милан Чугуровић, Математички факултет, Универзитет у Београду

РЕЗИМЕ: Оптимизације вођене профилима значајно доприносе генерисању ефикасних програма. За њихову имплементацију нужно је присуство профила извршавања програма. У овом раду анализирају се два начина за добијање профила извршавања програма: динамичко прикупљање и статичко предвиђање профила. Основне технике за динамичко прикупљање профила обухватају профајлирање инструментацијом и профајлирање узорковањем. Рад дискутује предности и мане ова два типа профајлирања као и могуће компромисе. Рад детаљно анализира технике статичког предвиђања профила извршавања програма, као знатно јефтиније алтернативе прикупљању профила. У оквиру статичког предвиђања, издвајају се статичке хеуристике и напредне технике машинског учења. Рад истиче разлику између класификационих и регресионих модела, као и изазове са којима се истраживачи сусрећу приликом дефинисања скупова атрибута којима се описују делови кода. Прегледом релевантне литературе, овај рад одговара на питање у којим ситуацијама је погодно користити који приступ за генерисање профила извршавања програма и приказује правце даљег развоја области статичког предвиђања профила.

КЉУЧНЕ РЕЧИ: профајлирање, статичко предвиђање профила, оптимизације вођене профилима, машинско учење, класификација, регресија

ABSTRACT: Profile-guided optimizations significantly contribute to generating efficient programs. Their implementation requires the presence of program execution profiles. This paper analyzes two ways of obtaining program execution profiles: dynamic profiling and static profile prediction. Basic techniques for dynamic profiling include instrumentation profiling and sample-based profiling. The paper discusses the advantages and disadvantages of these two types of profiling and possible compromises. The paper analyzes in detail techniques for static profile prediction as a considerably cheaper alternative to profiling. Static approaches include static heuristics and advanced machine-learning techniques. The paper highlights the difference between classification and regression models, as well as the challenges that researchers face when defining sets of features that characterize code segments. By reviewing relevant literature, this paper answers the question when it is appropriate to use which approach for generating program execution profiles. The paper discusses opportunities for further improvement of static profile prediction techniques.

KEY WORDS: profiling, static profile prediction, profile-guided optimizations, machine learning, classification, regression

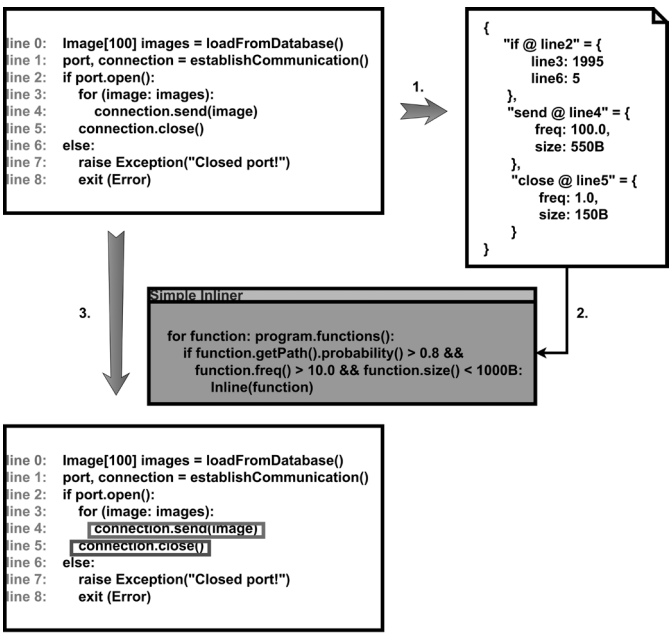
1. УВОД

Перформансе програма директно зависе од оптимизација које компајлер који тај програм преводи имплементира. Стога су оптимизације које један компајлер подржава веома значајне. Једна од најважнијих класа оптимизација су оптимизације вођене профилилма (ОВП, енг. *profile-guided optimizations*) [1]. Профили извршавања програма садрже различите врсте информација о извршавању програма укључујући број позива функција, број избора извршавања сваког понуђеног блока код наредбе гранања (односно број извршавања сваке гране), број извршавања блокова кода, број позива подтипова виртуелних позива, број позива монитора, итд. Знајући који делови кода ће бити извршени колико пута, агресивне компајлерске оптимизације као што су уметање (енг. *inlining*) [2, 3], дупликација кода [4], позиционирање кода [5], оптимизације кеша [6], оптимизације петљи [7], и анализа делимичног изласка (енг. *partial escape analysis*) [8] могу значајно побољшати перформансе преведених програма. Тиме ОВП надмашују домет статичких оптимизација које немају на располагању информације о понашању програма приликом његовог покретања [9, 10].

Да би оптимизације вођене профилима дале добре резултате, профили извршавања програма морају бити *квалитетни*. Квалитетне профиле карактерише прецизност и висока покривеност кода (информације о извршавању већег дела кода). То значи да они егзактно осликавају понашање програма приликом извршавања. На пример, уколико је доступна информација да ће корисници често извршавати одређене путање у програму, компајлер може уметнути позиве функција са те путање, тиме елиминисати трошкове ових позива и побољшати перформансе програма. Међутим, уколико су компајлеру доступни неквалитетни профили, они могу наводити ОВП на погрешне одлуке које резултују смањењем перформанси програма. На пример, у случају неквалитетних профила, ОВП ће уметнути функције на путањама које нису фреквентне и тиме потрошити буџет за уметање на погрешан начин, онемогућавајући уметање често позиваних функција. Важан пример су и оптимизације кеша: непрецизни профили могу водити до кеширања објеката са путања које неће бити извршене и, због ограничености кеша, изостанка кеширања фреквентних објеката. Ово узрокује промашаје у кешу, што доводи до смањења перформанси.

Како ОВП могу значајно побољшати перформансе програма, много пажње посвећено је прикупљању квалитетних профила [11]. *Профајлирање програма* (енг. *program profiling*) представља вид динамичке анализе програма којом се прикупљају подаци о извршавању програма. Стандардни начин профајлирања програма је профајлирање инструментацијом [12]. Иако изучавано и унапређивано годинама, профајлирање инструментацијом и даље има неке нежељене особине. Оно захтева креирање инструментализоване верзије програма, њено покретање, прикупљање профила, и тек затим креирање оптимизоване верзије програма. Све ово компликује процес компилације оптерећујући како програмере додатним послом, тако и машине на којима се код компајлира, обзиром да је инструментација програма обично временски и просторно захтевна. Ублажавање наведених нежељених особина нуде профајлери засновани на узорковању, нажалост често на уштрб одређеног смањења квалитета самих профила. Због свега наведеног, и даље постоје неки велики пројекти који не користе ОВП у продукцији [13].

Профили извршавања програма користе се да би се прилагодиле оптимизације компајлера и генерисали извршни програми у складу са жељеним карактеристикама. На пример, слика 1 илуструје уметање (енг. *inlining*) које користи профиле извршавања програма да генерише ефикаснији код. Уметање је компајлерска оптимизација која замењује позиве функција са кодом тих функција (односно интерном репрезентацијом функција уколико се оптимизација примењује над интерном репрезентацијом кода). На тај начин се елиминишу трошкови позива функција приликом извршавања програма чиме се добија ефикаснији код. Уметање функција повећава величину извршних програма. Због тога се не умећу све функције, већ само оне за које се процени да је то исплативо, односно да ће њихово уметање значајније допринети побољшању перформанси програма.



Слика 1: Илустрација уметања на основу профила извршавања програма

На слици 1 уметање је илустровано на примеру кода који учитава 100 слика из базе података, успоставља везу са сервером, и, у случају да је слике могуће послати на сервер исте шаље на сервер, док, у случају да слике није могуће послати на сервер моментално пријављује грешку и прекида извршавање програма. У првом кораку компилације извршава се програм и прикупљају се профили извршавања програма. У случају да се тест програм изврши 2000 пута, и да због заузетости ресурса на рачунару у 5 случајева дође до грешке а у 1995 случајева се слике успешно пошаљу на сервер, на основу профила извршавања наредбе гранања са линије 2 рачунају се фреквенције путање на којој се налазе функција за слање слика (енг. *sent*) и прекид конекције (енг. *close*). Фреквенција путање на којој се налазе ове функције је 0.9975 (=1995/2000). На слици 1 илустровани су и профили извршавања функција *sent* и *close* који чувају информације о величини функција и профајлираној фреквенцији извршавања функција (функција *close* има профајлирану фреквенцију извршавања 1.0 јер се изврши сваки пут када програм изврши тачну грану наредбе гранања, док функција *sent* има профајлирану фреквенцију извршавања 100.0 јер се изврши 100 пута када програм изврши тачну грану наредбе гранања).

На слици 1 приказан је поједностављен алгоритам уметања (енг. *simple inliner*) који умеће оне функције које се налазе на путањи која ће бити извршена у бар 80% случајева извршавања програма, чија је фреквенција извршавања већа од 10.0 и чија је величина мања од 1000 бајтова. Стога ће овакво уметање (под претпоставком да су функције *close* и *sent* довољно мале, нпр. 150 и 550 бајтова) уметнути функцију *sent*, док фунцкија *close* неће бити уметнута обзиром да је њена профилисана фреквенција мања од фреквенције коју захтева алгоритам. Важно је напоменути да се у пракси користе доста сложенији алгоритми уметања. У случају да се приликом компилације програма не прикупе профили извршавања програма, алгоритам уметања неће моћи да одлучује појединачно о уметању функција *sent* и *close* већ ће морати да се ослони на хеуристике. Овако оптимизован програм имаће лошије перформансе од програма оптимизованог коришћењем информација о извршавању програма јер хеуристике не могу прецизно да обухвате и опишу све случајеве употребе програма.

Статичко предвиђање профила је алтернатива динамичком профајлирању [14]. Наиме, још деведесетих година прошлог века примећено је да се на основу самог кода програма могу извести корисни закључци који се тичу његовог извршавања [15]. На пример, често је случај да се при извршавању петљи повратна грана извршава много већи број пута него грана која води ван петље. На основу таквих закључака дефинисани су скупови хеуристика који су коришћени за навођење ОВП. Иако је списак коришћених хеуристика био релативно мали, показало се да оне, у комбинацији са ОВП, воде побољшању перформанси програма [16]. Основни недостатак ових хеуристика је могућност описивања само мањег скупа једноставнијих правила, односно на овај начин није могуће ухватити комплексне зависности које постоје у коду. На примеру са слике 1 илустровани су профили извршавања програма који се прикупљају динамичким профај-

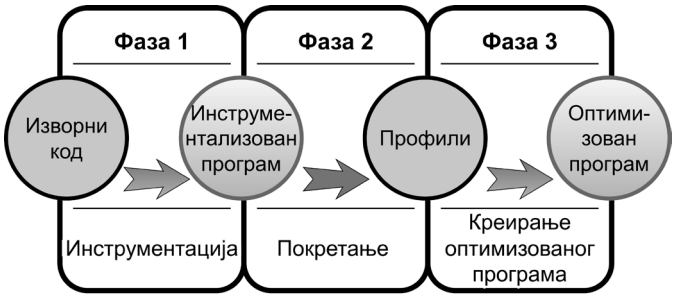
лирањем. Исте профиле могуће је статички предвидети. У том случају они се користе на исти начин као и динамички прикупљени профили.

Способност откривања најфинијих законитости које важе у подацима учиниле су да се модели машинског учења (МЛ модели) природно наметну као проширење статичких хеуристика [17]. Модели машинског учења користе се као алати за откривање најкомплекснијих веза које крије сама репрезентација изворног кода. Они омогућавају ОВП да искористе информације о извршавању програма до којих је могуће доћи без покретања програма. Први радови који су комбиновали моделе машинског учења и статичко предвиђање профила односили су се на предвиђање вероватноћа извршавања путања у коду [14, 15, 18]. Коришћени су класификатори попут стабала одлучивања (енг. *decision tree*) [19, 20], логистичке регресије (енг. *logistic regression*) [21, 22], и потпуно-повезаних дубоких неуронских мрежа (енг. *fully-connected feed-forward deep neural network*) [23, 24, 18]. Развојем области укључивани су нови класификациони модели машинског учења, попут методе κ најближих суседа (енг. *the k-nearest neighbors algorithm*) [25, 26] и рекурентних неуронских мрежа (енг. *recurrent neural network*) [27, 28, 29] и решавани други проблеми попут предвиђања фактора размотавања у петљи (енг. *unroll factor*) [26]. Мана класификације јесте чињеница да се путање у коду класификују обично у две категорије, док профили извршавања представљају континуалне вредности. Ово доводи до губитка информација. Стога се у новије време појављују радови који користе регресију у контексту предвиђања вероватноћа извршавања грана [30]. Што се тиче репрезентације програма из које се издвајају атрибути који описују делове кода, она варира од извршног кода [30, 10] па све до програмске графовске међурепрезентације [29].

У овом раду биће дискутовано прикупљање квалитетних профила извршавања програма. У секцији 2 описано је профајлирање, његове варијације, и проблеме које оно узрокује. У секцијама 3 и 4 описано је статичко предвиђање профила, алтернатива профајлирању која даје профиле високог квалитета, упоредиве са профилима добијеним профајлирањем. У овим секцијама анализирани су домети статичких хеуристика и техника за статичко предвиђање профила извршавања програма коришћењем машинског учења. У секцији 3 главни фокус је на статичкој предикцији грана док се у секцији 4 дискутују напреднији концепти, изазови и модерни аспекти статичке предикције профила извршавања програма. У секцији 5 сублимирани су закључци добијени компаративном анализом начина прикупљања профила извршавања програма и дате су препоруке на тему даљих праваца истраживања у области профајлирања и статичке предикције профила.

2. САКУПЉАЊЕ ПРОФИЛА

Стандардни начин прикупљања профила извршавања програма подразумева употребу профајлера заснованих на инструментацији (енг. *instrumentation-based profilers*), што је илустровано на слици 2. У првој фази компилације потребно је од изворног кода (програм на улазном програмском језику) креирати инструментализован програм. На овом месту програм се проширује кодом способним за колекцију и чување информација о извршавању. Фаза 2 подразумева покретање инструментализованог програма и сакупљање профила. На пример, приликом инструментације може се применити Кнутов алгоритам профајлирања грана [31] који за граф контроле тока у програму креира минимално разапињуће стабло и гранама које не припадају добијеном стаблу додаје бројач посета. Затим се увећавају бројачи сваки пут када се извршавањем инструментализованог програма уђе у неку од грана, чиме се сакупљају профили извршавања грана (за гране које не садрже бројаче, број посета се рачуна на основу бројача осталих грана). Сакупљене профиле након тога компајлер користи да примени ОВП и креира оптимизован програм (фаза 3).



Слика 2: Прикупљање профила извршавања инструментацијом програма

Поред повећавања комплексности процеса изградње софтвера, профајлирање засновано на инструментацији има још недостатка. Процес инструментације и покретања инструментализованог програма, у зависности од самог програма који се инструментализује може трајати знатно дуже и трошити много више меморије него покретање одговарајућег неинструментализованог програма (од 20% [32] до чак 105% [33]). Отуда инструментација и покретање инструментализованих програма некада чак није ни могућа. На пример у случају система са уграђеним рачунарима (енг. *embedded systems*) где је меморијски простор једно од најважнијих ограничења, уколико се инструментацијом премаше расположиви меморијски оквири, покретање инструментализованог програма постаје недоступно [34]. Слично, успорење које намеће инструментација може да угрози процес исправног извршавања код система за рад у реалном времену па и у таквим случајевима није могуће вршити профајлирање на овај начин.

Уколико се у обзир узме контекст модерног развоја софтвера који подразумева непрекидну интеграцију (енг. *continuous integration*) [35] временска и просторна захтевност инструментационих профајлера постаје још израженија. Наиме, непрекидна интеграција захтева стално интегрисање кода, покретање тестова, и креирање стабилне верзије апликације које се одвија и до по неколико пута на дневном нивоу. Како креирање стабилне и оптимизоване верзије апликације укључује инструментацију и покре-

тање инструментализованог кода са циљем прикупљања профила, потрошња ресурса још више добија на значају. Ово посебно долази до изражаја у случају великих апликација које се извршавају дуже време (нпр. центри за обраду података, енг. *data centers*) [10].

Дакле, профајлери засновани на инструментацији прикупљају квалитетне профиле извршавања програма по цену повећања комплексности инфраструктуре и потрошње веће количине ресурса. Алтернатива инструментационим профајлерима јесу профајлери засновани на узорковању (енг. *sample-based profilers*) који смањују трошкове инструментације на уштрб квалитета профила. Профајлери засновани на узорковању наизменично смењују покретање инструментализованог и неинструментализованог кода, чиме смањују потрошњу ресурса. На пример, *инстант профајлирање* (енг. *instant profiling*) [36] показало се као добар приступ инструментације великих апликација (центри за обраду података) успешно смањујући успорење извршавања приликом прикупљања профила на само 6%. Коришћењем профајлирања узорковањем успешно је креиран портабилан и ефикасан профајлер за Јава виртуелне машине који при континуираној продукционој употреби уноси успорење између 2% и 4% [37]. Сенка профајлирање (енг. *shadow profiling*) [38] комбинује узорковање делова инструментализованог кода паралелно са извршавањем програма и, ослањајући се на све већи број процесора на рачунарима, коришћењем паралелизације смањује трошкове прикупљања профила.

Уколико процесор подржава хардверске бројаче намењене профајлирању, они се могу искористити за прикупљање профила чиме се додатно смањују трошкови профајалирања. Најефикаснији тип профајлера су хардверски засновани профајлери (енг. *hardware-based profiling*) који користе хардверску подршку да генеришу профиле током извршавања самог програма. На овај начин превазилази се потреба за креирањем и посебним покретањем инструментализоване верзије програма зарад прикупљања профила. Овај приступ способан је да прикупи прецизне профиле уносећи минимално успорење компилације од 0.4 до 4.6% [39]. *Гуглово широко профајлирање* (енг. *Google wide-profiling*) представља инфраструктуру за профајлирање центара за обраду података и апликација у облаку (енг. *cloud applications*) засновану на профајлирању узорковањем. Са минималним утицајем на перформансе представља скалабилан алат који обезбеђује стабилне и прецизне профиле [40]. *LLVM* имплементација профајлирања узорковањем која користи хардверске бројаче да прикупи профиле извршавања програма показала се способном да сакупи квалитетне профиле [9]. На већим апликацијама попут обраде слика/видеа, серверу за веб претраге, и процесирању логова, ОВП на овако прикупљеним профилима воде до побољшања перформанси од 30%.

Коришћење хардверских бројача интегрисаних у процесоре подразумева прикупљање профила на нивоу извршне верзије програма. Отуда је у овом случају доминантан проблем за добијање прецизних профила пресликавање профила из извршног кода у репрезентацију над којом се могу користити ОВП. Профили се могу пресликати назад у различите делове ланца компилације: у време компилације (енг. compile time) [41], линковања (енг. link time) [42, 43], или тек након линковања (енг. post-link time) [10, 44]. На овом месту потребно је направити компромис између прецизности профила и броја ОВП фаза које их могу искористити: што раније профили постану доступни то већи број ОВП фаза може да их користи, али су са друге стране мање прецизни од профила који се пресликавају у касније делове ланца компилације, ближе извршном коду [10].

Поред свега овога, још један проблем који се јавља независно од типа профајлирања јесте проналажење довољно информативног скупа тест примера који ће служити за генерисање квалитетних профила. Ово посебно долази до изражаја у случају великих апликација, јер број могућих случајева употребе расте експоненцијално са порастом величине програма. Овај проблем се не може решити променом типа профајлера, већ је за његово решавање потребно променити приступ проблему, и у потпуности избећи профајлирање. Решење нуди статичко предвиђање профила.

3. СТАТИЧКО ПРЕДВИЂАЊЕ ПРОФИЛА

Алтернатива профајлирању јесте статичко предвиђање профила. Статичко предвиђање профила се, апстрахујући детаље, састоји само од једне фазе у којој се, коришћењем статичког предиктора генеришу профили извршавања делова кода које ОВП користе за креирање оптимизованог програма (слика 3). Статички предиктор своја предвиђања даје без покретања програма, предвиђајући профиле на основу самог програма (односно репрезентације коју користи, нпр. изворног кода, компајлерске међурепрезентације, или извршног кода[1]). Тиме се анулирају главни недостаци профајлера: комплексна инфраструктура и велика потрошња ресурса (процесора и радне меморије).

Слика 3: Оптимизација програма коришћењем статичког предиктора профила

Фишер и Фројденбергер су међу првима радили на статичком предвиђању контроле тока програма **[14]**. Они су на основу изучавања профила извршавања програма радили на статичком предвиђању грана које ће бити извршене након

1 Под извршним кодом подразумева се и бајткод, у случају језика који се извршавају на виртуелној машини.

наредби гранања (енг. *profile-based static branch prediction*, у наставку текста скраћено предвиђање грана). Показали су да, чак и у комплексним апликацијама попут језичких процесора (енг. *language processors*) и системских библиотека (енг. *system utilities*), након већине наредби гранања у коду извршава се само једна од понуђених грана. Као интуиција може послужити пример обраде изузетка у коду, што је грана за коју очекујемо да ће ретко бити посећена (само у изузетним случајевима, на пример када није могуће алоцирати ресурсе услед недостатка меморије на рачунару). Аутори показују да се, на основу ранијих покретања других програма, могу предвидети гране програма које ће бити извршене. Каснији радови у овој области баве се начинима на које је могуће статички (без покретања програма) предвидети које гране ће бити извршене при покретању програма.

3.1. *Хеуристике*

Најједноставнија врста статичких предиктора су хеуристике. Оне се могу посматрати као скуп правила којима се, на основу карактеристика самог програма који се преводи, усмерава компилација и генерише оптимизован програм. Бал и Ларус су развили статички предиктор заснован на крактеристикама програма (енг. *program-based branch predictor*) чији су квалитет доказали на већем броју различитих *C* и Фортран апликација [15]. Они дефинишу скуп ефикасних хеуристика које статички предвиђају извршавање грана у коду. Дефинисане хеуристике своја предвиђања дају само на основу извршног фајла (енг. *executable file*), и стога су јако једноставне за имплементацију. Све хеуристике мотивисане су интуицијом и посматрањем начина на које се програми извршавају. На пример, хеуристика која се заснива на операционом коду инструкције (енг. *opcode heuristic*) уско је везана за процесор МИПС P2000 (енг. *MIPS R2000*) који подржава инструкције гранања на основу поређења вредности регистра са 0. Како многи програми користе негативне вредности да идентификују стање грешке, то хеуристика предвиђа да гране које одговарају успешним поређењима регистра са ≤0, односно <0 неће бити извршене, док предвиђа извршавање грана које одговарају успешним поређењима регистра са ≥0, односно >0. Хеуристика поређења показивача (енг. *pointer comparison heuristic*) предвиђа извршавање грана које зависе од тачног поређења два показивача, док предвиђа прескакање извршавања грана које зависе од тачне вредности поређења показивача са вредношћу нул (енг. *null*), јер се очекује да поређење показивача са вредношћу нул у великом броју случајева одговара провери изузетака. Хеуристика петљи (енг. *loop heuristic*) углавном предвиђа извршавање петљи, пре него њихово прескакање, док хеуристика позива (енг. *call heuristic*) углавном предвиђа прескакање позива функција. Ово је објашњено великим бројем позива функција које обрађују изузетке, и стога се ретко извршавају.

У случају већег броја хеуристика, као што је то и у раду Бала и Ларуса, потребно је дефинисати приоритете односно начин комбиновања хеуристика. Бал и Ларус су имплементирали најједноставнију варијанту: сортирање хеуристика, и за сваку наредбу гранања пролазак кроз сортиране хеуристике редом и употреба прве применљиве хеуристике. Критеријум по коме су сортиране хеуристике, у овом случају одређен је експериментално.

Грешка класификације пријављена у раду Бала и Ларуса је 26% (грешка класификације грана на оне које ће бити извршене и оне које неће). У овом раду за евалуацију је коришћен скуп од 23 Јуникс (енг. *Unix*) програма попут компајлера *gcc* и *lcc,* алата *compress, grep, dcg,* итд.

Рад Бала и Ларуса омогућио је значајна побољшања перформанси програма. Због тога су настављна истраживања у том смеру и рађено је на даљем унапређењу и побољшању ових хеуристика. Демпстер-Шаферова теорија доказа [45] превазилази проблем комбиновања више хеуристика тако што, за сваку наредбу контроле тока агрегира предвиђања свих применљивих хеуристика. Тиме даје предвиђања вероватноћа извршавања грана, чиме се избегава губљење информације које класификација нужно уноси. На основу статичких предвиђања вероватноћа извршавања грана, Ву и Ларус [46] рачунају вероватноће извршавања блокова као и фреквенције позива функција. На овај начин проширују домен статичког предиктора са предвиђања грана на напредније концепте. Ву и Ларус за евалуацију користе бенчмарке SPEC92 [47] као и стандардне Јуникс програме *cal*, *cmp*, *ed*, *grep*, *factor*, *pack*, *split*, *sum*, *tsort*, *uniq*, и *wc*. Резултати евалуације приказани су у Табели 1.

Профили	**SPEC92**	**Јуникс команде**
Блокови	77%	78%
Гране	78%	79%
Позиви фуникција	85%	82%

Табела 1: Просечна тачност статичког профајлера Ву и Ларуса

Мана коришћења хеуристика као статичких предиктора јесте чињеница да је хеуристике потребно ручно дефинисати. Стога је на тај начин могуће описати само део зависности које постоје између изворног кода (или извршног кода) програма и начина на који се он извршава. Ово потврђује чињеница да побољшање постојећих и додавање нових хеуристика доводи до директног побољшавања перформанси. У свом раду на компајлеру ИМПАКТ (енг. *IMPACT compiler*) аутори профињују простојеће хеуристике, претходно дефинисане у раду Бала и Ларуса фокусирањем на баферовање (енг. *I-O buffering*), обраду грешке (енг. *error processing*), позивање функција исписа (енг. *print function*), меморијске алокације (енг. *memory allocation*), итд. [16]. На пример, хеуристику која се односи на показиваче, не примењују на показивачке променљиве које су елементи низова, јер је за њих примећено да имају већу вероватноћу да буду недефинисане (у поређењу са стандардним показивачким променљивим). Рад додаје и две нове хеуристике: хеуристику поређења карактера (енг. *character comparison heuristic*) и хеуристику поређења битова (енг. *restricted opcode AND heuristic*). На пример, хеуристика поређења карактера предвиђа да грана која

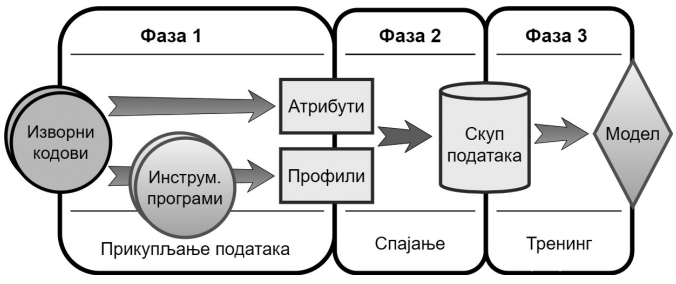
одговара једначењу промењиве са карактером неће бити извршена (ова хеуристика мотивисана је чињеницом да и карактер размак, као најфреквентнији карактер енглеског алфабета, има фреквенцију мању од 0.5). Још једно важно побољшање дао је Вонг [48], преласком на хеуристике које закључују на основу изворног кода улазног програма, а не на основу одговарајућег бинарног кода. Ово је мотивисано чињеницом да изворни код садржи већу количину информација о програму у поређењу са извршним кодом. Важан допринос овог рада јесте коришћење имена макроа, функција и промењљивих у хеуристикама. Ово има природну мотивацију, нпр. очекујемо да се функције за обраду грешке ретко позивају и очекујемо да су оне често слично именоване, што се може искористити приликом предвиђања профила извршавања.

Додавање нових и прецизнијих хеуристика има своју границу, те је за достизање максималних перформанси потребно променити приступ проблему. На овом месту, решење нуде технике машинског учења.

3.2. *Модели машинског учења*

како омогућавају проналазак и најскривенијих зависности које важе у подацима [17], МЛ модели показали су се као најбољи статички предиктори. На основу покретања већег броја програма и прикупљања информација о њиховом извршавању, могуће је обучити МЛ модел који ће предвиђати профиле програма без њиховог извршавања.

За обучавање МЛ модела потребно је креирати довољно велики и информативан скуп података. Схема тренирања МЛ модела дата је на слици 4. Пре свега, потребно је обезбедити довољно велики број улазних програма. У фази 1, из изворног кода се издвајају атрибути који описују делове кода, заједно са њиховим профилима извршавања. Добијање профила извршавања може се одрадити произвољним начином профајлирања. Тиме се након фазе 1, за сваки део кода издвајају атрибути који га описују и профили добијени његовим извршавањем. Спајањем атрибута и профила добија се скуп података (енг. *dataset*) за тренирање МЛ модела (фаза 2). Последњи корак јесте обучавање МЛ модела на креираном скупу података (фаза 3).



Слика 4: Илустрација процеса тренирања МЛ модела

Процедура креирања скупа података и обучавања МЛ модела се извршава само једном. Након тога, сама примена модела своди се на један корак предвиђања, као и примена било ког статичког предиктора (слика 3).

Калдер и остали дефинишу ЕСП (енг. *evidence-based static branch prediction, ESP*) [18], који, на основу понашања постојећих програма предвиђа понашање нових програма уз употребу техника надгледаног машинског учења. ЕСП користи дубоке неуронске мреже [23, 24] и стабла одлучивања [19] да преслика статички издвојене атрибуте грана у предвиђања да ли ће оне бити извршене или не. ЕСП показује значајне предности у односу на раније статичке предикторе засноване на хеуристикама: ЕСП је приступ заснован на моделима машинског учења који уче на основу атрибута издвојених из извршних фајлова, па је стога применљив на већи број компајлера, програмских језика и архитектура. Ово је последица чињенице да се ЕСП не заснива на ручно дефинисаним хеуристикама, које су тешке за прилагођавање новим архитектурама. Свака наредба гранања описана је атрибутима који се односе на инструкцију гранања (операциони код инструкције која је довела до гранања, регистри које та инструкција користи, итд.), атрибутима „тачне“ гране (енг. *features of the taken successor*) и атрибутима „нетачне“ гране (енг. *features of the non-taken successor*). Атрибути којима се описују тачна и нетачна грана фокусирају се на информације о доминаторима, пост-доминаторима, петљама, излазима из програма и слично. Прецизност класификације ЕСП модела је 80%, што је напредак од 5% у односу на до тада најбољи статички предиктор заснован на хеуристикама.

Мана ЕСП-а јесте дефинисање атрибута којима се описују наредбе гранања на нивоу извршног програма, чиме је скуп атрибута зависан од архитеткуре на којој се програм извршава. Бус и Вајмер [22] то превазилазе дефинишући атрибуте на нивоу изворног кода на вишем програмском језику (енг. *source code*), чиме се апстрахују појединости сваке појединачне архитектуре. Аутори користе логистичку регресију [21] да идентификују често извршаване путање у коду (енг. *hot paths*). Модел логистичке регресије успешно идентификује првих 5% путања које чине више од половине укупног времена извршавања програма.

Стабло одлучивања успешно оптимизује редослед примена хеуристика дефинисаних у раду Бала и Ларуса. Додатно, модел стабла одлучивања успешно је искоришћен за откривање две нове хеуристике статичког предвиђања профила: једне засноване на релацији постдоминације између базних блокова и друге засноване на растојању (у смислу броја инструкција) између гране и инструкције која је узрокује [20].

Статички предиктор *CrystallBall* [29] користи технике дубоког учења да предвиди извршене путање у програму. Рекурентна неуронска мрежа, конкретно дуга краткорочна меморија (енг. *long short-term memory*) [49] способна је за проналажење комплексних веза између секвенци улаза. У раду у којем је описан предиктор *CrystallBall* она је искоришћена за предвиђање секвенци базних блокова приликом извршавања програма. Додатно, у односу на раније дискутоване радове, предност система *CrystallBall* јесте чињеница да користи атрибуте издвојене из компајлерске међурепрезентације (енг. *intermediate representation*),

а не на нивоу извршног кода или изворног кода: тиме је постигнута потпуна независност од архитектуре на којој се апликација извршава као и од језика у коме је апликација написана. У раду аутори као оцену квалитета класификационог модела користе површину испод РОЦ криве (енг. *area under the receiver operating characteristic curve*) [50]. Мотивација за избор ове метрике лежи у чињеници да на овај начин аутори покушавају да постигну реалистичну оцену у контексту небалансираних класа (извршених односно неизвршених путања). У случају предвиђања извршених путања у коду, само мали број путања ће бити извршен при покретању програма, отуда постоји велика небалансираност класа и стандардне метрике попут тачности (енг. a*ccuracy)*, прецизности (енг. *precision*), или одзива (енг. *recall*) могу се врло лако максимизовати што води ка нереалистичној оцени квалитета модела.

Ротем и Куминс [51] тренирају ансамбл стабала одлучивања за вишекласну класификацију (енг. *multiclass classification*) којим предвиђају вероватноћу извршавања грана. Приликом креирања скупа података за тренинг аутори профајлирају наредбе гранања са две гране и прикупљају информације о односу вероватноћа извршавања леве и десне гране (енг. *ratio*). За сваку наредбу гранања са две гране, аутори однос између вероватноћа извршавања грана класификују у једну од 11 класа: 0.0, 0.1, 0.2, ..., 1.0. Тиме креирају тренинг скуп за вишекласну класификацију на коме обучавају ансамбл стабала одлучивања. Приликом обучавања модела, као функцију грешке аутори користе унакрсну ентропију (енг. *logloss*) [52]. Како се ради о вишекласној класификацији, ентропија која се минимизује рачуна се на основу излаза из фунцкије меког максимума (енg. *softmax*). Као начин агрегације стабала одлучивања аутори користе агрегирање градијентним појачавањем (енг. *XGBoost*) [53]. У 75% случајева модел предвиђа тачну класу (тачан однос између вероватноћа извршавања леве и десне гране наредбе гранања). Интеграција претренираног ансмабла у компајлер *LLVM* [54] дала је побољшање перформанси на 6 од 10 тест програма. Највеће побољшање од 16% остварено је на програмском језику Пајтон. Погоршање перформанси од 7% остварено је на тест програму *bzip2* [55]. Скуп атрибута којима се описују наредбе гранања односно њене гране састоји се од 54 атрибута издвојених на основу бинарне репрезентације програма: број позива функција, број позива наредби за учитавање и упис података (енг. *load and store instructions*), број наредби којима се прекида програм (енг. *return statements*), број блокова које грана доминира, информације о доминаторима у односу на леву и десну грану, итд.

Статички профајлер ВЕСПА (енг. *VESPA static profiler*) [30] користи дубоку неуронску мрежу за регресију као статички предиктор који предвиђа вероватноће извршавања грана (у наредбама гранања са две гране). Атрибуте којима се описују наредбе гранања ВЕСПА дефинише на нивоу извршних фајлова (конкретно бинарних фајлова, енг. *binary-level features*). ВЕСПА генерише листу од 56 атрибута који описују наредбе гранања (део атрибута преузет је из литературе, а део атрибута представљају нови атрибути, предложени у овом раду). Након тога, техником рекурзивне елиминације атрубута аутори бирају 26 најинформативнијих атрибута (16 атрибута презентованих у Калдеровом раду [18] и 9 нових атрибута) које користе за обучавање дубоке неуронске мреже. Бинарни оптимизатор БОЛТ (енг. *BOLT*) [10] коришћењем статичког предиктора ВЕСПА производи извршне фајлове, у просеку 5.47% брже него извршне фајлове генерисане без статичког предвиђања вероватноћа грана. Као скуп тест програма, аутори користе четири програма, компајлере *Clang* [56] и *GCC* [57], као и системе база података *MySQL* [13] и *PostgreSQL* [58].

4. ИЗАЗОВИ МАШИНСКОГ УЧЕЊА У КОНТЕКСТУ СТАТИЧКОГ ПРЕДВИЂАЊА ПРОФИЛА

Поред класификације грана и предвиђања вероватноћа извршавања грана, МЛ модели се могу користити и за статичко предвиђање других својстава програма. И овде значајну улогу има и дефинисање скупа атрибута којима се описују делови кода.

4.1. *Напредне употребе*

Модели машинског учења се могу обучити за предвиђање фактора одмотавања петљи, идентификацију фреквентних метода, одређивање редоследа оптимизација у компајлеру, оптимизацију фазе уметања функција и друга статичка предвиђања профила програма.

Метод *к* најближих суседа [25] успешно се користи као вишекласни класификатор који предвиђа фактор одмотавања петљи (енг. *unroll factor*) [26]. У овом раду аутори креирају тренинг скуп који се састоји од више од 1100 петљи прикупљених из 46 тест програма. На њему тренирају модел *к* најближих суседа који предвиђа да ли је, и који фактор одмотавања петље исплативо применити. На петљама из тест скупа, са прецизношћу од 88% модел предвиђа да ли је петљу потребно одмотати или не. Са тачношћу од 74% модел предвиђа и фактор одмотавања петље. Интегрисањем претренираног модела у *Open Research Compiler* [59], аутори остварују убрзање времена извршавања апликација од, у просеку, 6%. За евалуацију статичког предиктора коришћени су програми *SPEC* [60, 61].

Метод потпорних вектора (енг. *support vector machine*) [62] са тачношћу од 62.57% може идентификовати и предвидети фреквентне методе [63]. За креирање скупа података за тренинг аутори користе *C* програме које конвертују у *LLVM* међурепрезентацију. За обучавање класификационог модела потпорних вектора аутори за сваку од метода из *LLVM* међурепрезентације издвајају 10 статичких атрибута: број петљи у методи, просечну дубину петљи, број спољашљих петљи, број инструкција, број позива других метода у оквиру методе, број инструкција учитавања и чувања у оквиру методе (енг. *load and store instructions*), број наредби гранања, број базних блокова у методи и број позива (енг. *call sites*) сваког метода. У раду су коришћени тест програ-

ми *SPEC CPU2000 INT* и *UTDSP* [64] и унакрсна евалуација (енг. *leave-one-out cross validation*). Овај рад послужио је као инспирација истраживачима, стога су Махапатра и Патра [65] истраживали утицај избора кернела метода потпорних вектора за класификацију фреквентних метода. Аутори пријављују да најбољу прецизност од 65% остварују коришћењем радијалног кернела (енг. *radial basis function*), док остали кернели воде до знатно слабијих резултата.

Већина објектно оријентисаних програмских језика имплементира механизам виртуелних позива и тако подржава полиморфизам. Информације о фреквенцијама виртуелних позива у објектно оријентисаним језицима омогућавају креирање ефикаснијих прогама [66]. У програмима писаним у овим језицима модели машинског учења могу предвидети фреквенцију виртуелних позива на основу статички издвојених атрибута. Статички предиктор Фестивал (енг. *Festival, Frequency estimation of virtual call targets*) [67] користи дубоку неуронску мрежу да открије везу између релативних фреквенција виртуелних позива и 14 статички издвојених атрибута којима се те методе описују: дубина пакета у ком је дефинисана класа, број класа чије је име слично имену класе, број дефинисаних метода у класи, број атрибута дефинисаних у класи, индикатор да ли је класа апстрактна, индикатор да ли је класа анонимна, модификатор класе (приватна, јавна, заштићена, подразумевана), итд. Приликом креирања тренинг скупа, за сваки виртуелни позив са мање од 10 позива, класа 1 додељује се најфреквентнијем позиву, док се осталим позивима додељује класа 0. У случају виртуелних позива са више од 10 позива, 20% најфреквентнијих позива класификује се у класу 1, док се остали позиви класификују у класу 0. Као модел аутори користе дубоку неуронску мрежу са 14 улаза који одговарају издвојеним атрибутима. Приликом обучавања модела минимизује се средњеквадратна грешка , где представља класу додељену инстанци а предвиђање мреже (реалан број између 0 и 1) за инстанцу (вектор реалних бројева дужине 14). Када се модел обучи, за сваку тест инстанцу предвиђање модела представља фреквенцију виртуелног позива. Евалуација на тест програмима *DaCapo* [68] потврђује да је расподела фреквенција виртуелних позива предвиђена коришћењем статичког предиктора Фестивал много информативнија од униформне расподеле, коју статичка анализа подразумевано користи. Аутори не интегришу статички предикор Фестивал у компајлер да би тестирали његов утицај на перформансе већ исти процењују коришћењем Кендал Тау растојања [69] (мере сличности између предвиђених позива и профајлираних позива).

MILEPOST GCC (енг. *machine learning for embedded programs optimization*) [70, 71] је компајлер који у себи интегрише моделе машинског учења који му омогућавају да аутоматски прилагођава фазе оптимизације и оптимизује време извршавања, смањи величину извршних фајлова и време компилације на различитим програмима и архитектурама. Компајелр *MILEPOST GCC* користи метод κ најближих суседа да симултано одреди оптимални редослед оптимизација и прилагоди параметре тих оптимизација. Истраживачи са универзитета у Мичигену у раду пријављују убрзање од 11% на тест програмима *MiBench* [72].

MLGOPerf [73] представља радни оквир који омогућава унапређење компајлера *LLVM* коришћењем техника машинског учења за оптимизацију процеса уметања функција. Коришћењем учења условљавањем (енг. *reinforcement learning*) [74], аутори остварују убрзање од 1.8% до 2.2% у поређењу са компајлером *LLVM* и нивоом оптимизације *O3* (за евалуацију су коришћени програми *SPEC CPU2006* [55] и *Cbench*).

4.2. *Атрибути за обучавање модела машинског учења*

Дефинисање атрибута који описују делове кода (нпр. наредбе гранања или методе) јесте један од најтежих задатака приликом коришћења класичних метода машинског учења за статичко предвиђање профила извршавања програма (нпр. стабло одлучивања, алгоритам κ најближих суседа, метод потпорних вектора, потпуно повезана дубока неуронска мрежа, итд.). Овај задатак се углавном решава задавањем унапред одређеног скупа атрибута којима се описују делови кода. Ови атрибути се, од случаја до случаја издвајају из изворног кода програма, компајлерске међурепрезентације, или извршног фајла. Репрезентација програма *ProGraML* (енг. *program graphs for machine learning*) [75] је графовска репрезентација програма изведена из компајерске међурепрезентације. Репрезентација *ProGraML* је независна од програмског језика и за циљ има превазилажење проблема дефинисања статичког скупа атрибута који описују делове кода. Репрезентација *ProGraML* истовремено садржи информације о контроли тока у програму (енг. *program control flow*) и току података у програму (енг. *program data flow*). Прилагођавање заграђених графовских неуронских мрежа (енг. *gated graph neural network*) [76] за рад над репрезентацијом *ProGraML* остварује добре резултате (вредност метрике [77, 78], односно хармонијске средине прецизности и одзива класификације је) и надмашује класичне методе машинског учења.

Схих и остали [79] раде на предвиђањеу грана и показују да класични алгоритми машинског учења могу значајно профитирати од проширења скупа атрибута који описују наредбе гранања. Поред тога, показују и да примена графовских неуронских мрежа (енг. *graph neural networks*) [80, 81] које користе репрезентацију *ProGraML* остварује резултате сличне као и класични алгоритми при чему не захтева ручно дефинисање атрибута.

4.3. МОДЕРНО СТАТИЧКО ПРЕДВИЂАЊЕ ПРОФИЛА

У свом раду на компајлеру *GraalVM* [82] Чугуровић и остали обучавају ансамбл стабала одлучивања за статичко предвиђање вероватноћа извршавања грана [83]. Аутори користе стабла одлучивања за регресију која агрегирају коришћењем градијентног појачавања (енг. *XGBoost* [84]). За сваку наредбу гранања са две гране предвиђају вероватноћу извршавања примарне гране. У компајлеру *GraalVM* свака наредба гранања има дефинисану примарну грану чију фреквенцију извршавања аутори користе као лабелу

приликом обучавања модела односно циљну променљиву приликом примене модела (нпр. у случају *If* наредбе, грана која одговара тачном услову наредбе гранања јесте примарна грана, и њена вероватноћа се предвиђа).

Аутори дефинишу скуп атрибута којима се описују наредбе гранања на нивоу интерне графовске међурепрезентације високог нивоа (енг. *high-level graph based intermediate representation*) коју користи компајлер *GraalVM*. Тиме се постиже независност модела од архитектуре и изворног језика (исти скуп атрибута применљив је на све језике односно компајлере који користе графовску међурепрезентацију). Додатно, дефинисање атрибута на нивоу графовске репрезентације и истовремено коришћење стабала одлучивања омогућава интерпретабилност. Аутори у раду дефинишу и нове атрибуте којима описују наредбе гранања: прецизне статичке процене броја циклуса процесора, величине генерисаног асемблерског кода (енг. *assembly size*), као и процена броја јефтиних и скупих инструкција процесора. Тиме се омогућава употреба атрибута ниског нивоа (односно њихових апроксимација) на графовској међурепрезентацији високог нивоа.

Да би осигурали добре перформансе модела и у специјалним случајевима (подацима који одударају од података тренинг скупа), аутори користе две посебно дефинисане хеуристике: хеуристику петљи и хеуристику изласка из програма. Хеуристика петљи осигурава да модел не предвиди фреквентан код тела петље као код који се неће извршити и тиме драстично поквари перформансе програма. Она осигурава да предвиђена вероватноћа тела петље не буде мања од 0.2. Хеуристика изласка из програма осигурава да у ситуацијама када је једна грана довољно велика (процењен генерисани код у асемблеру ≥50) а друга грана садржи излаз из програма - грана која садржи излаз из програма не може имати вероватноћу мању од 0.8. Тиме се осигурава рационално трошење буџета, што има позитиван утицај на перформансе статичког предиктора. Параметре хеуристика аутори су одредили експериментално.

Коришћење ансамбла стабала одлучивања води до модела величине од 250 килобајта меморије који ефикасно предвиђа вероватноће извршавања грана. Ово омогућава интеграцију обученог ансамбла у компајлер *GraalVM* и креирање комплетног решења (енг. *end-to-end solution*). Евалуација на тест програмима из колекција колекција *Renaissance* [85], *DaCapo* [86], и *DaCapo con Scala* [87] показује убрзање од 7.46% у односу на подразумевану премијум верзију компајлера *GraalVM* (енг. *GraalVM Enterprise Edition 23.0*).

5. ЗАКЉУЧАК

Оптимизације вођене профилима једне су од најважнијих компајлерских оптимизација. Оне могу значајно побољшати перформансе програма. Отуда је велика пажња посвећена прикупљању квалитетних профила извршавања програма. У овом раду сумирани су најзначајнији радови који се баве прикупљањем профила. Два основна начина прикупљања профила јесу профајлирање инструментацијом и профајлирање узорковањем. Профајлирање инструментацијом прикупља квалитетније профиле него профајлирање засновано на узорковању уз знатно већу потрошњу ресурса. Апропо прикупљања профила, закључак који се намеће јесте да, у случају када ресурси нису ограничавајући фактор, профајлирање инструментацијом јесте решење које води до најбољих перформанси. Сходно томе, профајлирање инструментацијом би требало бити први избор за имплементацију у случају када су перформансе критичне (енг. *performance critical applications*) а за инструментацију постоје довољни ресурси. Када су ресурси уско грло (енг. *bottleneck*), задовољавајућа алтернатива јесте профајлирање узорковањем. У ситуацији када су ресурси ограничени и није могуће инструментализовати програм нити прикупљати профиле узорковањем, решење јесте статичко предвиђање профила.

Развој области статичког предвиђања профила извршавања програма траје сада већ дуже од тридесет година. У том временском интервалу компајлери и ОВП значајно су се променили. Различити радови користе различите приступе а самим тим и различите оцене квалитета модела, при томе често за евалуацију користе различите бенчмарке. На пример, оцена квалитета класификационих модела дата је у терминима грешке класификације док је оцена регресионих модела изражена у терминима средњеквадратне грешке. Због свега наведеног јако је тешко директно упоредити различите радове области статичког предвиђања профила и дати непристрасну оцену њиховог квалитета.

У овом раду издвојени су резултати најважнијих радова који се баве статичком предикцијом профила. Први радови који су статички предвиђали профиле користили су ручно дефинисане хеуристике да класификују гране на оне које ће бити извршене и оне које неће. МЛ модели су портабилнији од хеуристика и могу моделовати знатно комплексније зависности. Отуда су они де факто постали стандард статичког предвиђања профила. Једноставније варијанте укључују класичне МЛ моделе и ручно дефинисање атрибута, док напредније технике укључују графовске неуронске мреже које избегавају ручно дефинисање атрибута и моделе ненадгледаног учења. У новије време показано је да потпуно повезана дубока неуронска мрежа за регресију може дати боље резултате од класификационих модела статичког предвиђања профила.

Закључак компаративне анализе дискутованих радова области статичког предвиђања профила јесте да је у случају потребе за имплементацијом статичког предиктора профила најбоље користити регресиони МЛ модел. Доказани квалитет дубоких неуронских мрежа у контексту бинарних оптимизатора и ансамбла стабала одлучивања на графовској међурепрезентацији високог нивоа оправдава очекивање да и други регресиони МЛ модели могу остварити боље резултате него одговарајући класификациони МЛ модели. Отуда је за очекивати ће се у будућности радити на даљем развоју регресионих МЛ модела за статичко предвиђање профила. Закључак овог рада јесте и предвиђање да ће даљи развој области статичког предвиђања профила радити на проширивању и генерализацији атрибута који описују делове кода а који се користе за обучавање регресионих МЛ модела, обзиром да ту постоји још простора за напредак.

РЕФЕРЕНЦЕ

[1] R. Gupta, E. Mehofer and Y. Zhang, "Profile guided compiler optimizations," *Citeseer,* 2002.

[2] A. Ayers, R. Schooler и R. Gottlieb, „Aggressive inlining," *ACM SIGPLAN Notices,* т. 32, pp. 134-145, 1997.

[3] P. Chang, S. Mahlke, W. Chen и W.-M. Hwu, „Profile-guided automatic inline expansion for C programs," *Software: Practice and Experience,* т. 22, pp. 349-369, 1992.

[4] D. Leopoldseder, L. Stadler, T. Wurthinger, J. Eisl, D. Simon и H. Mossenbock, „Dominance-based duplication simulation (DBDS): code duplication to enable compiler optimizations," у *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018.

[5] K. Pettis и R. Hansen, „Profile Guided Code Positioning," *SIGPLAN Not.,* т. 25, pp. 16-27, 1990.

[6] S. Saxena, „Profile guided TLB and cache optimization," Google Patents, 1997.

[7] D. Bacon, S. Graham и O. Sharp, „Compiler transformations for high-performance computing," *ACM Computing Surveys (CSUR),* т. 26, pp. 345-420, 1994.

[8] L. Stadler, T. Wurthinger и H. Mossenbock, „Partial escape analysis and scalar replacement for Java," у *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014.

[9] D. Novillo, „SamplePGO - The Power of Profile Guided Optimizations without the Usability Burden," *2014 LLVM Compiler Infrastructure in HPC,* pp. 22-28, 2014.

[10] M. Panchenko, R. Auler, B. Nell и G. Ottoni, „BOLT: A Practical Binary Optimizer for Data Centers and Beyond," arXiv, 2018.

[11] A. L. Porter, A. Kongthon и a. J.-C. Lu., „Research profiling: Improving the literature review.," *Scientometrics,* т. 53, бр. 3, pp. 351-370, 2002.

[12] E. Metz и R. Lencevicius, „Efficient instrumentation for performance profiling.," *arXiv preprint cs/0307058,* 2003.

[13] P. DuBois, MySQL, Addison-Wesley Professional, 2013.

[14] J. Fisher и S. Freudenberger, „Predicting conditional branch directions from previous runs of a program," *ACM SIGPLAN Notices,* т. 27, pp. 85-95, 1992.

[15] T. Ball и J. Larus, „Branch prediction for free," *ACM SIGPLAN Notices,* т. 28, pp. 300-313, 1993.

[16] B. Deitrich, B. C. Chen и W.-m. Hwu, „Improving static branch prediction in a compiler," у *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*, 1998.

[17] M. Jordan и T. Mitchell, „Machine learning: Trends, perspectives, and prospects," *Science,* т. 349, pp. 255-260, 2015.

[18] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer и B. Zorn, „Evidence-based static branch prediction using machine learning," *ACM Transactions on Programming Languages and Systems (TOPLAS),* т. 19, pp. 188-222, 1997.

[19] R. Safavian и D. Landgrebe, „A survey of decision tree classifier methodology," *IEEE transactions on systems, man, and cybernetics,* т. 21, pp. 660-674, 1991.

[20] V. Desmet, L. Eeckhout и K. D. Bosschere, „Using decision trees to improve program-based and profile-based static branch prediction.," у *Advances in Computer Systems Architecture: 10th Asia-Pacific Conference, ACSAC*, Singapore, 2005.

[21] R. E. Wright, Logistic regression., 1995.

[22] R. Buse и W. Weimer, „The road not taken: Estimating path execution frequency statically," у *2009 IEEE 31st International Conference on Software Engineering*, 2009.

[23] T. Sanger, „Optimal unsupervised learning in a single-layer linear feedforward neural network," *Neural networks Elsevier,* т. 2, pp. 459-473, 1989.

[24] D. Svozil, V. Kvasnicka и J. Pospichal, „Introduction to multilayer feed-forward neural networks," *Chemometrics and intelligent laboratory systems,* т. 39, pp. 43-62, 1997.

[25] J. Keller, M. Gray и J. Givens, „A fuzzy k-nearest neighbor algorithm," *IEEE transactions on systems, man, and cybernetics,* pp. 580-585, 1985.

[26] M. Stephenson и S. Amarasinghe, „Predicting unroll factors using nearest neighbors," 2004.

[27] L. Medsker и L. Jain, Recurrent neural networks: design and applications, CRC press, 1999.

[28] K. Greff, R. Srivastava, J. Koutnik, B. Steunebrink и J. Schmidhuber, „LSTM: A search space odyssey," *IEEE transactions on neural networks and learning systems,* т. 28, pp. 2222-2232, 2016.

[29] S. Zekany, D. Rings, N. Harada, M. Laurenzano, L. Tang и J. Mars, „CrystalBall: Statically analyzing runtime behavior via deep sequence learning," у *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[30] A. Moreira, G. Ottoni и P. Fernando, „VESPA: Static Profiling for Binary Optimization," *Proceedings of the ACM on Programming Languages,* т. 5, pp. 1-28, 2021.

[31] D. E. Knuth и F. R. Stevenson, „Optimal measurementpoints for program frequency counts," *BIT Numerical Mathematics,* т. 3, бр. 13, pp. 313-322, 1973.

[32] L. Project, „How To Build Clang and LLVM with Profile-Guided Optimizations," https://llvm.org/docs/HowToBuildWithPGO.html, 2021.

[33] B. Wicht, R. Vitillo, D. Chen и D. Levinthal, „Hardware counted profile-guided optimization," *arXiv preprint,* 2014.

[34] R. Peri, S. Jinturkar и L. Fajardo, „A novel technique for profiling programs in embedded systems," у *ACM Workshop on Feedback-Directed and Dynamic Optimization*, 1999.

[35] M. Fowler и M. Foemmel, Continuous integration, 2006.

[36] H. K. Cho, T. Moseley, R. Hank, D. Bruening и S. Mahlke, „Instant profiling: Instrumentation sampling for profiling datacenter applications," у *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013.

[37] J. Whaley, „A portable sampling-based profiler for Java virtual machines," *Proceedings of the ACM 2000 conference on Java Grande,* pp. 78-87, 2000.

[38] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald и R. Peri, „Shadow profiling: Hiding instrumentation costs with parallelism," у *International Symposium on Code Generation and Optimization (CGO'07)*, 2007.

[39] T. Conte, B. Patel, K. Menezes и S. Cox, „Hardware-based profiling: An effective technique for profile-driven optimization," *International Journal of Parallel Programming,* т. 24, pp. 187-206, 1996.

[40] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus и R. Hundt, „Google-wide profiling: A continuous profiling infrastructure for data centers," *IEEE micro,* т. 30, pp. 65-79, 2010.

[41] D. Chen, T. Moseley и D. X. Li, „AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications," у *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2016.

[42] D. X. Li, R. Ashok и R. Hundt, „Lightweight feedback-directed cross-module optimization," у *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, 2010.

[43] G. Ottoni и B. Maher, „Optimizing function placement for large-scale data-center applications," у *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017.

[44] C.-K. Luk, R. Muth, H. Patil, R. Cohn и G. Lowney, „Ispike: a post-link optimizer for the Intel/spl reg/Itanium/spl reg/architecture," у *International Symposium on Code Generation and Optimization, CGO*, 2004.

[45] J. Gordon и E. Shortliffe, „The Dempster-Shafer theory of evidence," *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project,* т. 3, pp. 832-838, 1984.

[46] Y. Wu и J. Larus, „Static branch frequency and program profile analysis," Proceedings of the 27th annual international symposium on Microarchitecture.

[47] J. Gee, M. D. Hill, D. N. Pnevmatikatos и A. J. Smith, „Cache performance of the SPEC92 benchmark suite," *IEEE Micro,* т. 13, бр. 4, pp. 17-27, 1993.
[48] W.-F. Wong, „Source level static branch prediction," *The Computer Journal,* т. 42, pp. 142-149, 1999.
[49] Y. Yu, X. Si, C. Hu и J. Zhang, „A review of recurrent neural networks: LSTM cells and network architectures.," *Neural computation,* т. 31, бр. 7, pp. 1235-1270, 2019.
[50] H. Liu и T. Wu, „Estimating the Area under a Receiver Operating Characteristic Curve For Repeated Measures Design," *Journal of Statistical Software,* pp. 1-18, 2003.
[51] N. Rotem и C. Cummins, „Profile guided optimization without profiles: A machine learning approach.," *arXiv,* 2021.
[52] V. Vovk, „The fundamental nature of the log loss function.," *Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday,* pp. 307-318, 2015.
[53] T. Chen и a. C. Guestrin, „Xgboost: A scalable tree boosting system.," у *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data minin*, 2016.
[54] C. Lattner и V. Adve, „LLVM: A compilation framework for lifelong program analysis & transformation.," у *International symposium on code generation and optimization (CGO)*, 2004.
[55] J. L. Henning, „SPEC CPU2006 benchmark descriptions.," *ACM SIGARCH Computer Architecture News,* т. 34, бр. 4, pp. 1-17, 2006.
[56] C. Lattner, „LLVM and Clang: Next generation compiler technology," у *The BSD conference.*, 2008.
[57] B. Gough и R. Stallman, „ An Introduction to GCC," *Network Theory Limited,* 2004.
[58] B. Momjian, „PostgreSQL: introduction and concepts," *Addison-Wesley,* т. 192, 2001.
[59] R. Ju, S. Chan, C. Wu, R. Lian и T. Tuo, „Open Research Compiler (ORC) for ItaniumTM Processor Family.," 2001.
[60] K. Dixit, „The SPEC benchmarks.," *Parallel computing,* т. 17, pp. 1195-1201, 1991.
[61] K. Dixit, „Overview of the SPEC Benchmarks.," 1993.
[62] I. Steinwart и a. A. Christmann, Support vector machines, Springer Science & Business Media, 2008.
[63] SandraJohnson и S. Vall, „Hot method prediction using support vector machines," *Ubiquitous Computing and Communication Journal ,* т. 3, бр. 4, pp. 75-81, 2008.
[64] C. Lee, „UTDSP benchmark suite," 1998.
[65] A. Mahapatra и P. K. Patra, „Support Vector Machine for Frequently Executed Method Prediction," у *International Conference on Information Technology (ICIT)*, 2018.
[66] H. Urs и D. Ungar, „Optimizing dynamically-dispatched calls with run-time type feedback," *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation,* pp. 326-336, 1994.
[67] C. Zhang, H. Xu, S. Zhang, J. Zhao и Y. Chen, „Frequency estimation of virtual call targets for object-oriented programs.," у *ECOOP 2011–Object-Oriented Programming: 25th European Conference*, Lancaster, UK, 2011.
[68] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur и A. Diwan, „The DaCapo benchmarks: Java benchmarking development and analysis.," у *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2006.
[69] V. A. Cicirello, „Kendall tau sequence distance: Extending Kendall tau from ranks to sequences.," arXiv preprint, 2019.
[70] G. Fursin, C. Miranda, O. Temam, M. Namolaru, A. Zaks, B. Mendelson и E. Bonilla, „MILEPOST GCC: machine learning based research compiler.," у *GCC summit*, 2008.
[71] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru и E. Yom-Tov, „Milepost gcc: Machine learning enabled self-tuning compiler.," *International journal of parallel programming,* т. 39, pp. 296-327, 2011.
[72] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge и R. B. Brown, „MiBench: A free, commercially representative embedded benchmark suite.," у *Proceedings of the fourth annual IEEE international workshop on workload characterization.*, 2001.
[73] A. H. Ashouri, M. Elhoushi, Y. Hua, X. Wang, M. A. Manzoor, B. Chan и Y. Gao, „MLGOPerf: An {ML} Guided Inliner to Optimize Performanc," у *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2022.
[74] L. P. Kaelbling, M. L. Littman и A. W. Moore, „Reinforcement learning: A survey.," *Journal of artificial intelligence research ,* т. 4, pp. 237-285, 1996.
[75] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. O'Boyle и H. Leather, „Programl: A graph-based program representation for data flow analysis and compiler optimizations.," у *International Conference on Machine Learning*, 2021.
[76] Y. Li, D. Tarlow, M. Brockschmidt и R. Zemel., „Gated graph sequence neural networks.," *arXiv preprint,* 2015.
[77] N. Japkowicz, „Assessment metrics for imbalanced learning.," *Imbalanced learning: Foundations, algorithms, and applications ,* pp. 187-206, 2013.
[78] H. Dalianis и H. Dalianis, „Evaluation metrics and evaluation.," *Clinical text mining: secondary use of electronic patient records,* pp. 45-53, 2018.
[79] C.-Y. Shih, D. Svoboda, S.-J. Su и W.-C. Liao, „Static branch prediction for LLVM IRs using machine learning," 2021.
[80] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang и S. Y. Philip, „A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems,* т. 32, бр. 1, pp. 4-24, 2020.
[81] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li и M. Sun, „Graph neural networks: A review of methods and applications.," *AI open,* т. 1, pp. 57-81, 2020.
[82] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss и T. Würthinger., „Initialize once, start fast: application initialization at build time.," у *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2019.
[83] M. Čugurović, M. Vujošević Janičić, V. Jovanović и T. Würthinger, „GraalSP: Polyglot, Efficient, and Robust Machine Learning-Based Static Profiler," *The Journal of Systems & Software,* Submitted to.
[84] T. Chen и C. Guestrin, „Xgboost: A scalable tree boosting system.," у *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016.
[85] A. Prokopec, A. Rosà, D. Leopoldseder, G. Duboscq, P. Tůma, M. Studener, L. Bulej и e. al., „Renaissance: benchmarking suite for parallel applications on the JVM.," у *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019.
[86] S. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur и A. Diwan, „The DaCapo benchmarks: Java benchmarking development and analysis.," у *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2006.
[87] A. Sewe, M. Mezini, A. Sarimbekov и W. Binder, „Da capo con scala: Design and analysis of a scala benchmark suite for the java virtual machine.," у *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, 2011.

Милан Чугуровић, асистент,
Универзитет у Београду,
Математички факултет
Контакт: milan.cugurovic@matf.bg.ac.rs
Области интересовања: Машинско учење, компајлери, вештачка интелигенција