УНИВЕРЗИТЕТ У БЕОГРАДУ
ЕЛЕКТРОТЕХНИЧКИ ФАКУЛТЕТ

Маја Б. Вукасовић

ПОБОЉШАЊЕ ПЕРФОРМАНСИ ПРОГРАМА УПОТРЕБОМ ДЕЛИМИЧНО КОНТЕКСТНО ОСЕТЉИВИХ ПРОФИЛА

докторска дисертација

Београд, 2023.

UNIVERSITY OF BELGRADE
SCHOOL OF ELECTRICAL ENGINEERING

Maja B. Vukasovic

IMPROVING PROGRAM PERFORMANCE WITH PARTIALLY CONTEXT-SENSITIVE PROFILES

Doctoral Dissertation

Belgrade, 2023.

Ментор:

др Драган Бојић, редовни професор
Универзитет у Београду, Електротехнички факултет

Чланови комисије:

др Мило Томашевић, редовни професор
Универзитет у Београду, Електротехнички факултет

др Милош Цветановић, ванредни професор
Универзитет у Београду, Електротехнички факултет

др Милена Вујошевић Јаничић, ванредни професор
Универзитет у Београду, Математички факултет

др Павле Вулетић, ванредни професор
Универзитет у Београду, Електротехнички факултет

др Захарије Радивојевић, ванредни професор
Универзитет у Београду, Електротехнички факултет

Датум одбране: _____

Мами, деки и баки

Захвалница

Ова дисертација је резултат вишегодишњег рада на пројектима у оквиру компајлерске платформе *GraalVM* унутар организације *Oracle Labs*, која је део компаније *Oracle*. Захваљујем на сарадњи колегама из компаније, а посебно мом менаџеру др Александру Прокопецу, који је коаутор рада у коме су објављени главни резултати истраживања приказаног у овој тези.

Захваљујем свом ментору, проф. др Драгану Бојићу на подршци током докторских студија и израде дисертације, као и мојим колегама са Катедре за рачунарску технику и информатику са којима ближе сарађујем. Захваљујем члановима Комисије, који су конструктивним коментарима допринели побољшању квалитета дисертације, а посебно проф. др Милу Томашевићу на професионалној помоћи и пријатељским саветима, који су ми били изузетно корисни током израде дисертације као и у току моје целокупне досадашње каријере.

Највећу захвалност дугујем својој породици на великој љубави коју ми пружају. Ову дисертацију посвећујем мами, деки и баки, који су ми ветар у леђа и стална инспирација, потпора и снага. Они су ме научили истрајности и упорности, особинама које су се показале као једне од пресудних за успешну реализацију овог рада.

Наслов дисертације: Побољшање перформанси програма употребом делимично контекстно осетљивих профила

Резиме: Велика предност компајлера који раде превођење у току извршавања програма (JIT преводиоци) јесте доступност профила, који садрже информације о извршавању програма, током превођења. Профили усмеравају поредак превођења и оптимизације, које утичу на убрзавање извршавања програма. Превођење пре времена извршавања (AOT превођење) такође може да користи профиле, али добијене засебним извршавањима програма, намењеним прикупљању профила. Профили садрже метричке податке везане за програмски контекст. У зависности од дела програмског стека садржаног у контекстима профила, они могу бити контекстно осетљиви или неосетљиви. Како би се поједноставио процес прикупљања профила и смањила његова цена, уз очување прецизности, у многим системима се прикупљају делимично контекстно осетљиви профили, који садрже само суфикс програмског стека. Иако је уложено доста истраживачких напора у овој области, и даље постоји значајан потенцијал за унапређење перформанси програма на основу делимично контекстно осетљивих профила.

У овој тези је описан нов алгоритам под називом *PRINC* за унапређење процеса превођења и оптимизације инлајновања коришћењем делимично контекстно осетљивих профила. На основу профила, алгоритам идентификује често извршаване секције кода и преводи их са већим буџетом за оптимизације, притом не увећавајући значајно величину извршног кода. Ова техника је погодна за имплементацију у оквиру већине AOT преводилаца, који могу да користе делимично контекстно осетљиве профиле. Алгоритам *PRINC* укључује компоненту за детекцију често извршаваног кода како би реконструисао значајне целине кода, чијом агресивнијом оптимизацијом се постижу боље перформансе целокупног програма. Распоред превођења и оптимизација инлајновања су модификовани тако да користе информације о често извршаваном коду.

У тези је формално описан предложени алгоритам, укључујући његове компоненте и хеуристике. Такође, алгоритам *PRINC* је имплементиран као саставни део савременог AOT преводиоца *GraalVM Native Image*, а значајни делови имплементације су такође приказани у раду. Исцрпном евалуацијом предложеног алгоритма над 16 бенчмарка из скупова DaCapo, Scalabench и Renaissance показано је побољшање времена извршавања програма између 22% и 40% за 4 бенчмарка и између 2.5% и 10% за 5 бенчмарка. Увећање генерисаног кода варира између 0.8% и 9%, док је у случају 10 бенчмарка оно испод 2.5%. На основу добијених резултата показан је значајан потенцијал за унапређење перформанси, а у оквиру квалитативне евалуације показано је да алгоритам може бити имплементиран и у другим оптимизационим компајлерима уз адекватно прилагођавање конкретном окружењу.

Кључне речи: превођење, AOT преводиоци, распоред превођења, оптимизација инлајновања, профили, делимично контекстно осетљиви профили

Научна област: рачунарска техника и информатика

Ужа научна област: софтверско инжењерство

УДК број: 621.3

Dissertation title: Improving Program Performance with Partially Context-Sensitive Profiles

Abstract: Availability of profiling information is a major advantage of just-in-time (JIT) compilation. Profiles guide the compilation order and optimizations, thus substantially improving program performance. Ahead-of-time (AOT) compilation can also utilize profiles, obtained during separate profiling runs of the programs. Profiles can be context-sensitive, i.e., each profile entry is associated with a call-stack. To ease profile collection and reduce overheads, many systems collect partially context-sensitive profiles, which record only a call-stack suffix. Although there exists a significant portion of prior research work, there is still a lot of potential for further program performance improvement using partially context-sensitive profiles.

This thesis describes a novel technique that exploits partially context-sensitive profiles to determine which portions of code are hot, and compile them with additional compilation budget. This technique is applicable to most AOT compilers that can access partially context-sensitive profiles, and its goal is to improve program performance without significantly increasing code size. The technique relies on a new hot-code-detection algorithm to reconstruct hot regions based on the partial profiles. The compilation ordering and the inlining of the compiler are modified to exploit the information about the hot code. Proposed algorithm, named *PRINC*, is formally described, after which follows the description of the production-ready implementation inside GraalVM Native Image, a state-of-the-art AOT compiler for Java. Evaluation of the proposed technique on 16 benchmarks from DaCapo, Scalabench and Renaissance suites shows a performance improvement between 22% and 40% on 4 benchmarks, and between 2.5% and 10% on 5 benchmarks. Code-size increase ranges from $0.8 - 9\%$, where 10 benchmarks exhibit an increase of less than 2.5%. Based on the evaluation results it is proven that there exists a great potential for performance improvement and that the algorithm can be implemented in other modern compilers with respective algorithm adjustment.

Keywords: compilation, AOT compilers, compilation schedule, inlining optimization, profiles, partially context-sensitive profiles

Research area: computer science and information technology

Research sub-area: software engineering

UDC number: 621.3

Садржај

1 Увод **1**

2 Преглед области истраживања **5**
2.1 Стабла позива . . . 6
2.2 Компилација . . . 10
2.3 Профили . . . 11
2.3.1 Контекстна осетљивост профила . . . 12
2.3.2 Технике прикупљања профила . . . 13
2.3.3 Стратегија прикупљања профила у преводиоцу *GraalVM NI* . . . 15
2.4 Употреба профила у процесу превођења . . . 16
2.4.1 Поступак дупликације кода . . . 16
2.4.2 Оптимизација смештања блокова . . . 18
2.4.3 Оптимизације одмотавања петљи . . . 19
2.5 Изазови оптимизације инлајновања . . . 20
2.5.1 „Загађене" фреквенције профила . . . 20
2.5.2 „Загађени" профили позива виртуелних метода . . . 22
2.5.3 Инлајновање угњеждених полиморфних позива . . . 23
2.5.4 Занемарени ретко извршавани позиви . . . 24
2.5.5 Ретко позиване методе садрже често извршавани код . . . 25
2.6 Преглед постојеће литературе . . . 25
2.6.1 Стратегије и класификација прикупљања профила . . . 26
2.6.2 Делимична контекстна осетљивост профила . . . 27
2.6.3 Употреба профила у детекцији често извршаваних делова кода и оптимизацији инлајновања . . . 27

3 Екосистем *GraalVM* **30**
3.1 Структура система *GraalVM* . . . 30
3.2 Превођење у систему *GraalVM Native Image* . . . 31
3.3 Међурепрезентација компајлера . . . 33
3.4 Оптимизација инлајновања и позиционирање чворова у коду . . . 34
3.5 Прикупљање и употреба профила . . . 35
3.5.1 Поступак прикупљања профила . . . 36
3.5.2 Поступак употребе профила . . . 37

4 Поставка проблема **41**
4.1 Опис проблема . . . 41
4.2 Нацрт предложеног решења . . . 44

5 Структуре података за представљање често извршаваног кода **46**
5.1 Стаза трагова . . . 46
5.2 Шума стаза . . . 49

6 Нови алгоритам превођења **55**
6.1 Постојећи алгоритам превођења . . . 55
6.2 Улаз и излаз алгоритма . . . 56
6.3 Алгоритам *PRINC* . . . 58
6.4 Алгоритам за детекцију често извршаваног кода . . . 61
6.4.1 Стратегије детекције често извршаваног кода . . . 65
6.4.2 Фреквенција извршавања стазе трагова . . . 67
6.5 Модификација алгоритма за инлајновање . . . 72
6.6 Класификација преосталих често извршаваних позива . . . 75
6.7 Преглед параметара . . . 78

7 Детаљи имплементације **79**
7.1 Структуре података за анализу кода . . . 79
7.1.1 Структура података `Breadcrumb` . . . 80
7.1.2 Структура података `Trail` . . . 80
7.1.3 Структура података `TrailSet` . . . 82
7.2 Детаљи имплементације продужавања контекста . . . 83
7.2.1 Одређивање контекста позивања . . . 83
7.2.2 Одређивање фреквенције позива виртуелних метода . . . 83
7.2.3 Одређивање фреквенције директних позива метода . . . 84
7.2.4 Процена фактора умањења контекста позивања . . . 86
7.3 Пример компилационе јединице . . . 86
7.4 Пример превођења . . . 88

8 Евалуација **91**
8.1 Методологија експеримената . . . 91
8.1.1 Опис тестова . . . 93
8.2 Поређење перформанси релевантних алгоритама за инлајновање . . . 94
8.3 Утицај параметара инлајнера на перформансе програма . . . 96
8.3.1 Подешавање вредности параметара инлајнера . . . 96
8.3.2 Параметар *Expansion-Inertia Base Value* . . . 97
8.4 Утицај алгоритма на величину преведеног кода . . . 99
8.5 Утицај алгоритма на време превођења . . . 101
8.6 Утицај дужине контекста на перформансе . . . 102
8.7 Утицај хеуристике инлајнера на перформансе . . . 103
8.7.1 Транзитивно превођење често позиваних метода . . . 104

8.7.2 Ширење стазе трагова 104
8.7.3 Величина почетног скупа често извршаваних контекста 105
8.8 Утицај прикупљања профила на перформансе 106
8.9 Утицај различитих реализација механизама преводилаца *Graal* и *Native Image* на перформансе 107
8.9.1 Исечци кода 108
8.9.2 Ослобађање меморије 109
8.9.3 Спекулативне оптимизације 110

9 Поређење алгоритма *PRINC* са релевантним решењима 112
9.1 GCC 112
9.2 LLVM 113
9.3 Примењивост алгоритма *PRINC* у окружењима LLVM и GCC 114
9.4 Поређење са релевантном литературом 115

10 Закључак 118

Литература 121

Листа слика

2.1 Пример нотације позива потпрограма 6
2.2 Примери графова позива и максималних стабала позива 7
2.3 Примери релација *подскуп* и *уређен* 8
2.4 Примери стабла позива и активационог стабла 9
2.5 Примери излаза алгоритма за превођење 12
2.6 Контекстно осетљиви и контекстно неосетљиви профили 13
2.7 Пример трансформације графа дупликацијом кода 17
2.8 Пример оптимизације смештања кода 18

3.1 Главни кораци превођења 32
3.2 Пример међурепрезентације компајлера *Graal* 34
3.3 Пример графа са инлајновањем 35
3.4 Пример графа са инструментационим чворовима 37
3.5 Пример графа са означеним вероватноћама 39
3.6 Пример графа са означеним контекстно осетљивим вероватноћама 39

4.1 Пример активационог стабла са означеним компилационим јединицама . . . 42
4.2 Илустрација суштине предлога решења 44

5.1 Примери стаза трагова 47
5.2 Пример операције продужавања стаза трагова 48
5.3 Пример операције спајања стаза трагова 48
5.4 Пример одређивања поретка чворова стазе трагова 49
5.5 Примери операције спајања стазе ξ на све кандидате-тачке спајања стазе τ . 50
5.6 Примери операције спајања шума стаза 51
5.7 Пример операције самоспајања стаза 52
5.8 Пример операције кореног самоспајања стаза 53

6.1 Трансформација улазног мултиграфа 57
6.2 Пример распореда превођења 60
6.3 Поступак детекције често извршаваног кода 62
6.4 Процена времена извршавања дела кода обухваћеног стазом 67
6.5 Функција граничне вредности *threshold* 71
6.6 Упаривање стабла инлајновања са стазом трагова 76
6.7 Операције упаривања и пресецања стаза 77

7.1 Одређивање фреквенције позива виртуелних метода 84
7.2 Одређивање фреквенције директних позива метода 85
7.3 Део стабла инлајновања за методу `encode` из бенчмарка Mnemonics 87
7.4 Део стазе трагова који одговара методи `encode` из бенчмарка Mnemonics . . 88
7.5 Део графика извршавања који одговара бенчмарку Mnemonics преведеног уз примену алгоритма *PRINC* 89
7.6 Део графика извршавања који одговара бенчмарку Mnemonics преведеног уз примену постојећег алгоритма за превођење и инлајновање 89

8.1 Време постизања стабилних перформанси са преводиоцем *Native Image* . . . 92
8.2 Време извршавања бенчмарка 94
8.3 Издвојено поређење времена извршавања бенчмарка 95
8.4 Тражење најбоље вредности параметра *EIBV* 98
8.5 Време извршавања бенчмарка за оптималне вредности параметра *EIBV* . . . 99
8.6 Величина компајлираног кода 100
8.7 Време превођења 101
8.8 Утицај дужине контекста на време извршавања бенчмарка 102
8.9 Транзитивно превођење често позиваних метода као значајних јединица компилације 103
8.10 Тражење најбоље вредности прага за укључивање у иницијални скуп често извршаваних контекста 105
8.11 Разлика у имплементацији исечка `copyOf` 108
8.12 Утицај исечка за копирање низова на перформансе програма 109
8.13 Поређење алгоритама за ослобађање меморије 110
8.14 Утицај спекулативних оптимизација на перформансе 111

Листа табела

2.1 Преглед главних појмова дефинисаних у секцији 2.1 10

5.1 Преглед главних симбола и операција које описују модел 54

6.1 Преглед главних симбола и функција коришћених у секцији 6.4 70
6.2 Преглед главних симбола и функција из секција 6.5 и 6.6 77
6.3 Преглед главних параметара уведених у алгоритму *PRINC* 78
6.4 Преглед главних параметара у постојећем инлајнеру 78

8.1 Број итерација потребан за постизање стабилног стања бенчмарка 93
8.2 Утицај прикупљања профила на времена превођења и извршавања програма 107

1 Увод

Интерпретирање и компајлирање представљају две главне парадигме превођења изворног програмског кода. Интерпретирање подразумева превођење сваке наредбе програма засебно сваки пут када је дату наредбу потребно извршити, без памћења претходно преведених наредби. Са друге стране, компајлирање подразумева превођење целокупног програма или његових делова у циљни програмски код уз памћење резултата. Иако компајлирање у општем случају представља превођење из било ког изворног програмског кода (енг. *source code*) у било који циљни програмски код (енг. *target code*), у овој тези фокус ће бити на превођењу у извршни код и у даљем тексту ће се такав начин компајлирања подразумевати под појмовима компајлирање и превођење.

Код превођења програма написаних у програмском језику Java или другим језицима који се преводе и извршавају коришћењем виртуелних машина, користе се парадигме: превођење пре времена извршавања (енг. *ahead-of-time*, скр. *AOT*) и превођење у току времена извршавања (енг. *just-in-time*, скр. *JIT*). AOT превођење подразумева компајлирање програмског кода у целости пре него што програм почне са извршавањем, а време самог превођења не утиче директно на време извршавања програма. Са друге стране, време превођења у случају JIT компилације представља критичан фактор. Да би се смањила цена превођења кода у току извршавања програма, често се комбинују интерпретација и компајлирање кода, тако да се у извршни код преводи само део изворног програмског кода. Извршавање програма, у том случају, почиње у првој фази, која најчешће представља интерпретер, а како је процес компајлирања скуп, он се потом селективно спроводи над често извршаваним деловима кода (енг. *hot code*) [1]. Компајлирани код је бржи у односу на код који се интерпретира, па тако AOT превођење превазилази једну од главних мана JIT превођења, а то је споро почетно извршавање програма (енг. *start-up time*) [2, 3].

Осим генерисања исправног извршног кода, добри преводиоци имају за циљ да генеришу што ефикаснији извршни код и то постижу применом различитих оптимизационих пролаза у фази превођења програма. Неке од најзначајнијих оптимизација су девиртуализација полиморфних позива (енг. *devirtualization*) [4, 5], спекулативна миграција кода (енг. *speculative code motion*) [6], алокација регистара (енг. *register allocation*) [7], аутлајновање функција тј. издвајање тела функција (енг. *function outlining*) [8], инлајновање функција тј. уграђивање тела функција (енг. *function inlining*) [9, 10] као и бројне оптимизације петљи попут одмотавања петљи (енг. *loop unrolling*) [11] и љуштења петљи (енг. *loop peeling*) [12, 13].

Већина преводилаца ослања се првенствено на оптимизације у оквиру функција (енг.

intraprocedural optimizations) [14, 15, 16, 17], што представља главну мотивацију за фокусирање овог рада на оптимизацију инлајновања функција. Оптимизација инлајновања подразумева уграђивање тела позване функције на место позива и тиме елиминише трошкове самог позива, а омогућава оптимизовање уграђеног кода на основу локације уграђивања. Кључно је да ова оптимизација директно доводи до повећања тела функција у којима се десило уграђивање и, самим тим, је оптимизацијама које се примењују у наредним фазама превођења (енг. *compilation pipeline*) доступно више информација на основу којих је могуће донети боље одлуке. Превелико коришћење ове оптимизације доводи до значајног повећања генерисаног кода, па се она мора спроводити контролисано.

Селективна примена оптимизације спроводи се, у случају већине оптимизација, над често извршаваним деловима кода. У супротном, оптимизациони напори били би уложени и у код који се не извршава често и нема значајан утицај на перформансе програма, па би режијски трошкови спровођења оптимизација превазилазили корист добијену њиховом применом. За исправну детекцију често извршаваног кода, као и у циљу доношења што бољих оптимизационих одлука, хеуристике, које су део оптимизација, користе информације о извршавању програма који се преводи [18, 19, 20, 21, 22, 23, 24]. Како су ове информације доступне у току извршавања програма, JIT компилација може у потпуности да их искористи, док AOT компилација мора да пронађе алтернативне начине прибављања информација о извршавању програма.

Током AOT превођења могу се користити искључиво информације о претходним извршавањима истог или сличног програма [25, 26, 27]. Ове информације се називају профили, а добијене су процесом који се назива *offline* профилисање (енг. *offline profiling*). *Online* профилисање (енг. *online profiling*) везано је углавном за прикупљање профила у JIT преводиоцима. Метрички подаци који се налазе у профилима повезани су са програмским контекстом. Контекст подразумева низ од једне или више локација у коду, које се налазе на програмском стеку (стеку позива функција) у тренутку прикупљања профила. У зависности од броја локација у овом низу, профили се могу категоризовати као контекстно осетљиви (енг. *context-sensitive*) или контекстно неосетљиви (енг. *context-insensitive*). Контекстно неосетљиви профили садрже искључиво информацију о позицији догађаја који се прати у оквиру функције са врха програмског стека (дужина контекста је један). Контекстно осетљиви профили у општем случају садрже информације о већем броју локација на програмском стеку узимајући у обзир, поред директних, и индиректне позиваоце функције са врха стека. Захваљујуће томе, могуће је прецизније успоставити везу између метрике из профила са стварном локацијом у програму. Дужина контекста је у највећој мери условљена техником и начином прикупљања профила и директно утиче на њихову прецизност.

Док већина окружења заснованих на JIT превођењу, због једноставности и мале цене, прикупља контекстно неосетљиве профиле [16, 28, 29, 30, 31], *offline* профилисање омогућава прикупљање прецизнијих контекстно осетљивих профила. Ипак, чак и у *offline* окружењу прикупљање прецизних потпуно контекстно осетљивих профила (енг. *fully-context-sensitive profiles*), где сваки контекст профила садржи све локације са програмског стека, може бити скупо, па је развијено више приступа да се смањи цена

профилисања [32, 33, 34, 35, 36]. Један уобичајени приступ је прикупљање делимично контекстно осетљивих профила (енг. *partially context-sensitive profiles*). Контексти оваквих профила садрже суфикс програмског стека одређене дужине [14, 37, 38, 39, 40]. Чување делимично контекстно осетљивих профила је просторно ефикасније од чувања комплетних контекста, а, додатно, може и обезбедити довољну прецизност информација имајући у виду да потпуно контекстно осетљиви профили садрже и неке информације које нису значајне за постизање бољих перформанси програма. Прикупљање парцијалних профила може бити ефикасно као и прикупљање контекстно неосетљивих профила, али не постоје јасно дефинисане процедуре за коришћење оваквих профила у различитим окружењима у циљу постизања бољих перформанси програма. Начин употребе парцијалних контекста у оквиру различтих алгоритама и њихов допринос побољшавању перформанси програма умногоме зависи од садржаја и прецизности самих профила, као и од могућности њихове интеграције у конкретна окружења.

Циљ истраживања представљеног у овом раду је конструкција алгоритма за коришћење делимично контекстно осетљивих профила како би се побољшало AOT превођење програма. Предложени алгоритам користи парцијалне профиле како би детектовао делове често извршаваног кода, односно, како би груписао и означио секције кода значајне за перформансе целог програма, а затим формира јединице компилације које обухватају такве делове кода. Како се у оквиру ових јединица применом оптимизације инлајновања повећава количина кода коју преводилац „види", осим саме оптимизације, кључно је унапредити и сам избор метода које ће бити компајлиране, као и редослед њиховог превођења.

Прва фаза алгоритма подразумева опортунистичку реконструкцију делова често извршаваног кода спајањем делимично контекстно осетљивих профила, како би се направила шума стабала која прате често извршаване делове програма. У другој фази, алгоритам модификује редослед превођења издвајањем корена идентификованих стабала као јединица од којих превођење започиње. Током компилације метода које су означене као често извршаване, алгоритам повећава буџет оптимизације инлајновања и користи профиле како би унапредио одлуке ове оптимизације. Преостале јединице компилације, које се ретко извшавају, преводе се према оригиналној шеми компилације, без икаквих модификација. У наставку текста нови алгоритам за инлајновање и распоређивање компилације на темељу профила назван је *PRINC* (енг. *PRofile-driven INlining and Compilation scheduling*) алгоритам [41].

Доприноси тезе простиру се у неколико праваца:

- Опсежан преглед релевантних подобласти у домену преводилаца, укључујући процес прикупљања профила, оптимизације које користе профиле како би унапредиле одлуке, изазове оптимизације инлајновања итд. дат је у поглављу 2. Преглед конкретне инфраструктуре *GraalVM*, у којој је предлоено побољање имплементирано, са посебним нагласком на кључне компоненте коришћене у истраживању налази се у поглављу 3.

- Предложена је нова техника за превођење, која скраћује време извршавања AOT преведених програма. Како би ово постигла, ова техника користи делимично кон-

текстно осетљиве профиле за детекцију и ограничавање секција често извршаваног кода, за измену распореда превођења тако да оно почиње од издвојених јединица компилације и за модификцију постојеће оптимизације инлајновања. За детекцију често извршаваних делова кода, алгоритам користи хеуристику за апроксимацију дужих контекста позивања на основу краћих суфикса са програмског стека. Дефиниција и детаљи алгоритма приказани су у поглављу 6. Структуре података уведене за представљање модела значајних секција кода описане су у поглављу 5.

- Алгоритам је имплементиран као саставни део AOT компајлера GraalVM Native Image и детаљи имплементације налазе се у поглављу 7. Прецизније, у оквиру постојећег преводиоца GraalVM имплементирана је модификација реда за распоређивање превођења јединица компилације, додата је нова фаза за анализу профила, а резултати ове анализе су даље коришћени како би се побољшале одлуке оптимизације инлајновања.

- Спроведена је опсежна евалуација над 16 бенчмарка из скупова DaCapo [42], Scalabench [43] и Renaissance [44], чији резултати су приказани и детаљно дискутовани у поглављу 8. У поређењу са претходном имплементацијом алгоритма за превођење и оптимизације инлајновања у оквиру истог преводиоца, постигнуто је убрзање у опсегу од 22% до 40% на 4 бенчмарка и у опсегу од 2.5% до 10% на 5 бенчмарка. Алгоритам увећава величину генерисаног извршног кода између 0.8% и 9%, а време потребно за превођење програма се увећава за максимално 2.5% у случају 10 бенчмарка. Перформансе алгоритма су поређене и са другим релевантним конфигурацијама превођења и извршавања програма попут компајлирања преводиоцем GraalVM Native Image без употребе оптимизација на основу профила, стандардним GraalVM компајлером у JIT моду и подразумеваним C2 компајлером у оквиру виртуелне машине HotSpot. Евалуација такође приказује и процес подешавања параметара алгоритма ради постизања најбољих перформанси.

- Додатно, спроведена је и квалитативна евалуација, тј. детаљна анализа проблема које алгоритам решава (а који је постављен у поглављу 4) и његових компонената, као и перспективе за имплементацију ових компонената у оквиру два доминантно коришћена компајлерска окружења LLVM [45] и GCC [46]. Ова анализа заједно са поређењем новог алгоритма са најрелевантнијим радовима који решавају сличне проблеме приказана је у поглављу 9.

У закључку, који се налази у поглављу 10, сумирани су предмет истраживања и опис проблема. Такође, приказани су и главни практични доприноси рада. За крај, дати су и предлози за могућа унапређења алгоритма за превођење и оптимизације инлајновања, за које је очекивано да могу допринети даљем побољшању перформанси AOT преведених програма.

2 Преглед области истраживања

У овом поглављу биће детаљније објашњени кључни појмови потребни за разумевање предложеног алгоритма за измену распореда превођења и инлајновање. Биће дефинисане основне структуре података на које се ослања модел који се користи за детекцију често извршаваних делова кода, издвајање јединица компилације од којих превођење треба да започне, и интеграцију ових информација у оптимизацију инлајновања. Затим ће бити речи о начину превођења који се спроводи у оквиру конкретног преводиоца коришћеног током имплементације алгоритма *PRINC* [41], а који диктира улаз алгоритма. Такође, потребно је дати увид у појам профила са посебним нагласком на делимично контекстно осетљиве профиле, а биће описане и неке од оптимизација које користе профиле ради доношења бољих одлука. Како је један од значајних аспеката рада оптимизација инлајновања, биће речи о изазовима са којима се суочавају инлајнери из перспективе корисника, као и о начинима за њихово решавање, када је то могуће.

```
1  void foreach(int[] xs, int->void f) {
2    for (int i = 0; i < xs.length; i++)
3      f.apply(xs[i]);
4  }
5
6  void main(int[] args) {
7    min(args);
8    max(args);
9  }
10
11 int min(int[] xs) {
12   int m = Integer.MAX_VALUE;
13   foreach(xs, x -> if (x < m) m = x);
14   return m;
                      F.apply
15 }
16
17 int max(int[] xs) {
18   int m = Integer.MIN_VALUE;
19   foreach(xs, x -> if (x > m) m = x);
20   return m;
                      G.apply
21 }
```

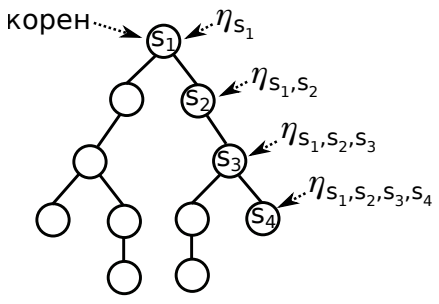
Листинг 2.1: Пример програма

Пример програма. На самом почетку приказан је репрезентативни пример програма, који ће у оквиру овог и у наредним поглављима бити коришћен како би се објасниле структуре података којима се моделује превођење и извршавање програма, а потом и алгоритам предложен у овом раду. Програм приказан у листингу 2.1 израчунава најмањи и највећи цео број из листе прослеђених аргумената. Овај програм садржи генеричку функцију `foreach` која примењује ламбда функцију `f` на сваки цео број у датом низу. Почевши од методе `main`, програм позива методе `min` и `max` једну за другом. Свака од ове две методе позива `foreach` са ламбда вредношћу која прати најмању, односно највећу целобројну вредност, респективно. Значајно је приметити да само на основу дефиниције методе `foreach` није могуће закључити која је конкретна имплементација ламбда функције `f` – ово зависи од локације позива `foreach` методе. Кажемо да је позив функције `f` индиректан [47]. Са друге стране, позиви метода `max`, `min` и `foreach` су директни.

2.1 Стабла позива

Нотација. Пре саме дефиниције стабала позива (енг. *call trees*) биће усвојена следећа шема која се односи на чворове у оквиру стабала. Претпоставимо да је сваки чвор у стаблу повезан са неком методом s програма. Може постојати више чворова у стаблу који се односе на исту методу програма. Тада се нотација $\eta_{s_1,s_2,\ldots,s_n}$ односи на чвор до кога се долази почев од корена стабла, чија је одговарајућа метода s_1, праћењем секвенце чворова којима одговарају методе s_2, s_3, итд. све до чвора који одговара методи s_n. Другим речима, ово стабло је могуће третирати као префиксно стабло [48], а секвенцу $\eta_{s_1,s_2,\ldots,s_n}$ као префикс који се чува у стаблу. За потребе овог рада, чвор $\eta_{s_1,s_2,\ldots,s_n}$ ће представљати садржај стека позива метода. Ова ситуација је илустрована на слици 2.1. Нотација η или η_x, која не садржи секвенцу у индексу, односи се на било који чвор у стаблу, а користи се када префикс чвора није значајан за конкретну дискусију. У случају да се у индексу налази секвенца од само једног елемента, биће посебно објашњено значење таквог симбола.

Дефиниција. За неки граф позива (енг. *call graph*) [49, 50] G, постоји функција $callees_G(s)$, која као резултат враћа скуп метода које могу бити позване из методе s, на основу графа позива G. У оквиру примера из листинга 2.1, функција



Слика 2.1: Пример нотације позива потпрограма

$callees_{Листинг\ 2.1}$(`foreach`) састоји се од `F.apply` и `G.apply`.

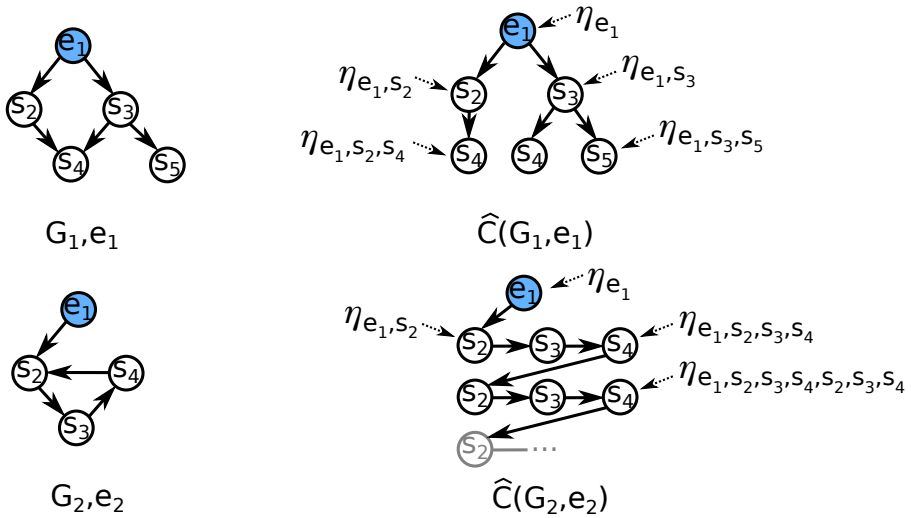
$$callees_G(s) \equiv \{s_2 : s \to s_2 \in E\} \qquad \textit{где је} \quad G = (S, E \subseteq S \times S) \tag{2.1}$$

Због лакше читљивости G ће касније у тексту бити изостављено из индекса и увек се мисли на граф G који је компајлиран.

Нека се програм $P = (G, e)$ састоји од графа позива G и почетне методе програма e (енг. *entry point*). Максимално стабло позива (енг. *maximum call tree*) $\widehat{C}(P)$ за програм P је пар вредности (N, E) који се састоји од скупа чворова N и скупа грана E, тако да важе следеће особине: корени чвор одговара улазној методи e програма P, сваки чвор $\eta_{e,\ldots,s_n}$ одговара стеку позива који се завршава методом $s_n = sub(\eta_{e,\ldots,s_n})$ и скуп чворова потомака чвора $\eta_{e,\ldots,s_n}$ је $\eta_{e,\ldots,s_n,s_{n+1}}$ тако да важи $s_{n+1} \in callees_G(s_n)$. Следећа једначина дефинише максимално стабло позива које одговара програму P.

$$\begin{aligned} \widehat{C}(P) \equiv (N, E \subseteq N \times N) \quad & \textit{где је} \quad \eta_e \in N \wedge \\ & \big(s_{n+1} \in callees_G(s_n) \Leftrightarrow \big(\eta_{e,\ldots,s_n} \in N \Leftrightarrow \eta_{e,\ldots,s_n,s_{n+1}} \in N\big)\big) \wedge \\ & \big(s_{n+1} \in callees_G(s_n) \Leftrightarrow \big(\eta_{e,\ldots,s_n} \in N \Leftrightarrow \eta_{e,\ldots,s_n} \to \eta_{e,\ldots,s_n,s_{n+1}} \in E\big)\big) \end{aligned} \tag{2.2}$$

Из једначине 2.2 следи да када је граф позива програма цикличан, другим речима, могући су рекурзивни позиви, максимално стабло позива представља бесконачни граф (енг. *infinite graph*). На слици 2.2 приказана су два графа позива и њима одговарајућа максимална стабла позива. У првом случају, у питању је коначан граф (енг. *finite graph*), а у другом бесконачан граф.



Слика 2.2: Примери графова позива и максималних стабала позива

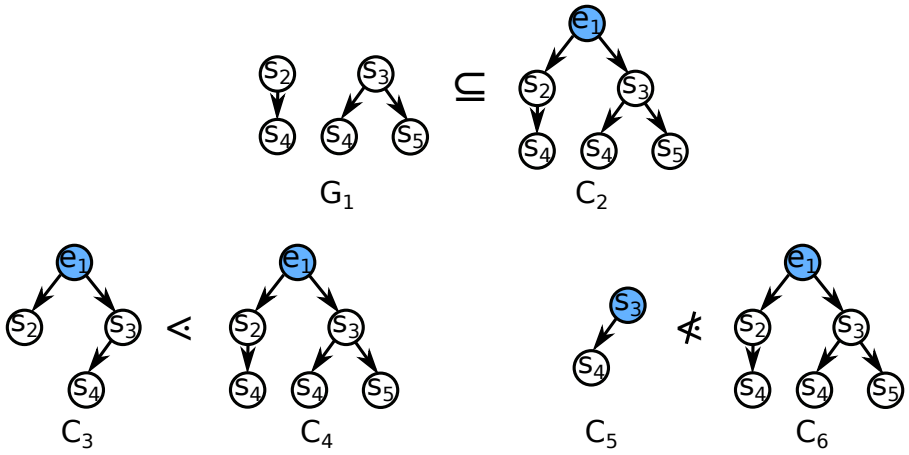
Затим, биће уведена два парцијална поретка над графовима. Релација *подскуп* (енг. *subset*) између графова $C_1 = (N_1, E_1)$ и $C_2 = (N_2, E_2)$ дефинисана је на следећи начин:

$$(N_1, E_1) \subseteq (N_2, E_2) \equiv N_1 \subseteq N_2 \wedge E_1 \subseteq E_2 \tag{2.3}$$

Граф $C_1 = (N_1, E_1)$ *уђњежден* је у граф $C_2 = (N_2, E_2)$ ако и само ако важе следећи услови. E_1 и E_2 уводе парцијални поредак над скуповима чворова N_1 и N_2 (другим речима C_1 и C_2 су усмерени ациклични графови), C_1 и C_2 имају заједнички корен (енг. *common infimum*) и C_1 је подскуп графа C_2. Нотација $\inf_{E^*} N$ подразумева да постоји јединствени последњи члан скупа чорова N, што даље имплицира да C_1 и C_2 морају бити повезани и ациклични:

$$(N_1, E_1) \lessdot (N_2, E_2) \equiv \inf_{E_1^*} N_1 = \inf_{E_2^*} N_2 \wedge (N_1, E_1) \subseteq (N_2, E_2) \tag{2.4}$$

Релације *подскуп* и *уђњежден* илустроване су примером са слике 2.3. Граф G_1, који се састоји од две неповезане компоненте, не представља стабло позива, али јесте подскуп стабла позива C_2. Стабло позива C_3 угњеждено је у C_4, али стабло позива C_5 није угњеждено у C_6 зато што немају заједнички корен.

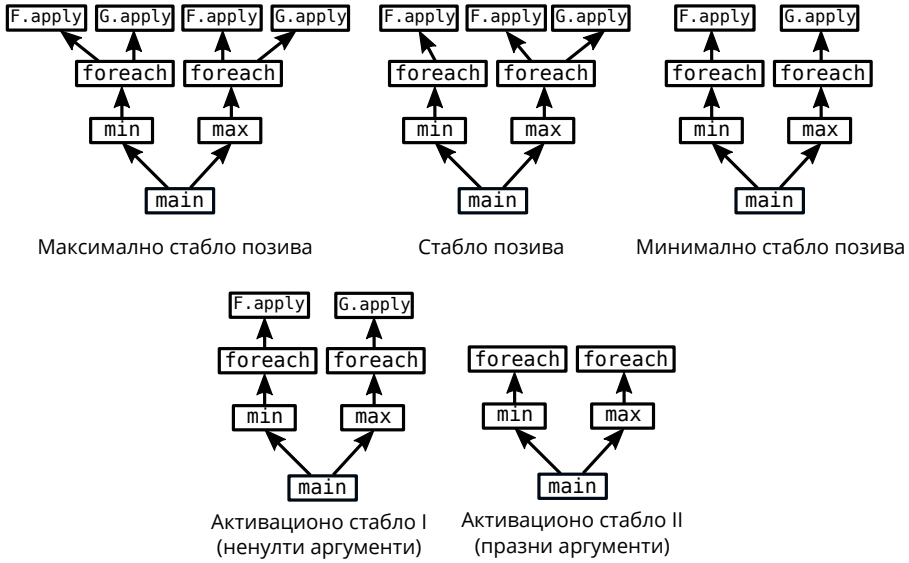


Слика 2.3: Примери релација *подскуп* и *уђњежден*

Сада је могуће направити разлику између дефиниција максималних стабала позива и *активационих стабала* (енг. *activation trees*), који садрже само оне путање (енг. *trace*) које одговарају садржајима стека позива за дату инстанцу програма. Самим тим одговарају конкретном извршавању програма над специфичним улазом ι. Под претпоставком да скуп $trace(P, \iota)$ садржи све стекове позива који се формирају током извршавања програма P за дати улаз ι важи да активационо стабло $A(P, \iota)$ представља префиксно стабло које садржи све ове путање:

$$\begin{aligned} A(P,\iota) \equiv (N, E \subseteq N \times N) \qquad & \textit{тако да важи} \quad \eta_e \in N \wedge \\ & (\langle e, \ldots, s_n \rangle \in trace(P,\iota) \Leftrightarrow \eta_{e,\ldots,s_n} \in N) \wedge \\ & \big(\langle e, \ldots, s_n, s_{n+1} \rangle \in trace(P,\iota) \Leftrightarrow \eta_{e,\ldots,s_n} \rightarrow \eta_{e,\ldots,s_n,s_{n+1}} \in E\big) \end{aligned} \tag{2.5}$$

За све програме који се завршавају, одговарајућа активациона стабла су увек коначна. За програме који се не завршавају, активациона стабла могу бити бесконачна. На основу дефиниција активационог стабла и максималног стабла позива, може се приметити да су сва активациона стабла угњежедена у максимална стабла позива, односно



Слика 2.4: Примери стабла позива и активационог стабла

$\forall \iota, A(P,\iota) \lessdot \widehat{C}(P)$. Ипак, алгоритамско одређивање тачног садржаја активационог стабла у општем случају захтева извршавање програма над задатим улазним вредностима, и зато је неодлучиво.

Сада је могуће дефинисати *минимално стабло позива* (енг. *minimum call tree*) $\check{C}(P)$, које садржи путању $\eta_e, \ldots, \eta_{e,\ldots,s_n}$ ако и само ако се стек позива $e, \ldots, s_n$ појављује у скупу *путање*(P,ι) барем за неки улаз ι програма P. Тада минимално стабло позива може бити дефинисано и као унија свих активационих стабала за све улазе програма:

$$\check{C}(P) \equiv \bigcup_{\iota} A(P,\iota) \tag{2.6}$$

Унија два графа $(N_1,E_1)\cup(N_2,E_2)$ представља унију њихових чворова и грана $(N_1\cup N_2, E_1\cup E_2)$. Свако минимално стабло позива угњеждено је у максимално стабло позива, односно $\check{C}(P) \lessdot \widehat{C}(P)$. Ако је минимално стабло позива бесконачно, тада је програм рекурзиван и не терминира за неке улазе (супротно не мора да важи).

Минимално стабло позива показује које стекове позива програм може формирати. Када се оптимизује јединица компилације која одговара методи e, преводилац би идеално искористио одговарајуће минимално стабло позива. Међутим, извођење овог стабла је у општем случају неодлучиво, као и израчунавање и формирање њему припадајућих активационих стабала. Самим тим, преводилац је приморан да користи апроксимацију $\check{C}(P)$ коју може да израчуна – алгоритам за инлајновање типично примењује комплексну анализу која врши одсецање над максималним стаблом позива.

Скуп стабала позива $\mathbb{C}(P)$ састоји се од свих стабала C угњеждених у максимална стабла позива $\widehat{C}(P)$, таквих да је минимална стабла $\check{C}(P)$ угњеждена у стабла C.

$$\mathbb{C}(P) \equiv \{C : \check{C}(P) \lessdot C \lessdot \widehat{C}(P)\} \tag{2.7}$$

Пример. У наставку ће ови концепти бити представљени на конкретним примерима. Прво стабло слева на слици 2.4 је максимално стабло позива за програм из примера 2.1.

Табела 2.1: Преглед главних појмова дефинисаних у секцији 2.1

Симбол	**Име**	**Објашњење**
(G, e)	Програм	Граф позива G са процедуром e, која је улазна тачка програма.
$callees_G(s)$	Callees (Јед. 2.1)	Скуп метода које могу бити позване из s у графу позива G (излазне гране из чвора s).
$\eta_{s_1,\ldots,s_n}$	Чвор (у стаблу)	Чвор који одговара стеку позива метода s_1, s_2, ..., s_n у стаблу позива са кореном методом s_1.
$\widehat{C}(G, e)$	Максимално стабло позива (Јед. 2.2)	Стабло чији је корен улазна метода e, и сваки чвор $\eta_{e,\ldots,s_n}$ има потомке $\eta_{e,\ldots,s_n,s_n+1}$, тако да важи $s_{n+1} \in callees_G(s_n)$.
$trace(P, \iota)$	Скуп путања	Скуп стекова позива који се појављују током извршавања програма P над улазом ι.
$A(P, \iota)$	Активационо стабло (Јед. 2.5)	Део $\widehat{C}$ који садржи само оне стекове позива за дату инстанцу програма.
$\breve{C}(G, e)$	Минимално стабло позива (Јед. 2.6)	Стабло добијено унирањем свих стекова позива који се појављују у бар неком извршавању програма.
$C_1 \lessdot C_2$	Релација угњежденост (Јед. 2.4)	Стабло C_1 угњеждено је у стабло C_2 ако имају заједнички корен, и стабло C_1 је подскуп стабла C_2.
$\mathbb{C}(G, e)$	Стабла позива (Јед. 2.7)	Скуп стабала C, таквих да су стабла C угњеждена у $\widehat{C}(G, e)$ и $\breve{C}(G, e)$ је угњеждено у C.

Друго стабло представља једно могуће стабло позива према дефиницији у једначини 2.7. Треће стабло је минимално стабло позива које преводилац може да одреди за овај конкретан програм на основу једноставне анализе тока података (енг. *dataflow analysis*). Четврто стабло је једно активационо стабло за програм из листинга 2.1 које је у овом случају идентично минималном стаблу позива. Последње активационо стабло одговара улазу за који важи да је `args` низ аргумената дужине 0, у ком случају `foreach` метода не позива ниједну од ламбди.

Табела 2.1 приказује преглед симбола и структура података дефинисаних у овој секцији.

2.2 Компилација

Као што је речено у уводу, поступак компилације подразумева превођење целокупног изворног програмског кода у извршни код, који потом може исправно да се изврши на циљној машини. Програмска целина се током превођења дели у засебне јединице компилације. Иако компилациона јединица представља широк појам јер, у зависности од изворног програмског језика и преводиоца, може обухватати више појмова, у даљем тексту под појмом *компилациона јединица* (енг. *compilation unit*) подразумева се метода која се компајлира укључујући целокупан код који након процеса превођења представља део те методе (иако то можда није био пре процеса превођења). *Распоред превођења*

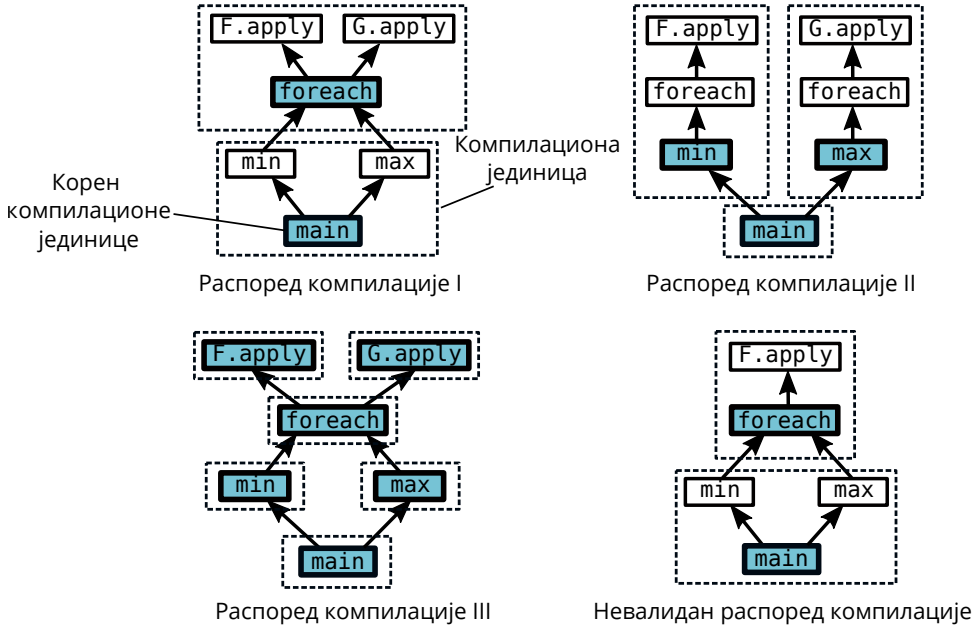
(енг. *compilation schedule*) моделује се усмереним графом у коме сваки чвор одговара јединици компилације, а свака усмерена грана одговара позиву између две компилационе јединице. Нпр. уколико компилациона јединица A позива компилациону јединицу B, у одговарајућем графу распореда превођења ће постојати грана од чвора који одговара компилационој јединици A до чвора који одговара компилационој јединици B.

Компилациона јединица A представља методу S_A програма који се преводи, укључујући све методе чије је тело уграђено у методу A током оптимизације инлајновања. Каже се да компилациона јединица A *позива* другу компилациону јединицу B уколико постоји локација у коду методе A са које се позива метода S_B, која је корен компилационе јединице B. Распоред компилације је *валидан* уколико постоји компилациона јединица чији корен одговара улазној методи програма (важи за сваку улазну методу програма ако их има више) и уколико постоји усмерена грана за сваки позив између две компилационе јединице. *Буџет компилације* (енг. *compilation budget*) је функција која додељује одређену количину ресурса за израчунавање, коју неки оптимизациони компајлер може потрошити када преводи одређену компилациону јединицу.

Пример. Слика 2.5 садржи неколико могућих излаза алгоритма компилације који је примењен над улазним програмским кодом из листинга 2.1. Сваки од четири приказана распореда компилације анотиран је на следећи начин. Компилационе јединице су оивичене испрекиданим затвореним линијама, а све корене методе јединица обојене су плавом бојом. Преостале методе су инлајноване и на слици су обојене белом бојом. Распоред превођења I састоји се од две компилационе јединице – компилациона јединица са кореном методом `main` обухвата методе `min` и `max` инлајноване у методу `main`, а свака од ове две методе позива другу компилациону јеидницу са кореном методом `foreach`. Како се друга компилациона јединица позива из два различита контекста, у `foreach` методу се полиморфно инлајнују обе имплементације функције `apply` [4]. У распореду превођења II, компилациона јединица која се односи на `main` не укључује ниједну функцију која се позива из `main` методе, па `min` и `max` постају корени две засебне компилационе јединице, где свака инлајнује засебну копију `foreach` методе. Предност другог распореда је та да `foreach` може инлајновати тачно једну ламбда функцију и тиме, за разлику од распореда I, избећи испитивање типа објекта над којим ће се позвати метода (енг. *type-check*), на основу кога ће се извршити одговарајућа ламбда. У распореду превођења III не долази до инлајновања, па се свака компилациона јединица састоји само од по једне методе. Сви примери распореда компилације наведени до сада су валидни. Последњи распоред компилације није валидан зато што одговарајући граф не укључује грану за позив функције `G.apply`, који може да се оствари током извршавања програма.

2.3 Профили

Једну од најранијих употреба профила описао је Кнут [51], који је дефинисао *профил* (енг. *profile*) као скуп бројних вредности о извршавању програма прикупљених у току тог извршавања. Током времена, појам профила се проширио тако да укључује било



Слика 2.5: Примери излаза алгоритма за превођење

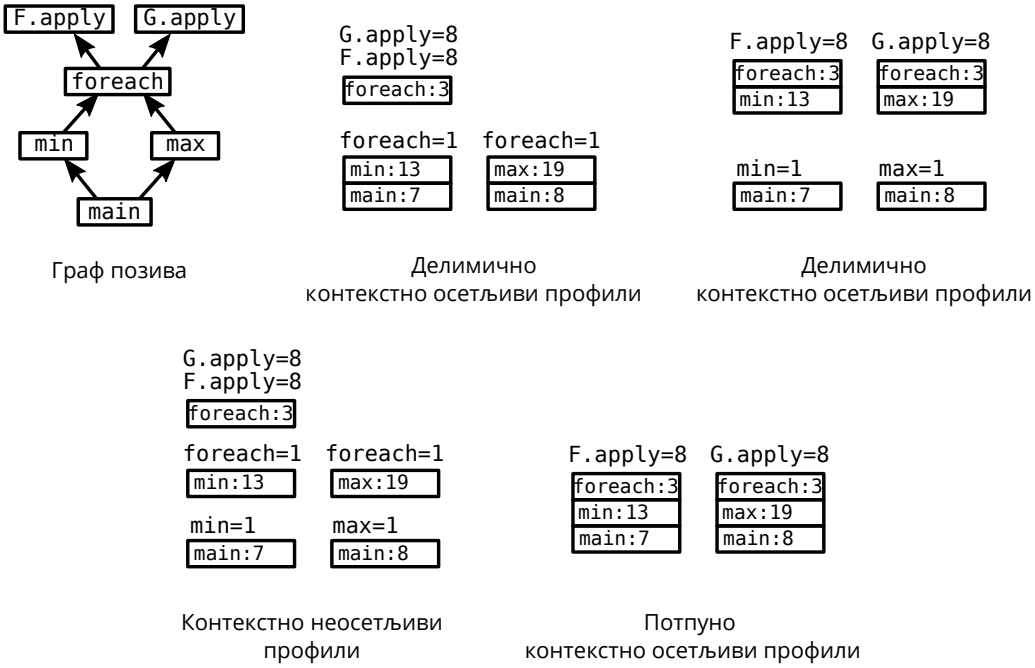
које метрике које описују понашање програма и које су прикупљење током извршавања тог програма. Увођење процеса прикупљања профила, тј. профилисања, довело је до питања како профили могу бити искоришћени у циљу оптимизације извршавања програма, које информације профили треба да садрже, која је цена прикупљања профила, као и како та цена може бити смањена [52].

2.3.1 Контекстна осетљивост профила

Позиција у коду догађаја који се профилише бележи се у оквиру контекста профила. У односу на број локација садржаних у контекстима, профили могу бити шире категоризовани као контекстно осетљиви или контекстно неосетљиви. Код потпуно контекстно осетљивих профила, улазни профили садрже све локације са програмског стека, које су активне приликом прикупљања тог профила. Уколико контекст садржи само неки суфикс са стека позива, такав контекст се сматра парцијалним.

Граф позива који одговара примеру 2.1 дат је на слици 2.6. У наставку ће бити анализирано неколико профила који прате број извршавања свих метода која могу бити позване на одређеној локацији у коду. Сваки контекст је приказан као стек који садржи парове локација у коду, где је свака локација означена паром метода - број линије. Контексти у примеру су везани за метрику која представља број извршавања сваке конкретне методе која се позива са дате локације, и у датом контексту. Две ламбда функције које користе методе `min` и `max` назване су `F` и `G`, респективно.

Први делимично осетљиви контекст „сече" стек позива у оквиру `min` и `max` метода, тако да метода `foreach` позива обе ламбда функције `F` и `G`. У другом парцијалном профилу из примера, контексти су „пресечени" у `main` методи, што чини два `foreach` контекста прецизнијим – сада је јасно да `F` може бити позвано искључиво када је метода `foreach` позвана из `min`, односно `G` може бити позвано искључиво када је метода



Слика 2.6: Контекстно осетљиви и контекстно неосетљиви профили

`foreach` позвана из `max`. На основу трећег примера, који приказује контекстно неосетљиве профиле, може се закључити да они не садрже мање информација, у погледу позива ламбди, у односу на први пример у коме се налазе парцијални профили. Наиме, позив методе `min` на локацији `main:7` је директан, што значи да са ове локације у програмском коду може бити позвана једино ова метода, а у овом једноставном примеру, то је, такође и једина локација позива за ову методу. Самим тим улаз профила који садржи контекст `min:13`, а који није контекстно осетљив, може бити продужен контекстом `main:7`. Слично се може применити и над контекстима `max:19` и `main:8`, чиме се формирају парцијални конктексти (садрже више од једне локације) који обухватају идентичну информацију о извршавању. Слично, потпуно контекстно осетљиви профили, приказани у последњем примеру, не уводе више информација у односу на парцијалне профиле приказане у другом примеру са слике.

2.3.2 Технике прикупљања профила

У зависности од тога на којој се техници заснива, поступак прикупљања профила може бити категоризован као процес инструментације (енг. *instrumentation-based profiling*) или процес прикупљања узорака (енг. *sampling-based profiling*). Техника инструментације подразумева уграђивање делова кода намењених прикупљању профила у програмски код чије се понашање прати. Ово уграђивање може се реализовати на различитим нивоима, попут међурепрезентације коју неки преводилац формира на основу улазног програмског кода или директно унутар извршног кода. Најчешће, додатни инструментациони код у финалном извршном коду садржи инструкције за увећавање вредности бројача, који бележе извршавање догађаја од интереса у програму. Захваљујући овоме, примена потпуне инструментације (енг. *exhaustive instrumentation*), тј.

инструментације над комплетним програмским кодом омогућава прецизне метрике за све локације програма које се прате.

Профилисање применом стратегије прикупљања узорака уведено је да смањи трошкове профилисања, посебно у *online* окружењима [32, 33, 53]. Ова стратегија подразумева периодично снимање садржаја стека позива функција током извршавања програма. Значајан број често коришћених алата за профилисање доминантно користи ову технику. Прикупљање узорака може бити реализовано на више начина. Већина алата за узорковање Java програма прикупљају узорке у тренуцима извршавања програма одређеним механизмом сигурних тачака (енг. *safepoint*). Када се покрене овај механизам, безбедно је снимити тренутни садржај стека позива. Недостатак оваквог приступа лежи у томе што се безбедне тачке за заустављање програма не окидају у равноправним интервалима током извршавања програма, што за последицу може имати „пристрасне" профиле [54, 55]. Другим речима, однос тако прикупљених профила не мора реално осликавати време проведено у различитим сегментима кода, већ зависи од распореда момената узороковања. Како се профили користе за апроксимацију времена проведеног у одређеним деловима кода, ово може довести до неисправне идентификације значајних секција. Mytkowicz и др. [56] спровели су анализу неких од најкоришћенијих алата за профилисање Java програма као што су `xprof` [57], `hprof` [58], *JProfiler* [59] и *YourKit* [60] како би показали претходно наведене мане неких од њих. GNU GCC алат за профилисање `gprof` [61] намењен је за прикупљање профила над C и C++ програмима. Овај алат користи тајмер оперативног система како би прикупљао узорке у правилним размацима, па тачност профила није нарушена избором тренутка прикупљања.

Велики број аутора истиче прецизност профила добијених применом технике инструментације наспрам профила добијених применом технике узорковања [18, 62]. Неке верзије алата Netbeans IDE [63] и Eclipse Test and Performance Tool Platform [64] укључују и алате за прикупљање профила инструментацијом. Ипак, имајући у виду да извршавање додатног инструментационог кода може значајно утицати на време потребно да се програм изврши, стратегија инструментације у највећем броју случајева није повољна за превођење у току извршавања (енг. *online collection*) [65]. Самим тим, ова техника се доминантно користи код превођења пре времена извршавања (енг. *offline collection*). Према ауторима Arnold и др. [35], инструментација може деградирати перформансе програма од 30% па чак до 1000%. *GraalVM Native Image* окружење, у којем је имплементиран алгоритам из ове тезе, подразумева *offline* прикупљање профила, због чега је употреба инструментације прихватљива како би се прикупиле прецизне метрике догађаја који се прате.

Подела на *online* и *offline* технике прикупљања профила представља други метод на основу кога се алати могу класификовати [66]. Офлајн прикупљање профила подразумева извршавање засебне инстанце програма намењене праћењу метрика, након које следи процес превођења и извршавања програма, који користи профиле прикупљене у првој фази. AOT преводиоци углавном користе овакву шему. Један од њих је и AOT Scala Native преводилац [26]. У једном режиму рада, GCC преводилац ствара инструментовани извршни код, а профиле добијене његовим извршавањем користи приликом формирања оптимизованог извршног кода [23]. Онлајн прикупљање профила се обавља

у истом процесу извршавања програма у коме се и дати програм оптимизује употребом тих истих профила. Ова парадигма се углавном спроводи у оквиру JIT преводилаца. За разлику од алата `gprof`, који се примарно користи за прикупљање профила AOT преведених програма, већина профајлера, попут алата *Valgrind* [67], *Java Mission Control* [68], *JProfiler*, *YourKit*, `hprof` и `perf` нису стриктно везани ни за један од два режима компилације (AOT или JIT). Другим речима, користе се за прикупљање профила програма преведених на оба начина.

2.3.3 Стратегија прикупљања профила у преводиоцу *GraalVM Native Image*

На виртуелној машини HotSpot програм се профилише током интерпретирања, које подразумева прву фазу превођења, пре JIT компилације [28, 29, 30, 31]. Како се оптимизација инлајновања, која по дефиницији уводи контекстну осетљивост у компилационе јединице, спроводи током JIT компилације, профили прикупљени у интерпретеру су конктекстно неосетљиви, тј. њихови контексти садрже само једну локацију. Дугим речима, интерпретер одржава исти скуп бројача за неку методу независно од тога са које локације у програму је дата метода позвана. У случају AOT компилације, целокупан програм се преводи пре него што почне да се извршава, а интерпретер типично није доступан. Из ових разлога, *GraalVM Native Image* подржава два мода превођења – формирање инструментационог извршног фајла (енг. *instrumentation image*), где је извршни код допуњен наменским кодом за прикупљање профила, и формирање оптимизованог извршног фајла (енг. *optimization image*), где је извршни код оптимизован на основу профила и у највећем броју случајева не садржи никакав код за профилисање.

Програм се на почетку преводи у инструментациони извршни код, који се извршава једном (или више пута) како би се прикупили профили за тај програм. Профили се уписују у датотеку када се извршавање иснтрументованог програма заврши. Затим се програм преводи у оптимизовани извршни код, који опционо користи једну или више датотека са профилима. Учитани профили се потом користе за унапређење компајлерских оптимизација. Уколико датотека није учитана, током превођења оптимизације неће користити профиле у оквиру својих хеуристика. Генерисање наменског инструментационог извршног кода је уобичајен приступ за *offline* профилисање [23, 26, 14], као и употреба тако добијених профила како би се генерисао оптимизовани извршни код са циљем унапређења његових перформанси.

Типови профила. Посебне инструкције које су од интереса у програмима који се профилишу називају се догађаји (енг. *events*). Како би се смањио утицај поступка профилисања на перформансе инструментационог извршног кода, већина виртуелних машина и преводилаца прате само оне догађаје који се сматрају корисним за унапређење ефеката компајлерских оптимизација. *GraalVM Native image* прикупља профиле за три различита типа догађаја:

- **Извршавање метода.** Бележи се број улазака у методу M, (енг. *method hotness*). Тело методе M може бити уграђено у неку другу методу, уколико је дошло до

њеног инлајновања. На пример, метода M може бити инлајнована у оквиру контекста C, и у том случају овај тип догађаја се односи на број извршавања методе M у специфичном случају када је она инлајнована у оквиру контекста C.

- **Контролне структуре.** Бележи се број извршавања сваке гране неког условног гранања (`if` или `switch` исказа). Свака грана је повезана са локацијом у бајткоду у оквиру методе M, а метода, у општем случају, може бити инлајнована у неком контексту C.

- **Виртуелни позиви.** Чува се низ бројача, који одговарају могућим типовима објекта над којим се може позвати дата виртуелна метода (енг. *receiver type*). За сваки тип, виртуелна метода се разрешава у конкретну циљну методу позива (енг. *target method*). Низ бројача памти број извршавања одговарајућих конкретних циљних метода за сваки тип који је снимљен у профилима. Локација виртуелног позива повезана је са позицијом у бајткоду у оквиру методе M у којој се налази сам позив, а та метода, на сличан начин као за претходне типове догађаја, може бити инлајнована у контекст C.

Како се на истој линији изворног програмског кода може наћи више догађаја, није довољно да појединачне локације контекста неког профила садрже методу и линију у коду. Уместо тога, за прецизније позиционирање у оквиру методе, потребно је користити индекс у бајткоду (енг. *bytecode index*, скр. *BCI*). Java бајткод за сваку методу представља листу инструкција [69], такву да свака инструкција има јединствен индекс. Другим речима, BCI представља померај дате инструкције у односу на почетак методе у оквиру које се налази изражено у бајтовима. За неку методу, индекси у бајткоду почињу од 0 и иду све до вредности индекса последње инструкције те методе.

2.4 Употреба профила у процесу превођења

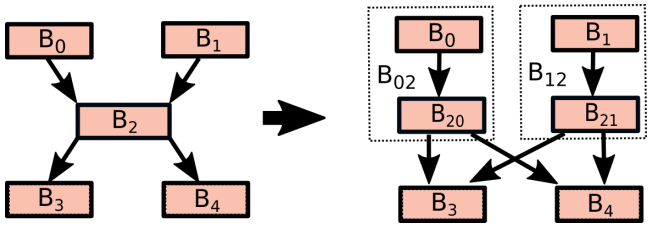
Како би генерисао што ефикаснији код на основу улазног програма, преводилац га анализира и спроводи низ оптимизационих пролаза. Већина оптимизација доноси боље одлуке када њихове хеуристике користе информације о извршавању програма, тј. динамичке информације, уместо искључиво статичких информација о програму. Када је реч о превођењу програма пре његовог извршавања, динамичке информације о програму се обезбеђују употребом профила. Многе компајлерске оптимизације попут инлајновања метода [9], смештања блокова [70], одмотавања петљи и других [71] користе профиле. У овој секцији биће детаљније објашњене неке од ових оптимизација са посебним нагласком на корист коју оне имају од употребе профила при доношњу одлука.

2.4.1 Поступак дупликације кода

Како су неке компајлерске оптимизације ограничене унутар базичних блокова методе, чворови који спајају различите путање у графу тока контроле (енг. *merge nodes*) онемогућавају примену ових оптимизација или умањују њихово дејство [19]. Елиминацијом ових чворова долази до повећања базичних блокова који им претходе, чиме је

могуће очувати информације о току података, који могу утицати на успешност примена оптимизација које следе. Уклањање чвора спајања имплицитно подразумева дупликацију инструкција базичног блока који следи тај чвор у оригиналном графу (пре примене трансформације). Дупликација кода омогућава даљу специјализацију секције кода у оквиру неке методе применом оптимизација над том копијом кода независно од оптимизовања осталих копија исте секције кода [72].

Слика 2.7 садржи граф тока контроле дела програма пре и након примене трансформације дупликације кода. Базични блокови B_0 и B_1 претходе спајању тока контроле који припада блоку B_2. На крају базичног блока B_2 постоји операција гранања, која даље раздваја ток контроле на блокове B_3 и B_4. Пример са слике илуструје елиминацију чвора спајања дупликацијом операција базичног блока B_2, чиме се добијају блокови B_{20} и B_{21}. Како у примеру не постоје операције које уводе друге токове контроле, операције блока B_{20} припадају блоку B_0, односно операције блока B_{21} се могу придружити блоку B_1 (на слици означено испрекиданим правоугаоницима). Исправност програма мора бити очувана, па се извршавање програма након било ког од резултујућих блокова B_{02} и B_{12} може наставити у блоковима следбеницима B_3 и B_4 из оригиналног програмског кода.



Слика 2.7: Пример трансформације графа дупликацијом кода

Неке од оптимизација које дупликација може омогућити су:

- *Елиминација редунданшних йрисшуūа мемоūији* [73, 74]. Могуће је уклонити редундантне уписе у меморију и читање из меморије. Ово је посебно значајно зато што комуникација са меморијом траје дуже од аритметичких операција.
- *Елиминација кондиционала (енī. conditional elimination)* [75]. Уколико се докаже да је неки услов испуњен, његово испитивање може бити уклоњено.
- *Девиūшуелизација* [4, 5]. Виртуелне позиве је могуће специјализовати уколико је познат тип објекта над којим се метода позива.
- *Уīрађивање консшанши (енī. constant folding)* [76]. Употребу променљивих могуће је елиминисати уколико се докаже да увек имају константу вредност у специјализованом коду.

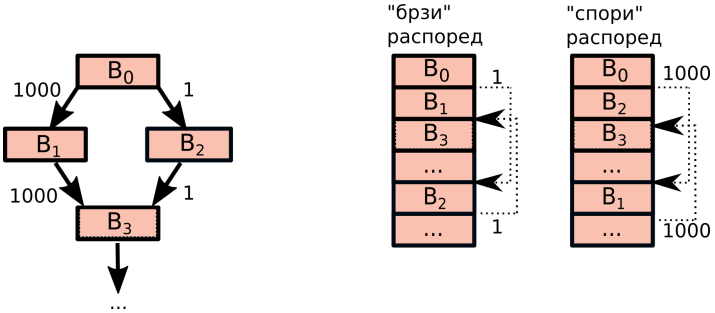
Како елиминација свих чворова који спајају токове контроле доводи до експоненцијалног увећања генерисаног кода, дупликација се мора спроводити селективно. Док се једноставне статичке хеуристике не показују применљивим у пракси обзиром на значајан, иако не експоненцијални раст кода, посебну улогу игра примена профила како би се

идентификовале секције кода над којима има смисла извршити трансформацију [19]. На основу профила могуће је идентификовати често извршаване делове кода над којима има смисла агресивније примењивати дупликацију. Како се показује и у овом раду, проценат таквих делова кода није велики и његово повећање не доводи до значајног повећања укупног генерисаног кода.

2.4.2 Оптимизација смештања блокова

Свака контролна структура *if* гранања у коду има два базична блока потомка. Један потомак је онај на који постоји директан скок уколико је неки услов задовољен, а други је онај на који се директно прелази „пропадањем" уколико услов скока није задовољен. Цена ових гранања је различита у већини процесора, па се оптимизација смештања блокова ослања на идеју да је извршавање суседног блока пропадањем ефикасније и у складу са тим распоређује базичне блокове на гранањима.

На слици 2.8 приказан је граф базичних блокова, који илуструје претходно описану ситуацију. Гранање на крају базичног блока B_0 показује асиметричну фреквенцију извршавања. Грана која води до блока B_1 извршава се 1000 пута чешће у односу на грану која води до блока B_2. Како је пропадање у суседну инструкцију генерисаног кода ефикасније од операције скока, оптимизација треба да смести блок који се чешће извршава суседно базичном блоку са гранањем. У конкретном случају, након блока B_0 биће смештен блок B_1, док ће блок B_2 бити смештен негде даље у генерисаном коду.



Слика 2.8: Пример оптимизације смештања кода

Слика 2.8 илуструје два распореда блокова која одговарају графу тока контроле из истог примера. Први распоред блокова је такозвани „брзи" распоред и одговара претходно описаном смештању блокова. Између блокова B_0 и B_1 уграђена је инструкција скока на блок B_2, која се извршава једном на сваких 1000 извршавања пропадања између блокова B_0 и B_1. Како између блокова B_1 и B_3 постоји директно пропадање, блок B_3 се смешта непосредно након блока B_1. На крају блока B_2 генерише се скок на B_3. „Спорији" распоред са десне стране подразумева другачије смештање блокова. Након блока B_0 смешта се блок B_2, а уграђује скок ка удаљеном блоку B_1. Пропадање кроз блокове ће се у овом случају реализовати само једном у поређењу са 1000 скокова на блок B_1 и скокова из овог блока на B_3.

Спровођење ове оптимизације зависи од информације коју преводилац има о фреквенцији извршавања гранања. JIT преводиоци дату информацију добијају на основу

првог стадијума извршавања, који најчешће подразумева интерпретирање, док AOT преводиоци морају имати доступне профиле који се односе на извршавање гранања, тј. засебних базичних блокова.

Исплативост оптимизације зависи од фреквенција гранања конкретне контролне структуре у односу на различиту цену прелазака у наредни базични блок. Уколико је фреквенција извршавања обе гране приближно једнака, тада ће цена смештања оба блока бити приближно једнака у општем случају и нема значајније користи од примене ове оптимизације, а са друге стране постоје режијски трошкови њене имплементације.

Осим размештања блокова самог гранања, применом сличне логике, оптимизација утиче на распоређивање свих базичних блокова неке методе. На основу фреквенција извршавања појединачних блокова методе, одређује се „највероватнија" путања као низ блокова који су највероватнији за извршавање. Такви блокови се у одређеном распореду смештају суседно како би се обезбедило пропадање између њих. Затим се преостали блокови смештају иза већ распоређених блокова уз додавање инструкција скокова, које ће одржати исправно извршавање методе.

2.4.3 Оптимизације одмотавања петљи

Одмотавање петљи представља једну од најчешћих оптимизација над петљама. Ова оптимизација повећава тело петље његовом репликацијом k пута, а како би се очувао укупан број операција тела оригиналне петље, умањује број итерација петље k пута. Пример једноставне *for* петље са N итерација дата је у листингу 2.2. Еквивалентна петља над којом је примењена оптимизација одмотавања са фактором k приказана је у листингу 2.3.

Умањењем броја итерација петље смањују се и режијски трошкови петље, тј. елиминише се значајан број испитивања услова петље, као и операција инкрементирања бројача петље. Са друге стране повећање тела петље директно доводи до увећања кода на два нивоа. Повећање генерисаног кода може додатно утицати и на инструкцијски кеш, док повећање кода на нивоу међурепрезентације програма, коју преводилац одржава и над којом се ова оптимизација најчешће спроводи, може додатно продужити време превођења програма.

Осим директних бенефита и цена које ова оптимизација уводи, постоје и индиректне последице њене примене. Повећање величине базичних блокова тела петље може утицати на хеуристике оптимизација које се током превођења спроводе након одмотавања. Повећање броја међусобно независних операција тела петље може довести до бољег распоређивања инструкција и искоришћења функционалних јединица.

Како примена оптимизације није увек исплатива, погодно је примењивати је агресивније над петљама које се често извршавају. Прецизније говорећи, могуће је користити профиле како би се идентификовале петље значајне за извршавање програма и потом над таквим петљама спровела оптимизација са већим фактором одмотавања. Над преосталим петљама, које се на основу профила не идентификују као значајне, може бити спроведено одмотавање са нижим фактором или оптимизација може бити у потпуности елиминисана.

```
for (i = 0; i < N; i++) {
  Op_i : ...
}
```

Листинг 2.2: Петља пре примене оптимизације одмотавања

```
for (i = 0; i < N/k; i+=k) {
  Op_{i,0} : ...
  Op_{i,1} : ...
  Op_{i,2} : ...
  Op_{i,3} : ...
}
```

Листинг 2.3: Петља након примене оптимизације одмотавања

2.5 Изазови оптимизације инлајновања

Инлајновање [77, 78, 79] је оптимизација замене позива неке методе копијом кода која одговара тој методи. Како се одлука о примени оптимизације доноси наменски, за свако место позива у програму (енг. *callsite*), не постоји оптималан алгоритам инлајновања који би доносио најбоље одлуке за све програме и преводиоце. Значајан број алгоритма за инлајновање подразумевају хеуристичка решења која укључују различите анализе користи и цене сваке примене оптимизације (енг. *const-beneffit analysis*) [5, 10, 80, 81, 82, 78, 83, 84, 85, 9]. Док неки инлајнери доминантно користе статичке информације о програму као што је величина метода [86], употреба профила у највећем броју случајева доводи до знатно бољих перформанси.

У оквиру ове секције биће приказано неколико шаблона кода који представљају посебне изазове алгоритмима инлајновања. Акценат ће бити на објектно оријентисаним и функционалним програмским језицима. Циљ је илустровати неке од проблема са којима се суочавају инлајнери, посматрано из перспективе програмера, као и омогућити умањење ефеката ових препрека техникама које би програмер могао да примени, када сам алгоритам за инлајновање није довољно софистициран да то учини аутоматски. Примери не покривају све изазове, а и предложене технике нису исцрпне, већ су засноване на личном искуству аутора истраживања у раду са инлајнером и анализама перформанси различитих програма.

2.5.1 „Загађене" фреквенције профила

Како су одлуке које инлајнер доноси типично усмерене на основу фреквенција из профила, чест проблем је непрецизност информација коју профили могу да садрже. Узмимо за пример профиле који се односе на неки конструкт тока контроле попут петље. Ако је метода у оквиру које се налази петља позвана са више различитих програмских локација, тада број извршавања петље може варирати у зависности од локације позива. Контекстно неосетљив профил ће садржати обједињену вредност фреквенције извршавања за све позиваоце. У таквим случајевима сматра се да су фреквенције про-

фила „загађене” (енг. *polluted profile*). И делимично контекстно осетљив профил може садржати непрецизне фреквенције извршавања уколико контекст није довољно дугачак да једнозначно омогући дистинкцију између локација позива које доводе до другачијег понашања петље. Ако „загађеност” профила резултује лошом проценом фреквенције позива, могуће је да инлајновање методе неће бити спроведено над неким позивом критичним за перформансе програма.

Пример. Листинг 2.4 приказује имплементацију алгоритма за сортирање Quicksort написаног у програмском језику Scala. Имплементација се састоји од две функције `sort` и `part`. Функција `part` дели неку секвенцу коришћењем последњег елемента као пивота и смешта елементе са мањом вредношћу лево од коначне позиције пивота, а елементе са већом вредношћу на позиције десно. Функција `sort` уређује прослеђену секвенцу тако што позива функцију `part`, а потом рекурзивно позива функцију `sort` над добијеним партицијама лево и десно од коначне позиције пивота.

`While` петља у оквиру `part` функције позвана је много пута рекурзивно, и то сваки следећи пут над краћим секвенцама. Како је већина позива функције `part` близу листовима рекурзивног стабла позива, просечна фреквенција петље је пристрасна према краћим петљама. Због тога, инлајнер може да одлучи да не инлајнује позив функције `swap` унутар тела петље, што значајно мења перформансе извршног кода.

Решење. Програмер може вештачки да уведе контекстну осетљивост у код копирајући делове кода зависно од контекста у ком се извршавају. У конкретном примеру из листинга 2.4, програмер може поновити `while` петљу у оквиру две гране, раздвајајући их на основу услова као што је `hi - lo < THRESHOLD`. Ово обезбеђује да за две копије петљи буду формирани засебни профили, што за последицу има да инлајнер може да инлајнује позив функције `swap` бар на једном месту у оквиру петље која се извршава чешће. Алтернативно, програмер може декларисати функцију `part` као C или Scala

```
def sort(xs: Array[Int], lo: Int, hi: Int) {
  if (lo >= hi || lo < 0) return
  val p = part(xs, lo, hi)
  sort(xs, lo, p - 1)
  sort(xs, p + 1, hi)
}
def part(xs: Array[Int], lo: Int, hi: Int) = {
  var pIndex = lo - 1
  var j = lo
  while (j <= hi - 1) {
    if (xs(j) <= xs(hi)) {
      pIndex += 1
      swap(xs, pIndex, j)
    }
    j += 1
  }
  swap(xs, pIndex + 1, hi)
  return pIndex + 1
}
```

polluted frequency profile

affected inlining decision

Листинг 2.4: Quicksort алгоритам

макро или C++ шаблон (енг. *template*), а затим инстанцирати тело два пута позивајући другу инстанцу у зависности од разлике између `hi` и `lo`.

2.5.2 „Загађени" профили позива виртуелних метода

Како би се инлајновао индиректни позив (позив виртуелне методе), инлајнер прави претпоставку која ће циљна метода, тј. имплементација виртуелне методе бити заправо позвана. Инлајнер бира неколико највероватнијих циљних метода, емитује код који врши проверу циљне адресе, тј. типа објекта чија се метода позива, а затим инлајнује сваку од тих конкретних циљних метода у оквиру гране која одговара провери типа одговарајућих објеката. Динамичко разрешавање позива виртуелне методе ће тако бити елимисано за највероватније типове пријемника на некој локацији. Успех ове технике тако зависи од квалитета информација о вероватноћама типова објеката чија ће се метода позвати, тј. од профила позива виртуелних метода. Ако је профил сувише нетачан, претпоставке направљене на темељу тог профила ће бити неисправне. Овом трансформацијом кода неће бити нарушена исправност извршавања, али ће се повећати број неуспешних испитивања типа објекта над којим се позива метода, а затим ће свакако бити извршен индиректни позив.

Пример. Свака колекција у стандарној библиотеци програмског језика *Scala* садржи функцију `foreach` која примењује кориснички дефинисану функцију над сваким елементом те колекције. Пример из листинга 2.5 приказује `foreach` функцију која одговара колекцији `HashSet`. Функција итерира кроз низ и примењује кориснички дефинисану функцију `f` над сваким постојећим елементом низа. Позив функције `f` је индиректан зато што конкретна имплементација није позната. Како би се девиртуелизовао позив функције `f`, инлајнер се ослања на профиле позива виртуелних метода, који садрже типове пријемника за конкретан позив и учестаности сваког од њих. Како је `foreach` једна од најчешће коришћених операција над колекцијама у програмском језику *Scala*, профили позива виртуелних метода ће по правилу бити загађени осим у случају најједноставнијих програмских кодова.

```
def foreach[U](f: A => U) {
  var i = 0
  while (i < table.length) {
    val entry = table(i)
    if (entry ne null) f(elem(entry))
    i += 1
  }
}
```

polluted receiver-type profile

Листинг 2.5: Scala функција `HashSet#foreach`

Решење. Програмер може утицати на поправљање квалитета профила који се односе на позиве виртуелних метода. Један приступ се односи на употребу метапрограмирања приликом дизајнирања библиотеке тако да се често позиване функције (као што је `foreach`) имплементирају као макрои у програмском језику *Scala* или као шаблони у

програмском језику *C++*, што ће резултовати инстанцирањем кода на свакој локацији позива. Ако је позив функције `foreach` индиректан, корисник може ручно уметнути проверу типа пријемника, за све типове који се очекују на конкретној локацији позива, а потом претворити пријемник у конкретан тип пре позива функције као што је `foreach`. У примеру из листинга 2.5, корисник такође може и искористити итератор и `while` петљу уместо `foreach` функције. На тај начин се елимише индиректан позив функције `f` или се креира засебна локација позива за коју се могу прикупити профили, што за последицу има смањење загађености профила.

2.5.3 Инлајновање угњеждених полиморфних позива

Претходно описана техника девиртуелизације позива на основу профила елиминише индиректне позиве инлајновањем неколико највероватнијих имплементација одговарајуће виртуелне методе. Која ће се инлајнована имплементација извршити, зависи од резултата провере типа пријемника, реализоване у оквиру генерисаног гранања за сваки од највероватнијих типова пријемника. Број највероватнијих типова објеката који одговарају виртуелној методи је углавном мали, уобичајено до три, али упркос томе инлајновање свих имплементација које одговарају овим објектима може узроковати значајно увећање величине кода посебно са увећањем дубине полиморфних позива.

Како сваки ниво у стаблу инлајновања ради предвиђање за више конкретних имплементација виртуелне методе која се може позвати на некој локацији, а оне даље могу садржати индиректне позиве, број комбинација инлајновања расте експоненцијално иако ће само неки мањи број ових путања бити заправо извршен. Већина инлајнера ограничава овакав раст кода лимитирајући величину јединице компилације. Увећање јединице спречава покушај инлајновања других делова стабла позива. Експлозија кода утиче и на друге оптимизације које узимају у обзир величину генерисаног кода.

Пример. Листинг 2.6 приказује имплементацију операције `foldLeft` интерфејса `TraversableOnce`. Овај интерфејс обухвата колекције дефинисане искључиво у контексту `foreach` функције. Операција `foldLeft` спаја елементе колекције с лева на десно, почев од нултог елемента `z`, примењујући операцију `op` над кумулативном („склопљеном", енг. *folded*) вредношћу колекције која је до сада процесирана и следећим појединачним елементом те колекције. Функција `foldLeft` позива `foreach` прослеђујући функцију која извршава операцију `op` и ажурира кумулативну вредност `result`. Како су оба позива `foreach` и `op` индиректни (први зато што се тип `this` објекта мења, а други зато што се кориснички дефинисан параметар `op` мења), неколико конкретних циљних функција се инлајнују за `foreach` позив, а затим за сваку тако инлајновану функцију, неколико конкретних циљних функција које одговарају позиву `op` се рекурзивно инлајнују.

Решење. Ако програмски језик подржава метапрограмирање, програмер може и у овом случају искористити макрое за инстанцирање објеката. У примеру из листинга 2.6, дефинисањем функције `foldLeft` као макро или шаблон пријемник позива `foreach` и `op` постаје видљив, па преводилац може конвертовати ове индиректне позиве у директне. Чак иако пријемници нису видљиви, инстанцирање кода операције `foldLeft`

```
def foldLeft[B](z: B)(op: (B, A) => B): B = {
  var result = z
  this.foreach(x => result = op(result, x))
  result
}
```
code-bloating nested indirect calls

Листинг 2.6: Scala функција `TraversableOnce#foldLeft`

на одређеној локацији позива креира нови догађај за профилисање, чиме се смањује загађеност профила.

2.5.4 Занемарени ретко извршавани позиви

Инлајнери су неретко веома пристрасни према често позиваним потпрограмима, што за последицу има да се инлајновањем оваквих позива буџет оптимизације потроши пре него што ретко позивани потпрограми дођу на ред. Иако у доста случајева ретко позивани потпрограми не утичу доминантно на перформансе целокупног програма, некада они садрже додатне информације, које могу бити искоришћене од стране оптимизација које следе. Самим тим, уколико инлајнер одлучи да не инлајнује такве позиве, то може довести до лошије примене оптимизација, а на крају и до слабијих перформанси целокупног програма.

Пример. Функција `aggregate` програмског језика *Scala* приказа у листингу 2.7 је операција за рад са колекцијама која генерализује функцију `foldLeft` и може имати паралелену имплементацију. Подразумевано, функција је имплементирана у контексту `foldLeft` у оквиру `TraversableOnce` интерфејса.

Потпис функције `aggregate` узима нулти елемент `z` као аргумент по имену, што значи да се његова вредност не евалуира одмах на локацији позива, већ се уместо тога прослеђује функција која представља израз за израчунавање. У оквиру `main` функције, користи се `aggregate` функција, која као повратну вредност има суму дужина свих аргумената. У овом примеру, нулти елемент `0` се не израчунава одмах, већ се, уместо тога, прослеђује као функција која враћа вредност `0`. Последично, чак ако дође до инлајновања функције `foldLeft` из листинга 2.6 у јединицу компилације `aggregate` и поједностављења у једноставну `while` петљу, позив функције која представља параметар `z` потенцијално неће бити инлајнован. Ако се параметар који се преноси по имену `z` не инлајнује, враћа се као запакована вредност типа `java.lang.Integer`. Запакована вредност касније захтева отпакивање и паковање у самој `while` петљи, што шкоди перформансама кода у коме се проводи доста времена. У случају инлајновања позива `z`, преводилац може једноставније применити друге оптимизације (нпр. анализу која елиминише потребу за паковањем вредности простих типова, (енг. *escape analysis*)).

Решење. Пројектанти библиотеке треба да избегавају употребу аргумената који се преноси по имену (као што су функције `=>B`, код којих долази до „лењог" израчунавања аргумената), а уместо тога треба да користе параметре који се преносе по вредности. У примеру из листинга 2.7, директна употреба `B` уместо `=>B` представљала би боље решење. Генерално говорећи, уколико је нискофреквентан позив директан, корисник

```
def aggregate[B](z: =>B)(
  seqop: (B, A) => B,
  combop: (B, B) => B
): B =
  foldLeft(z)(seqop)

def main(args: Array[String]) {
  aggregate(0)(_ + _.length, _ + _)
}
```

Листинг 2.7: Scala `TraversableOnce#aggregate`

може да захтева инлајновање овог позива употребом директиве `@inline` када је она доступна. Без употребе анализе тока у оквиру инлајнера, нема много других опција за корисника осим наведених.

2.5.5 Ретко позиване методе садрже често извршавани код

Још један уобичајени проблем који постоји код инлајнера чија се хеуристика базира на учестаности позива јесте инлајновање ретко позиваних потпрограма који у себи садрже често извршавани код, нпр. петљу са великим бројем итерација. Уколико такав позвани потпрограм добија параметре од позивалаца на основу којих је могуће спровести девиртуелизацију или *escape* анализу параметара алоцираних на локацији позива, тада спречавање инлајновања ових ретко извршаваних позива може значајно утицати на перформансе.

Пример. Функција `foldLeft` из листинга 2.6 демонстрира описани проблем. Позив функције `foreach` догађа се само једном у телу `foldLeft` функције, али сама `foreach` функција садржи петљу, чије се итерације извршавају значајан број пута и чије тело садржи неколико индиректних позива који укључују прослеђивање параметара.

Решење. За директне позиве, програмер поново може прибећи употреби концепата метапрограмирања, или анотације `@inline` када је то могуће. У примеру из листинга 2.6, корисно је да аутори библиотеке преклопе (енг. *override*) `foldLeft` за често коришћене колекције, тако да имплементације ове функције у тим колекцијама садже директно петљу уместо позива `foreach`. Други начин да се превазиђе употреба индиректних позива је да се у оквиру самог програмског језика претпостави конкретан тип објекта `this`. Другим речима, предњи део компајлера за програмски језик који подржава метапрограмирање може спекулативно да инстанцира све имплементације функције `foreach` преко макроа.

2.6 Преглед постојеће литературе

Ова секција садржи преглед постојеће литературе у области прикупљања и примене профила, са акцентом на употребу профилних информација како би се „потпомогли" процес превођења, хеуристике инлајнера и друге компајлерске оптимизације. Детаљан

осврт на релевантна и блиска решења као и поређење са имплементираним решењем у овом раду и повезана дискусија биће приказани у оквиру секције 9.4. Мотивација за овакву поделу је ниво детаља који је неопходан, а који ће бити уведен у наредним секцијама, како би се спровела дискусија о аспектима предложеног решења у односу на остала значајна решења из литературе.

2.6.1 Стратегије и класификација прикупљања профила

У секцији 2.3 описане су две главне стратегије прикупљања профила – инструментација и прикупљање узорака. Познато је да се применом технике инструментације добијају прецизнији профили који се односе на фреквенцију позивања и извршавања делова програмског кода, али по већој цени у односу на технику прикупљања узорака. Постоји више приступа које је могуће користити када је потребно направити компромис између прецизности и цене прикупљања.

Једно од компромисних решења комбинује обе стратегије. Осим постизања баланса између цене и прецизности, комбинација две технике има смисла и због примене профила у оптимизацијама. Профили добијени инструментацијом изражавају број извршавања профилисаног догађаја, не урачунавајући време проведено у оквиру датог сегмента кода. Са друге стране, уколико се неки догађај извршава дужи интервал времена, биће снимљен као део програмског стека вишеструко у току истог извршавања. На тај начин техника узорковања може дати реалнију слику трајања неког догађаја, док са друге стране може у потпуности занемарити догађај који се често извршава, али траје кратко (иако акумулирано време може бити значајно за перформансе програма). Traub и др. користе технику прикупљања узорака како би снимили понашање програма током одређеног периода времена, уз периодично примењивање инструментације [65]. Компромис је овде постигнут директно контролисањем периода током којег се уграђује код за инструментацију – чешћом применом инструментације побољшава се прецизност профила, али се повећава цена прикупљања. Jikes RVM [87, 19] такође користи обе технике прикупљања профила, тј. инструментацију и узорковање.

Arnold и др. користе другачију хибридну технику тако што помоћу бројача периодично инструментују код, а извршавање се пребацује између инструментованог и неинструментованог кода [35]. Постоји доста радова на тему профилисања путања (енг. *path profiling*) [34, 88, 89], где је главна идеја да се цена извршавања кода за инструментацију смањи смештањем појединачног бројача на сваку путању тока контроле програма уместо на сваку грану или позив догађаја који се прати.

Већина виртуелних машина преводи програме у току извршавања, а профиле прикупља у првој фази извршавања програма [16, 28, 29, 30, 31]. У ређим ситуацијама профили се прикупљају и у компајлираном коду [54]. Аутори Flückiger и др. [22] предложили су JIT превођење на два нивоа кода написаног у R програмском језику, које користи инструментацију на првом нивоу, како би се оптимизовао улазни програмски код, а потом прикупљање узорака оптимизованог кода, како би се извршила реоптимизација на другом нивоу специјализованијим кодом.

У оквиру неких проширења преводилаца GCC [18] и LLVM [90] могуће је користити профиле прикупљене од стране екстерних алата како би се потпомогле неке оптимиза-

ције током превођења. AOT *Scala Native* преводилац [26] подржава прикупљање профила инструментацијом и користи профиле у оптимизацијама попут девиртуализације, дупликације метода и смештања блокова.

2.6.2 Делимична контекстна осетљивост профила

Док примена исцрпне инструментације омогућава највећу прецизност, мањи број апликација заправо може имати користи од таквог нивоа прецизности – попут детекције упада (енг. *intrusion detection*) или дебаговања (енг. *debugging*) [36]. Неколико аутора је испитивало да ли некомплетне профилне информације могу бити довољно прецизне да би биле употребљиве у већини уобичајених оптимизација компајлера, као и на који начин могу бити апроксимиране [91, 92, 37]. Примећено је и да се прикупљањем потпуно контекстно осетљивих профила, који почињу од корена програма, не побољшава значајно квалитет оптимизација. Даље, чување ових профила повећава меморијско заузеће структуре података у којој су смештени профили. Због тога више аутора заступа став да је погодно имати дужински-ограничене контексте позивања [37, 38]. Ausiello и др. [39] прикупљају профиле чији су контексти позивања дужине највише k, коришћењем структуре података која се зове *шума k контекста позивања* (енг. *k-calling-context forest*, скр. kCCF) и која користи мање простора у односу на структуру података која би чувала потпуне контексте.

Serrano и Zhuang [40] прикупљају делимичне трагове позива праћењем трагова на хардверском нивоу (енг. *hardware-level tracing*) и користе их да реконструишу информације о контекстима позивања. У техници коју су предложили аутори, делимични трагови се спајају уколико садрже значајна преклапања. Детаљно поређење рада двојице аутора са решењем имплементираним у овој тези биће дато након представљања појединости предложеног алгоритма у секцији 9.4.

2.6.3 Употреба профила у детекцији често извршаваних делова кода и оптимизацији инлајновања

Употреба профила у домену унапређења оптимизационих одлука је широко распрострањена у разним преводиоцима [18, 19, 20, 21, 22, 23]. Бројне су оптимизације које имају корист од улазних профила попут дупликације путања [72], инлајновања [9, 10], алокације регистара [7] и померања кода [93]. Фокус ће у наставку бити на употреби профила у циљу детекције значајних делова кода, над којима има смисла уложити највише оптимизационих напора, као и унутар оптимизације инлајновања, обзиром на то да ове теме представљају срж овог истраживања.

Mytkowicz ет ал. [56] су користили детекцију често извршаваних делова кода као критеријум за евалуацију Java профајлера који раде по принципу прикупљања узорака. Ова метрика је значајна имајући у виду да ће погрешном идентификацијом често извршаваних делова кода значајан део буџета оптимизација бити потрошен на оптимизацију програмских целина које нису значајне за перформансе целокупног програма. Важност исправне детекције фреквентно извршаваних делова кода истакнута је од стране аутора Kistler и др. [94], у чијем се раду спроводи контуирано ре-оптимизовање

најчешће извршаваног кода. Како би се идентификовали често извршавани делови кода и инлајновали потпрограми на релевантним локацијама позива, Krintz [25] је проширио JikesRVM [21] омогућивши и *offline* и *online* прикупљање профила. Коришћењем комбинованих профила, аутор је анотирао често извршаване потпрограме и одговарајуће локације позива за инлајновање. У датом раду коришћени су бројачи позива метода како би се израчунала учестаност извршавања методе, а инлајновање на одређеним локацијама позива заснива се на употреби контекстно неосетљивих профила.

Бројне хеуристике алгоритама за инлајновање користе различите анализе користи и цене за инлајновање неког потпрограма на одређеној локацији позива [5, 10, 80, 81, 82, 78, 83, 84, 85, 9]. Док неки инлајнери користе искључиво статичке инфомације о програму [86], већина преводилаца се ослања на информације добијене током извршавања неког програма, тј. на динамичке информације. Arnold и група аутора [10] пореде перформансе неколико техника инлајновања. Примећено је да када су само статичке информације расположиве, неки потпрограм је или инлајнован на свим локацијама позива тог потпрограма, или ни на једној локацији, независно од фреквенција позивања сваке од тих локација. Други приступ који је евалуиран користи профиле како би израчунао корист конкретног инлајновања. Резултати су показали да више контекстно осетљивих информација дозвољава инлајнеру да доноси боље одлуке, што последично доводи до бољих перформанси програма. Алгоритам за глобално инлајновање описан у раду аутора Scheifler [78] даје предност инлајновању најчешће извршаваних потпрограма, притом обезбеђујући да глобално ограничење величине кода није нарушено – фреквенције контекстно неосетљивих позива су обезбеђене на основу профилисања програма.

Техника превођења предложења од стране аутора Suganuma и др. [95] ослања се на хибридни приступ прикупљања профила како би се детектовали најзначајнији делови програма у циљу њихове агресивније оптимизације. У раду се користе једноставне хеуристике засноване на сатичким информацијама како би се оптимизовале најмање методе, док се скупље и захтевније оптимизације спроводе селективно над често извршаваним секцијама програма. Њихова техника прикупљања узорака одлучује које често извршаване методе треба да буду компајлиране, а потом се ове методе инструментују како би се добили прецизније профилне информације о често извршаваним деловима кода. Профили које аутори овог рада прикупљају су контекстно неосетљиви, што утиче на њихову прецизност.

Оптимизацију инлајновања „ометају" индиректни позиви, који су уобичајени у објектно оријентисаним и функционалним програмским језицима. Како није позната конкретна функција која се позива, инлајновање не може бити прецизно изведено. Овај проблем је ублажен прикупљањем профила за типове пријемника за сваку локацију индиректног позива. Као што су описали Grove и др. [96] и Hölzle и др. [4], полиморфно инлајновање може значајно побољшати перформансе програма. Ипак, полиморфно инлајновање биће онолико успешно колико и профили на чијим темељима је спроведено. На већим програмима, у случају контекстно неосетљивих профила, примећено је да профили типа пријемника могу бити доста „загађени". Експеримент спроведен од стране аутора Grove и др. [96] показују да контекстно осетљиви профили који одгова-

рају локацијама индиректних позива могу значајно да побољшају перформансе већих програма.

Уместо пројектовања алгоритма са фиксним вредностима параметара за све улазне програме, Cooper и др. [97] предлажу алгоритам за инлајновање који подразумева специфичне одлуке за сваки програм. Они описују систем који адаптивно тражи вредности параметара за неки програм ефикасно претражујући простор параметара. У раду је приказано како хеуристика направљена специфично за програм у комбинацији са вредностима параметара одабраним на исти начин показују најбоље перформансе инлајнера који их користи.

Аутори Møller и Veileborg [98] презентују статичку анализу алгоритма предвиђених за оптимизацију специфичне библиотеке JDK 8 Streams. Иако алгоритам имплементиран и описан у оквиру ове тезе тежи да побољша перформансе програма генерално, то такође укључује и програме који интерагују са библиотеком JDK 8 Streams, па ће поређење ова два рада бити дато у секцији 9.4.

3 Екосистем *GraalVM*

Ово поглавље садржи информације о екосистему *GraalVM* [99], у оквиру кога је имплементиран предложени алгоритам за унапређење превођења и оптимизације инлајновања. У секцији 3.1 приказане су главне компоненте система и у кратким цртама је објашњено како оне функционишу. Секција 3.2 у фокус ставља компоненту система *GraalVM* која обухвата AOT преводилац. Детаљи међурепрезентације компајлера *Graal*, која је значајна за разумевање имплементације, као и улаза предложеног алгоритма, приказани су у секцији 3.3. У секцији 3.4 дате су специфичности система значајне за оптимизацију инлајновања, а у секцији 3.5 појединости прикупљања и употребе профила.

3.1 Структура система *GraalVM*

Систем *GraalVM* [99] омогућава превођење улазног програмског кода написаног у различитим програмским језицима у машински код који одговара различитим циљним архитектурама, као и извршавање генерисаног машинског кода употребом заједничких компоненти система, независно од конкретног изворног језика или циљне архитектуре. Компајлер *Graal* [100] је јединствени компајлер у оквиру система *GraalVM*. Ово је оптимизациони компајлер који укључује бројне оптимизационе пролазе у циљу генерисања ефикасног извршног кода.

Између осталог, систем *GraalVM* може бити коришћен као JIT или AOT преводилац за програмски језик Java. У секцији 3.2 биће детаљно објашњен систем *GraalVM Native Image* [101], у оквиру ког је омогућено превођење програма пре његовог извршавања. За све компоненте које се користе током извршавања генерисаног кода, попут компоненте за ослобађање меморије (енг. *garbage collection*), систем *GraalVM Native Image* користи наменску виртуелну машину *SubstrateVM*. *GraalVM* и *SubstrateVM* имплементирани су у програмском језику Java.

Truffle [102] представља оквир за једноставну имплементацију интепретера за језике попут Ruby, R, Python, JavaScript и омогућава високе перформансе извршавања програмских кодова написаних употребом ових језика. Компонента *Truffle* аутоматски омогућава и коришћење компајлера за језике за које постоје интепретери у самом оквиру. Захваљујући интеграцији оквира *Truffle* и компајлера *Graal*, систем *GraalVM* омогућава ефикасно превођење и извршавање генерисаног кода независно од програмског језика улазног кода. Посебно је значајно то што се за постизање високих перформанси приликом извршавања улазног програмског кода не захтева засебан преводилац за тај језик,

односно оптимизовање програмских конструката предвиђеним конкретним језиком, већ се оптимизације на исти начин спроводе у компајлеру. Специјализацијом интерпретера за конкретни програмски језик, као и компајлирањем делова кода, примењује се техника парцијалне евалуације [103].

Кључни кораци за интеграцију нових језика и њихово јединствено превођење и оптимизацију су у наставку. За имплементацију интепретера новог језика користи се структура апстрактног синтаксног стабла (енг. *abstract syntax tree*, скр. AST). Циљ је да интепретер језика у коме је писан улазни програмски код буде једноставан и да исправно описује семантику тог програмског језика. Током интепретирања, на основу информација о извршавању текућег програма, тј. профилима, чвор стабла може бити замењен другим чвором, што представља специјализацију за специфичну операцију улазног програмског језика. Када се извршава компајлирање делова стабла који се односе на често извршаване делове кода, уз примену различитих оптимизација попут инлајновања, ови делови стабла се преводе у оптимизовани машински код.

Генерисани машински код је специјализован на основу типова и вредности који су снимљени током фазе интерпретирања. Операција замене чворова се не преводи у машински код, већ се омогућава операција деоптимизације преведеног кода, која је неопходна уколико се покаже да коришћена специјализација није исправна у неком случају извршавања програмског кода. На основу операције деоптимизације, одбацује се генерисани машински код и извршавање тог дела програма се враћа у AST интепретер. Након спровођења неопходних замена чворова на основу информација које су довеле до деоптимизације, нема препрека да се код поново компајлира, специјализован на основу нових информација. Како појединости улазног програмског језика долазе до изражаја на вишем, апстрактном нивоу, компајлер не мора да познаје његову семантику.

3.2 Превођење у систему *GraalVM Native Image*

GraalVM Native Image преводи улазни програм у извршни фајл, који се назива *native image* (скр. *NI*) и који одговара конкретном оперативном систему и архитектури. Улаз система *GraalVM Native Image* представља скуп класних датотека (енг. *class files*), које садрже Java бајткод [104] програма који се преводи, као и име улазне методе (или улазних метода) програма. Оригинално, улазни програмски код може написан у неком од језика као што су Java, Kotlin или Scala, који се преводе у Java бајткод.

Потребно је разликовати процес генерисања извршног фајла NI (енг. *image build*), односно време генерисања фајла NI (енг. *image build time*) и извршавање фајла NI (енг. *image run*), односно време извршавања фајла NI (енг. *image running time*). Поступак генерисања извршног фајла ослања се на претпоставку о затвореном свету (енг. *closed-world assumption*), што значи да целокупан код који је потребан за превођење и извршавање улазног програмског кода мора бити доступан током генерисања извршног фајла NI. Код свих потребних библиотека, JDK и компоненте виртуелне машине процесирају се на исти начин као и улазни програмски код.

Процес формирања извршног фајла NI се у оквиру система *GraalVM Native Image* састоји од неколико главних корака (приказаних на слици 3.1):

- *Points-to* анализа, која укључује иницијализацију класа (енг. *class initiaization*) и снимање садржаја хипа (енг. *heap snapshotting*)
- AOT превођење
- Попуњавање хипа извршног кода (енг. *image-heap writing*)

Points-to анализа одређује скуп класа, метода и поља који могу бити коришћени (енг. *reachable*) у току извршавања фајла NI [105]. Анализа почиње од улазних тачака програма, нпр. main методе апликације и итеративно обрађује све дохватљиве методе све до постизања стабилног стања, тј. стања у коме нема промена. Анализа се ослања на фазе предњег дела компајлера, када се од улазног бајткода формира међурепрезентација компајлера на вишем нивоу (*GraalIR*), која ће бити детаљније описана у следећој секцији. Графовска репрезентација се трансформише у граф тока типова (енг. *type-flow graph*). Структура графа тока типова је следећа: чворови представљају инструкције које оперишу над типовима објеката, и садрже информације о свим типовима који могу одговарати том чвору, а гране повезују одговарајуће чворове у релацији дефиниција - коришћење. Један граф одговара целом улазном програму и укључује информације о релацијама између потпрограма.

Једна од главних предности система *GraalVM Native Image* је брже иницијално извршавање програма (енг. *fast start-up time*). Између осталог, ово се постиже извршавањем кода за иницијализацију током генерисања извршног фајла, уместо током његовог извршавања. Само неколико корака иницијализације морају бити спроведени у фази извршавања NI фајла, на самом почетку, пре извршавања улазне тачке апликације, попут мапирања меморије хипа, док сви остали кораци могу бити спроведени у оба тренутка. Постоји подршка за једноставно дефинисање у ком тренутку ће бити извршен специфичан код за иницијализацију и ово може бити остављено програмеру да одлучи.

На основу иницијализационог кода, алоцирају се одговарајући Java објекти, који, потом морају бити доступни током извршавања програма. Како би се ово постигло, Points-to анализа их означава као дохватљиве. У фази снимања садржаја хипа формира се граф алоцираних и дохватљивих објеката и затим се смешта у део хипа који ће бити доступан приликом покретања извршног фајла (енг. *image heap*).

Након завршетка фазе снимања садржаја хипа, поново се покреће анализа како би се пропагирале нове информације у оквиру графа тока типова. Након постизања стабилног стања, поново се покреће иницијализациони код, а такође и снимање садржаја хипа. Итеративно се ове три фазе извршавају све до постизања глобалног стабилног стања, након чега није могуће уочити никакву промену додатним понављањем итерација. Крајњи скуп дохватљивих класа, метода и поља образује универзум, који се у датом систему назива *Analysis Universe* и у наставку ће бити реферисан као *АУниверзум*.

Слика 3.1: Главни кораци превођења

Компилациони корак затим, почевши од улазних метода програма, транзитивно компајлира методе из АУниверзума једну по једну, формирајући на основу њих компилационе јединице које ће бити укључене у нови универзум, овде назван *Hosted Universe*, односно *ХУниверзум*. Дакле, у извршни код се преводе само оне методе које је анализа прогласила за дохватљиве. За саму компилацију, користи се *Graal*, оптимизациони компајлер из система *GraalVM*. Детаљнији опис алгоритма за компилацију биће приказан у секцији 6.

На крају, креира се извршни фајл који садржи компајлирани код и почетно стање хипа (које се састоји од објеката који су транзитивно дохватљиви из статичких поља иницијализованих класа). Сви објекти садржани у хипу извршног фајла, биће уписани у његову *data* секцију, док се генерисани извршни код уписује у *text* секцију фајла NI.

3.3 Међурепрезентација компајлера

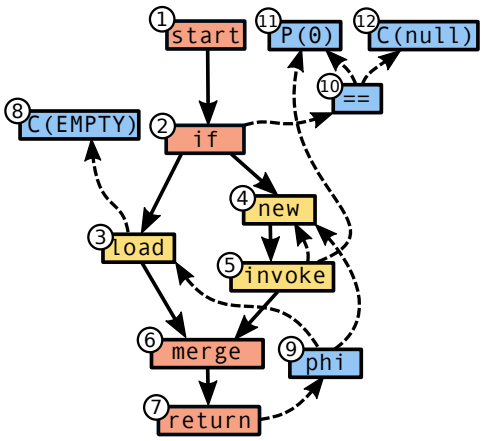
У екосистему *GraalVM*, за обе парадигме превођења AOT и JIT – окружење *Native Image* и виртуелна машина *HotSpot* користе оптимизациони компајлер *Graal*. Након учитавања бајткода програма, његовог парсирања и формирања одговарајуће међурепрезентације кода (енг. *intermediate representation*, скр. IR) [106], компајлер *Graal* примењује бројне оптимизационе пролазе над датом међурепрезентацијом. На неки начин *Graal* се може сматрати компајлером који оптимизације ради унутар процедура (енг. *intraprocedural optimizing compiler*). Свака метода програма се парсира и формира се засебна јединица компилације, а све оптимизације које се спроводе у процесу компилације примењују се над сваком компилационом јединицом посебно. Као што је већ објашњено у секцији 2.2, појам компилационе јединице није еквивалентан појму методе. Осим методе која је корен компилационе јединице, она у највећем броју случајева обухвата више метода (након примене оптимизације инлајновања).

Међурепрезентација у компајлеру *Graal* има структуру усмереног графа. Овај граф истовремено садржи зависности тока контроле (енг. *control flow*) и тока података (енг. *data flow*) између појединачних корака извршавања у одговарајућем делу програма [106], слично репрезентацији *sea-of-nodes* [107]. Сваки корак извршавања представљен је засебним чвором у графу. Чворови се међусобно разликују по комплексности у односу на операцију коју репрезентују, а која може бити операција на високом нивоу апстракције попут копирања низа или операција са стринговима, или може бити на веома ниском нивоу попут учитавања из меморије. Ово пре свега зависи од фазе у којој се граф налази током превођења (енг. *compilation pipeline*). Чворови тока контроле су фиксирани у графу, што значи да се типично не померају у графу у односу на друге чворове. Чворови тока података могу да „плутају” (енг. *floating nodes*), а њихова позиција у графу је ограничена искључиво зависностима међу подацима (енг. *dataflow dependencies*). Како би енкодовали вредности које се могу разликовати у зависности од тока контроле, чворови тока података се ослањају на форму јединствене статичке доделе вредности (енг. *single static assignment*, скр. *SSA*) [108].

На слици 3.2 приказан је поједностављен пример репрезентације графа која одговара методи JDK 8 `Optional.ofNullable` [109]. Метода је дефинисана на следећи начин:

```
<T> Optional<T> ofNullable(T x) {
  return x == null ? EMPTY : new Optional<>(x);
}
```

Ова метода као повратну вредност има објекат уникат (енг. *singleton*) `EMPTY` или обмотава неку специфичну вредност параметра `x`. Гране у графу су представљене пуним линијама када се односе на зависности тока контроле, док се испрекидане линије односе на зависности тока података. Улазна тачка методе којој одговара граф са слике обележена је чвором `start`. Гранање је означено `if` чвором, а постоји грана која се односи на ток података између чвора који испитује услов једнакости и датог чвора гранања. Ова грана значи да исход гранања зависи од резултата израза `P(0) == C(null)`, где је `P(0)` нулти параметар методе (параметар `x`), а `C(null)` константа вредност за `null`. У левој грани налази се чвор за учитавање из статичког поља, а десна грана садржи алокацију `Optional` омотача преко чвора `new` и позив конструктора класе `Optional`. Овај позив узима алоцирани објекат и параметар методе као аргументе, што се у графу види на основу испрекиданих линија од чвора позива (`invoke`) до одговарајућа два чвора. Чвор `merge` односи се на спајање грана тока контроле, а чвор `phi` имплементира логику која разрешава вредност резултата, који зависи од тога контроле. У овом случају то је повратна вредност функције, која зависи од тога која грана тока контроле ће бити извршена. На крају методе, резултат се враћа преко чвора `return`.

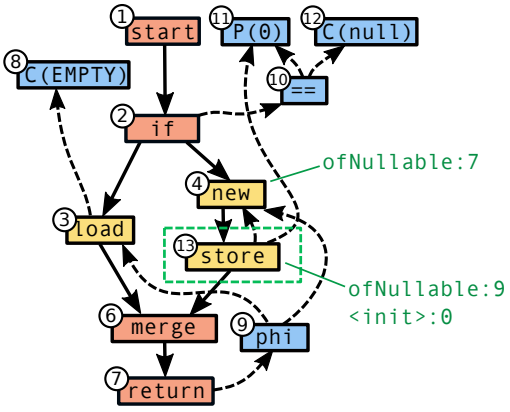


Слика 3.2: Пример међурепрезентације компајлера *Graal*

3.4 Оптимизација инлајновања и позиционирање чворова у коду

Компајлер *Graal* трансформише графовску међупрезентацију кроз низ фаза, од којих је једна фаза инлајновања [5]. У примеру графа који се добија парсирањем методе `ofNullable` чвор `invoke` представља позив конструктора који има улогу да додели вредност параметра конструктора једном пољу објекта класе `Optional`. Код веома једноставних метода, попут ове, већина хеуристика инлајнера ће заменити позив конструктора директним уграђивањем његовог тела [86]. Резултујући граф приказан је на слици 3.3.

Инлајновањем позива конструктора чвор `invoke` замењен је телом тог конструктора, што у овом конкретном случају представља један `store` чвор за смештање прослеђене вредности. На основу овог примера биће објашњен концепт одређивања позиције чвора у изворном коду (енг. *node souce position*, скр. *NSP*) у оквиру компајлера *Graal*. Прецизније, мисли се на позицију чвора у бајткоду јер се више инструкција из бајткода може налазити на истој линији улазног програма. У секцији 2.3 дефинисан је појам индекса у бајткоду (*BCI*) неког Java програма као померај инструкције у односу на почетак методе у којој се налази, изражено у бајтовима. Сваки чвор у *IR* графу неке методе повезан је са инструкцијом у бајткоду на основу које је настао и самим тим са позицијом те инструкције. Дакле, сваки чвор је повезан са паром (метода:BCI) који идентификује локацију са које потиче чвор. Након инлајновања неке методе у компилациону јединицу којој припада њена метода позивалац, информација о позицијама чворова графа инлајноване методе мора бити модификована тако да укључује информацију о конкретном инлајновању. Другим речима, чворови инстанце графа инлајноване методе не могу имати исту позиције као чворови графа те исте методе која није инлајнована (него се компајлира), или која је инлајнована на некој другој локацији у коду. Уместо чувања једног пара који идентификује локацију, компајлер *Graal* додељује листу парова (метода:BCI), тако да сваком пару те листе одговара позиција у оквиру једне методе. Ова листа представља финални *NSP*.



Слика 3.3: Пример графа са инлајновањем

У претходном примеру, на слици 3.3, позиција чвора `new` у бајткоду (NSP) је `ofNullable:7` зато што овај чвор изворно потиче из корена компилационе јединице – методе `ofNullable` и није инлајнован. Позиција чвора `store` је `ofNullable:9,<init>:0` јер је овај чвор део графа конструктора, тј. методе `<init>`, која је инлајнована у методу `ofNullable` на индексу 9. Како је инструкција `store` прва у конструктору, налази се на индексу 0. Контекст позивања (енг. *calling context*) представља позицију инлајновања позива (*NSP*) `invoke`. За чвор `store`, одговарајући контекст је `ofNullable:9`.

3.5 Прикупљање и употреба профила

Прикупљање профила се у окружењу *GraalVM Native Image* традиционално обезбеђује применом технике инструментације формирањем и извршавањем наменског из-

вршног фајла. Овај фајл садржи додатни инструментациони код намењен бројању догађаја који се прате. Након извршавања генерисаног кода преведеног програма, профили се уписују у датотеку по специфичном формату. Датотеке са профилима је могуће учитати током формирања оптимизационог извршног фајла и потом користити профиле ради побољшања процеса превођења. Процес прикупљања профила и њихове примене у окружењу *GraalVM Native Image* описан је у секцији 2.3.3. У наставку биће описани детаљи ових операција, који су блиско повезани са имплементацијом окружења.

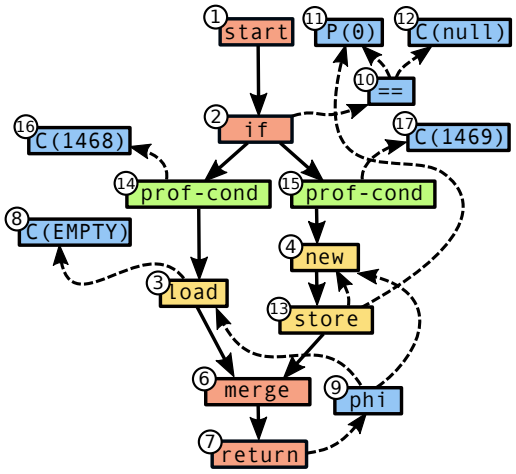
3.5.1 Поступак прикупљања профила

Инструментација је у преводиоцу *Native Image* реализована на нивоу међурепрезентације компајлера *Graal*. Граф сваке компилационе јединице модификује се уграђивањем чворова који служе као бројачи претходно одабраних догађаја за профилисање. Сваки чвор уграђен у графовску међурепрезентацију преводи се у секвенцу инструкција које ажурирају вредност бројача за догађај, који је повезан са локацијом у коду преко позиције чвора бројача (*NSP*). Бројачи су смештени у глобални низ, а индексирају се употребом јединственог идентификатора који је изведен из позиције самог чвора (*NSP*). Како *NSP* обухвата контекст у оквиру компилационе јединице, бројачи повезани са сваким профилисаним догађајем су контекстно осетљиви. Компилациона јединица у највећем броју случајева не садржи улазну методу програма, па је контекстна осетљивост парцијална.

Бројачи имплементирани у постојећој инфраструктури нису атомични. Оваква имплементација усвојена је из следећих разлога. Екстензивни експерименти показали су да су непрецизности уведене применом неатомичних бројача мале, док је успорење извршавања инструментованог програма значајно у случају увођења синхронизације. Осим тога, оптимизационе одлуке донете на основу вредности бројача из профила су у највећој мери хеуристичка решења и као таква су апроксимативна. Непрецизност бројача се овде огледа у могућим мањим варијацијама вредности самог бројача, али значајно је да се не може десити да се изгуби информација о неком догађају у потпуности. Другим речима, било који контекст који се појавио на стеку позива у тренутку прикупљања профила налазиће се сигурно у резултујућој датотеци са профилима, а вредност његовог бројача ће бити већа од нуле. Садржај контекста мора бити исправан, тј. сигурно је да ће контекст позива садржати све локације са стека позива. На крају, циљ овог истраживања је поређење предложене модификоване технике превођења у односу на постојећу примену профила у овом домену унутар окружења *Native Image*, тако да је постојећи механизам профилисања задржан, а модификација инфраструктуре бројача превазилази опсег овог истраживања.

Слика 3.4 приказује граф *IR* који одговара методи из претходног примера, који је допуњен са два чвора намењена инструментацији грана `if` контролне структуре. Ова два `prof-cond` чвора односе се на инкрементирање бројача 1468 и 1469. У каснијим фазама компилације, инструментациони чворови биће замењени са инструкцијама читања вредности бројача из меморије, инкрементирања ове вредности и уписивања новог садржаја на исту локацију у меморији. Инструментациони чворови који који прате извршавања метода или грана контролних структура само увећавају вредност бројача,

док су чворови намењени инструментацији виртуелних позива одговорни и за бележење конкретних типова који одговарају виртуелном позиву неке методе. Они се преводе у код који мапира конкретан тип на одговарајући бројач. По завршетку извршавања инструментационог кода, сви профили су обједињени и уписани у датотеку на диск, која касније може бити коришћена у оптимизационом извршном фајлу.



Слика 3.4: Пример графа са инструментационим чворовима

Формат профила одговара улазу алгоритма дефинисаном у секцији 6.2. Бројач сваког улаза профила повезан је са контекстом директно изведеним из позиције инструментационог чвора (*NSP*). Прва локација у листи локација контекста одговара кореној методи компилационе јединице, док се последња локација односи на пар вредности (метода:BCI) који одговара инструкцији на основу које је инструментациони чвор конструисан.

Један од главних разлога због којих је одлучено да буду коришћени делимично контекстно осетљиви профили (уместо потпуно контекстно осетљивих профила) је ефикасно прикупљање профила као и мало меморијско заузеће бројача. Ефикасност се огледа у мапирању контекста и њима припадајућих бројача на основу позиције чвора у графу компилационе јединице. Као што је већ описано, инструментација омогућава бележење тачног броја извршавања догађаја који се прати у датом контексту. Мана овог приступа лежи у томе што парцијални профили могу бити „загађени" у оквиру компилационих јединица које могу имати велики број позивалаца, а чије понашање се разликује на основу локација позива. Објашњење појма „загађених" профила уз илустративне примере дато је у секцији 2.5. Ово може бити узрок погрешних одлука оптимизације инлајновања. Дужина парцијалних контекста може бити увећана повећањем количине инлајновања у инструментационом коду, али само до извесне границе, што ће бити показано у секцији 8.6.

3.5.2 Поступак употребе профила

Подршка за оптимизације на основу профила је претходно имплементирана у оквиру преводиоца *Native Image* на следећи начин. Чворови у репрезентацији *Graal IR* су допуњени информацијама о извршавању програма који се преводи, попут вероватноће да

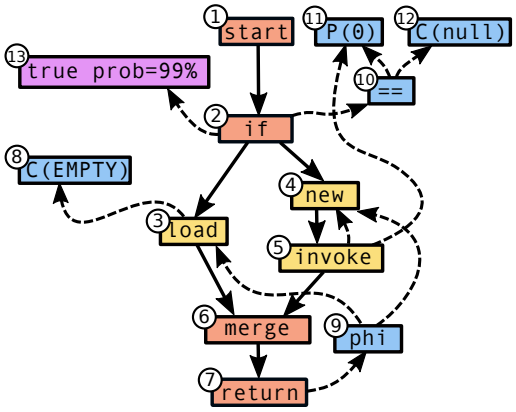
конкретна грана `if` контролне структуре буде извршена или вероватноће да конкретан тип пријемника одговара неком виртуелном позиву методе. Ове информације се израчунавају директно на основу профила када су они доступни и потом се користе како би се усмеравале постојеће оптимизације у компајлеру *Graal*. На пример, вероватноће извршавања грана користе се да се израчунају учестаности базичних блокова, на основу којих се спроводи оптимизација дупликације путања [72], као и бројне трансформације петљи. Вероватноће конкретних типова објеката над којима се позива виртуелна метода користе се за одређивање конкретне имплементације тог позива на некој локацији како би се спровела девиртуелизација [4, 5], тј. замена виртуелног позива директним, чиме се значано смањују трошкови позива, а потенцијално и омогућава инлајновање директног позива. У наставку ће бити објашњени кораци примене парцијалних профила у међурепрезантицији у оквиру постојеће инфраструктуре.

Током формирања оптимизованог извршног фајла, профили се први пут употребљавају у фази парсирања бајткода (енг. *bytecode-parsing phase*), која конструише графове међурепрезентације за све методе програма. Током парсирања, профили се користе као контекстно неосетљиви, другим речима акумулирају се бројачи неког догађаја из свих контекста инлајновања којима припада тај догађај. У тренутку парсирања профиле није могуће користити као контекстно осетљиве зато што до тог момента нису започеле фазе компилације, између осталих инлајновање које уводи појам контекста. Други пут ће профили бити искоришћени управо током оптимизације инлајновања, али овај пут као контекстно осетљиви.

Два главна корака фазе оптимизације инлајновања у екосистему *GraalVM* су експандовање (ширење) и инлајновање. *Native Image* користи профиле као контекстно осетљиве током корака експанзије инлајнера на следећи начин. У графу методе која се компајлира и представља корен компилационе јединице, вероватноће гранања свих контролних структура (`if` и `switch` чворова) као и вероватноће конкретних типова објекта који одговарају виртуелним позивима (`invoke` чворови) израчунавају се на основу искључиво оних профила који одговарају контекстима чворова који се разматрају у стаблу инлајновања (енг. *inlining tree*), као што ће бити објашњено у наставку.

Пример употребе профила. Размотримо слику 3.5, која илуструје ситуацију у којој се профили добијени извршавањем инструментованог кода примењују у компилационој јединици `ofNullable` у оптимизационом извршном коду. Љубичасти чвор са идентификатором 13 означава вероватноћу извршавања `true` гране `if` чвора са идентификатором 2. Као што је речено, ова вероватноћа може бити коришћена у оквиру бројних оптимизација. На пример, оптимизација инлајновања може да закључи како је изузетно мало вероватно да ће бити позван конструктор у оквиру гране `false`, тако да инлајновање овог позива неће значајно допринети убрзању програмског кода. Са друге стране, касније оптимизације које померају код могу да донесу одлуку о измештању чвора `load` са ознаком 3 на позицију изнад гранања (чвор 2), обзиром да спекулативно учитавање (енг. *speculative prefetching*) може резултовати побољшањем перформанси [110, 6].

Пример употребе контекстно осетљивих профила. Разматра се следећи програмски код, који као повратну вредност има или празан објекат JDK `Optional` [109]

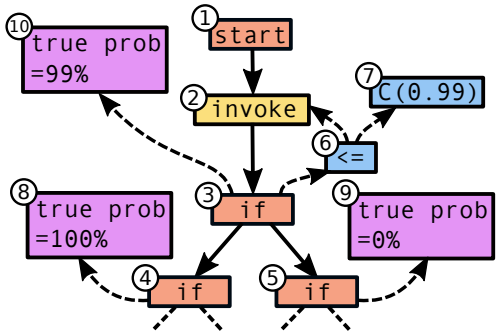


Слика 3.5: Пример графа са означеним вероватноћама

или објекат `Optional Double` вредности између 0.99 и 1.0:

```
double x = Math.random();
if (x <= 0.99) return ofNullable(null) else return ofNullable(x);
```

У датом примеру налазе се две локације позива методе `ofNullable` – на првој се метода увек позива са `null` вредношћу параметра, док се на другој локацији, позив увек обавља пролеђивањем не-`null` параметра. На основу програма је јасно да је очекивана вероватноћа извршавања `true` гране `if` чвора 3 са слике 3.6, који одговара испитивању услова `x <= 0.99` 99%, захваљујући униформном генератору који дохвата случајне вредности у опсегу $[0, 1]$. Након инлајновања методе `ofNullable` на обе локације позива, графовска репрезентација програма садржи по један `if` чвор за сваку копију методе `ofNullable`, који одговарају чворовима 4 и 5. Вероватноћа извршавања `true` грана ових кондиционала се мења захваљујући уведеном контексту – за чвор 4 она ће износити 100%, док ће за чвор 5 износити 0%. Последично, могуће је елиминисати непотребно испитивање услова унутар инлајнованих метода. Контекстно-неосетљива вероватноћа оригиналног `if` чвора која припада методи `ofNullable` и даље остаје 99% како важи да је $99\% \cdot 100\% + 1\% \cdot 0\% = 99\%$.



Слика 3.6: Пример графа са означеним контекстно осетљивим вероватноћама

Упит и агрегација профила. Приликом превођења оптимизованог кода потребно је мапирати позиције у коду и број извршавања делова кода на тој локацији. Значај овога је показан кроз претходни пример. Када контексти позива у оквиру сваке

компилационе јединице извршног фајла намењеног инструментацији одговарају тачно контекстима позива оптимизованог фајла, ово мапирање је тривијално. Ипак, контексти су, у општем случају различити између ове две извршне датотеке, имајући у виду да се на основу профилиних информација доносе другачије оптимизационе одлуке, нпр. током оптимизације инлајновања.

У датом примеру, ради илустрације, биће узето да током формирања оптимизационог извршног фајла, може бити донета одлука да се не инлајнује позив методе `ofNullable` у `else` грани, у ком случају вероватноћа одговарајућег `if` чвора који припада `ofNullable` јединици компилације оптимизационог извршног фајла мора бити апроксимирана на основу два дужа појединачна профила, која су настала извршавањем инструментационог извршног фајла.

У окружењу Native Image, приликом упита фреквенције извршавања кода на основу краћих контекста, тј. контекста који садрже мање локација $\ell_1, \ldots, \ell_n$, правило је да се сумирају све фреквенције асоциране са свим контекстима исте или веће дужине, којима одговара позиција у коду $\ell_1, \ldots, \ell_n, \ldots, \ell_m$ (m може бити једнако n када постоји потпуно поклапање контекста). У конкретном случају чвору `if` у оквиру компилације `ofNullable` би одговарала вероватноћа 99%, као што је претходно наглашено. Вероватноћа ће у датом случају бити израчуната на основу профила исте или веће контекстне осетљивости, који одговарају овој локацији, а добијени су извршавањем инструментованог програма.

Са друге стране, током формирања инструментационог фајла, могуће је не инлајновати `ofNullable` ни на једној од две локације, па ће постојати само краћи контексти позива који ће одговарати јединици компилације `ofNullable`. Тада вероватноће извршавања `true` грана чворова 4 и 5 морају бити апроксимиране на основу ових контекстно неосетљивих профила `if` чвора у оквиру `ofNullable`, чија вероватноћа је 99%.

У окружењу Native Image, када је скуп профила за локацију $\ell_1, \ldots, \ell_n$ исте или веће контекстне осетљивости празан, правило је да се метрике профила, одговарајућих краћих контекста сабирају. У конкретном случају, чворовима 4 и 5, који се у оптимизационом извршном коду инлајнују у методу `ofNullable`, биће додељена апроксимирана вероватноћа 99%.

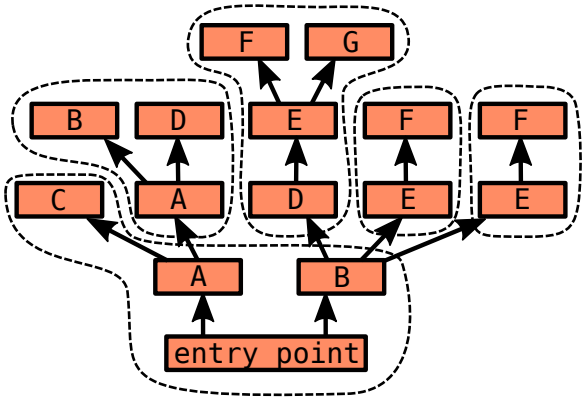
4 Поставка проблема

Након прецизних објашњења релевантних термина приказаних у претходним секцијама, у овом поглављу биће концизно описан проблем који се решава у оквиру истраживања описаног у овој тези. Ово подразумева опис улаза алгоритма за превођење, као и претпоставке и ограничења која важе у конкретном алгоритму. На крају поглавља биће дат и нацрт решења проблема, који ће омогућити једноставније разумевање детаља алгоритма предложеног у овој тези, а који ће бити разрађени у поглављима која следе.

4.1 Опис проблема

Ово истраживање се бави проблемом коришћења прецизних, али делимично контекстно осетљивих профила како би се унапредили распоред превођења и одлуке које се доносе у оптимизацији инлајновања тако да се убрза извршавање преведених програма. Уз то, увећање генерисаног извршног кода преведеног програма не сме бити значајно. Улаз овог проблема је скуп профила, који се састоје од парцијалних контекста и прецизних бројева извршавања везаних за тај контекст. Профили садрже барем метрике које се односе на број позива метода, извршавање контролних структура у програму и виртуелне позиве. Појединачни профил садржи или конкретан број извршавања (нпр. грана контролне структуре) или мапирање типова на број појављивања тих типова за виртуелни позив на некој локацији у коду. Како су сви профили парцијални, подразумева се да њихови контексти не садрже све локације са програмског стека почев од улазне методе програма (нпр. main метода) па до локације догађаја за који је прикупљен профил. Уместо тога, контекст садржи искључиво суфикс садржаја програмског стека (обавезно се завршава локацијом која се прати). Прецизност профила се огледа у постојању профила за сваку локацију у програму која је од интереса и извршена је током покретања извршног кода наменски преведеног за профилисање. Повезане метрике су, такође, прецизне захваљујући техници прикупљања и не представљају апроксимацију вредности.

Значајна претпоставка која се односи на улаз алгоритма је да скуп профила формира партицију на нивоу компилационе јединице (енг. *compilation-unit-wise partition*). Ово ће бити објашњено на основу слике 4.1 на којој је приказан пример активационог стабла. Као што је формално дефинисано у секцији 2.1, активационо стабло садржи све стекове позива који се догађају током извршавања програма. Сваки чвор у таквом стаблу представља једну методу – на пример улазна метода програма `entry point` је корен



Слика 4.1: Пример активационог стабла са означеним компилационим јединицама

стабла, а позване методе `A` и `B` из улазне методе представљене су чворовима потомцима корена овог активационог стабла.

Компилациона једница се састоји од методе у корену и свих повезаних метода које су инлајноване у корену методу. На слици, компилационе јединице су међусобно одвојене испрекиданим затвореним линијама. Скуп профила формира партицију на нивоу компилационе јединице уколико важе следеће тврдње:

- свака метода у преведеном програму за коју су прикупљени профили представља корен највише једне компилационе јединице,
- сваки профил садржи контекст који одговара некој компилационој јединици (контекст почиње методом која је корен те компилационе јединице, и садржи само оне методе које су укључене у ту компилациону јединицу).

У датом примеру, постоје четири компилационе јединице: једна са кореном методом `entry point`, једна са кореном методом `A`, трећа са кореном методом `D` и четврта са кореном методом `E`. Компилациона јединица са кореном методом `E` позвана је са две локације у коду, док су преостале компилационе јединица позване само једном. Контексти `entry point`→`A`→`C` и `A`→`D` су исправни контексти на нивоу компилације, зато што сваки од њих почиње методом која је корен компилационе јединице и све локације контекста припадају истој компилационој јединици. Контекст `E`→`G` није валидан на нивоу компилационе јединице зато што не почиње методом која је корен јединице. Контекст `entry point`→`B`→`D` такође није валидан, зато што није садржан у оквиру једне компилационе јединице. Претпоставка је да улазни профили садрже искључиво валидне контексте на нивоу једне компилационе јединице.

Образложење. Ограничења наметнута улазним профилима могу се најпре чинити вештачким, али су заправо природна последица примене поступка инструментације у оквиру компилационе јединице. Свакој позицији у оквиру компилационе јединице (укључујући и инлајноване методе) додељен је скуп јединствених бројача. Самим тим сваком бројачу одговара неки контекст који се састоји од локација у оквиру те компилационе јединице. Преводилац *GraalVM Native Image*, у оквиру кога је имплементиран предложени алгоритам, прикупља прецизне парцијално контекстно осетљиве профиле који формирају партицију на нивоу компилационе јединице. У конкретном преводиоцу

ово је реализовано превођењем програма у наменски извршни код за инструментацију кода, чијим се извршавањем прикупљају профили за тај програм. Оптимизовани извршни код се потом генерише превођењем истог програма употребом ових профила. Овај приступ у два корака је уобичајен; на сличан начин се инструментациони извршни код генерише и за GCC [23], LLVM [111] и Scala Native [26] преводиоце.

Превођење програма AOT компајлером подразумева компилацију свих метода програма пре његовог извршавања, уз евентуални изузетак оних метода које су на свим локацијама позива инлајноване и никада се експлицитно не позивају. Већина метода програма садрже једну или више локација позива неке циљне методе. Током појединачног превођења неке методе, прецизније, током спровођења оптимизације инлајновања, за сваки позив унутар тела те методе доноси се одлука да ли треба применити оптимизацију или не. Уколико се реализује инлајновање неке позване методе, сам позив се мења телом позиване методе и, транзитивно, методе које се позивају из инлајноване методе даље улазе у разматрање за инлајновање. Све преостале методе које се позивају директно или транзитивно, али нису инлајноване у процесу превођења те јединице компилације, распоређују се касније за превођење. У оваквој конфигурацији, методе се преводе само једном, независно од контекста њиховог позивања.

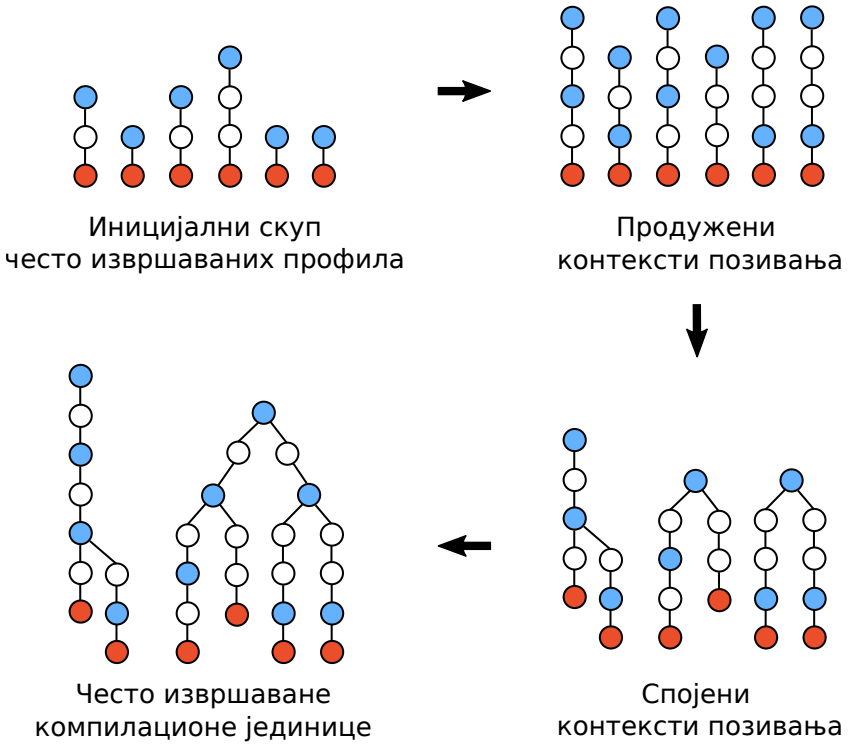
Обзиром да нека метода може бити позвана са више различитих локација у коду, како би се избегло вишеструко превођење исте методе, већина AOT компајлера превођење спроводи у дефинисаном поретку. Како величина и садржај компилационих јединица нису унапред одређени пре саме компилације, поредак превођења засебних метода (*compilation schedule*) може бити дефинисан тек у току извођења саме компилације. Уобичајено, превођење почиње од улазних метода програма, тј. главне методе програма као и метода од којих започињу извршавање других нити програма, уколико их има. Након улазних метода, транзитивно се распоређују све оне методе које нису инлајноване у управо преведене јединице компилације, које одговарају улазним методама програма, а позивају се из њих. Процес се извршава на сличан начин, итеративно, док се не преведе целокупан програм. Распоређивање је имплементирано смештањем у ред за чекање, а приликом дохватања из реда и пре започињања превођења појединачних метода, врши се провера да ли су оне претходно преведене како би се избегло вишеструко превођење, чиме је јасно да алгоритам конвергира.

У зависности од контекста позивања неке методе, она може манифестовати другачије понашање и због тога има смисла издвојити методе и контексте позивања значајне за извршавање целокупног програма. Алгоритам *PRINC*, предложен и описан у овој тези, раздваја често извршаване јединице компилације од осталих, и, као такав, модификује процес превођења како би обезбедио да, уколико се на основу позива неке методе из бар једног контекста одреди да се у њој проводи доста времена, се таква метода преведе у складу са стратегијом за компилацију често извршаваних делова кода. Другим речима, уколико за исту методу постоји један или више контекста позива за који та метода значајно доприноси извршавању програма, и постоји један или више контекста позива из којих се иста метода ретко позива, или се у њој не проводи довољно времена, имајући у виду да свака метода може бити преведена само једном, она мора бити преведена употребом већих оптимизационих напора. Дакле, не сме бити препуштено случају да ли

ће у ред за распоређивање јединица за компилацију најпре доћи захтев за компилацију ове методе на основу значајног контекста или неког од преосталих контекста позивања. Када би метода прво била компајлирана на основу неког мање значајног контекста, за њено превођење би био употребљен мањи буџет и не би могла бити касније преведена са већим степеном оптимизација. Због свега наведеног, предложени алгоритам прво преводи све значајне компилационе јединице, а тек потом процесира преостале. Правилним избором метода од којих започиње компилација и увећањем буџета оптимизација, а посебно оптимизације инлајновања у оквиру ових компилација, могуће је обухватити више кључних информација, које ће допинети генерисању ефикаснијег извршног кода за такве компилационе јединице.

4.2 Нацрт предложеног решења

Улазни скуп делимично контекстно осетљивих профила садржи контексте који се чешће појављују на стеку позива и описују фрагменте често извршаваних делова кода, као и контексте који се ређе извршавају. Алгоритам који је тема овог рада има за циљ да пронађе и повеже компилационе јединице које покривају те често извршаване делове кода. Алгоритми за инлајновање типично функционишу у оквиру процеса компилације појединачне процедуре. Самим тим одлучивање о примени оптимизације почиње унутар процедуре која се преводи, а затим оптимизација повећава компилациону јединицу од корена наниже, ка позваним метода (енг. *top-down manner*) и то ширењем стабла позива док алгоритам не одлучи да заустави „истраживање" у стаблу, тиме формирајући скуп листова стабла позива. Супротно томе, предложени алгоритам креће се „од листова ка



Слика 4.2: Илустрација суштине предлога решења

корену" (енг. *bottom-up manner*).

Неколико главних фаза алгоритма приказано је на слици 4.2. У првој фази, идентификују се парцијални профили који садрже најчешће извршене контексте, који представљају делове кода у којима се проводи највише времена. На основу ових профила, алгоритам формира шуму стабала, у овом раду названих стазама трагова, која моделују фрагменте често извршаваних делова кода, а након тога итеративно продужава стабла употребом највероватнијих позивалаца и спаја она стабла која се односе на исту јединицу компилације.

Крајњи скуп стаза трагова обухвата стабла чији корени представљају значајне компилационе јединице и често позиване методе најближе корену активационог стабла, које одговара извршавању неког програма (енг. *hot entry points*). Стабла се не користе као планови за тачно примењивање инлајновања, већ, уместо тога, као смернице које наводе инлајнер да истражује и инлајнује дуж путања на њима, имајући у виду да га она воде до делова кода у којима је проведено највише времена у програму. Ове смернице су интегрисане у постојећу хеуристику инлајнера и користе се заједно са другим логичким целинама оптимизације. Једном када су често извршавани делови кода преведени, остатак програмског кода се компајлира на стандардни начин.

5 Структуре података за представљање често извршаваног кода

У оквиру овог поглавља дефинисане су структуре података које се користе за детекцију и моделовање значајних секција кода. Главна уведена структура података базира се на структури стабла и у овом раду се зове „стаза трагова" (енг. *Breadcrumb Trail*). Стабла се користе у оквиру новог алгоритма како би се парцијални контексти из профила надовезивали и груписали у циљу апроксимације учестаности извршавања делова кода обухваћених стаблом. Састоје од чворова који се овде зову „трагови" (енг. *Breadcrumb*), а односе се на појединачне локације из контекста профила. „Праћењем трагова" током компилације, инлајновање се усмерава ка често позиваним методама, чиме се постиже боља оптимизација делова кода кључних за перформансе програма. За моделовање свих детектованих фрамената често извршаваног кода биће коришћен скуп стабала, тј. „шума стаза трагова" (енг. (*Trail Set*)). У наставку ће бити дефинисане главне операције над сваком уведеном структуром података. Ове операције представљају саставне делове дефиниције алгоритма *PRINC*, који ће бити детаљно описан у поглављу 6.3.

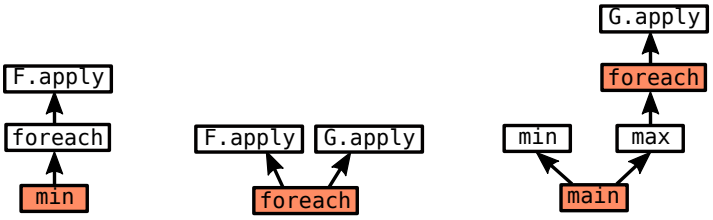
5.1 Стаза трагова

Стаза трагова (скр. *стаза*) која одговара програму P је неки коначан повезан подграф неког стабла позива за програм P. Дефинишемо скуп стаза трагова програма P ($\mathbb{B}(P)$) на следећи начин (у једначини 5.1, $\aleph_0$ означава величину скупа природних бројева $\mathbb{N}$, па се овде користи нотација $|N| < \aleph_0$ како би се изразило да је нешто коначно):

$$\mathbb{B}(P) \equiv \{(N, E) : |N| < \aleph_0 \wedge E \subseteq N \times N \wedge \exists x, x = \inf_{E^*} N \wedge \exists C \in \mathbb{C}(P), (N, E) \subseteq C\} \quad (5.1)$$

На основу једначине 5.1, стаза трагова β у општем случају не представља стабло позива. Њено главно ограничење јесте то што потомци сваког чвора η припадају скупу $callees(sub(\eta))$. Ипак, стаза $\beta = (N, E)$ јесте стабло са дефинисаним кореном $\inf_{E^*} N$. Чвор у стази трагова називамо *траг*.

Слика 5.1 садржи примере стаза трагова. На основу стазе са кореном методом `min` долази се до `F.apply`, док се на основу стазе са кореном методом `foreach` долази до



Слика 5.1: Примери стаза трагова

оба позива `F.apply` и `G.apply`. Обе стазе служе као упутство за проналажење често извршаваних метода када превођење започиње из одређеног контекста.

У оквиру овог рада стазе ће бити коришћене, грубо говорећи, да транзитивно повежу одговарајуће место позива са методама које могу бити позване са те локације, а идентификоване су као методе које се често извршавају, тј. са методама за које је процењено да се значајна количина времена проводи у њима када су позване из датог контекста. Ипак, дефиниција самих структура података ничим не намеће овакву употребу, па самим тим стазе могу бити коришћене и у друге сврхе.

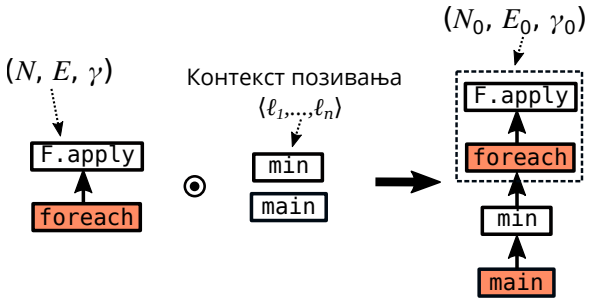
На слици 5.1, која приказује стазе, неки чворови су приказани у наранџастој боји. Корен је увек обојен наранџасто, али и други чворови стабла могу бити истакнути на исти начин. Овако анотирани чворови у наставку називају се тачкама спајања (енг. *graft points*), а стазе који их садрже називају се анотираним стазама трагова (енг. *Annotated Breadcrumb Trails*). Скуп анотираних стаза који одговара програму P ($\mathbb{T}(P)$) дефинисан је на следећи начин:

$$\mathbb{T}(P) \equiv \{(N, E, \gamma) : (N, E) \in \mathbb{B}(P) \wedge \gamma \subseteq N \wedge \inf_{E^*} N \in \gamma\} \tag{5.2}$$

У наставку текста ради поједностављења, када се користи термин *стаза (трагова)* подразумеваће се анотирано стабло.

Операције. Како би алгоритми описани у секцији 6.3 били прецизно дефинисани, потребно је дефинисати операције над анотираним стазама трагова. Претпоставимо да постоји „релативно кратка" стаза, тј. стаза која описује путању до често извршаваног кода из позиваоца који су „веома близу". „Близина" се у овом смислу изражава у броју локација са програмског стека садржаних у контексту. Нека је једна таква стаза $\tau = (N, E, \gamma)$, који почиње кореном $root(\tau) = \inf_{E^*} N$. Од интереса је продужити стазу употребом контекста $\ell_1, \ldots, \ell_n$. Операција $\odot$, која ће у даљем тексту бити названа продужавањем односно експанзијом стаза (енг. *breadcrumb-trail expansion*), формира нову стазу која почиње ланцем чворова-трагова који одговарају локацијама контекста којим се врши експанзија, тј. $\ell_1, \ldots, \ell_n$, и чији чвор који одговара локацији ℓ_n као потомка узима чвор $root(\tau)$. Услов за овакву експанзију је наравно да се са локације ℓ_n позива процедура $sub(root(\tau))$. Пример са слике 5.2 приказује стазу са кореном методом `foreach`, која се експандује у нову стазу са кореном који одговара методи `main`, чиме се омогућава да се често извршавани код детектује са „веће даљине".

У једначини 5.3 постојећи чворови су преименовани тако да укључују префикс $\ell_1, \ldots, \ell_n$, а за сваки позив у контексту $\ell_1, \ldots, \ell_n$, нови скуп чворова и грана је укључен.

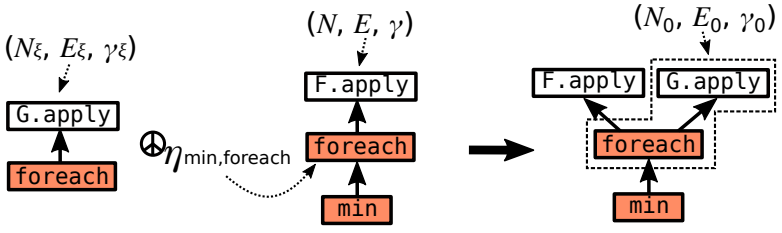


Слика 5.2: Пример операције продужавања стаза трагова

$$(N, E, \gamma) \odot \langle \ell_1, ..., \ell_n \rangle \equiv (\{\eta_{\ell_1}, ..., \eta_{\ell_1,...,\ell_n}\} \cup N_0, \{\eta_{\ell_1} \to \eta_{\ell_1,\ell_2}, ..., \eta_{\ell_1,...,\ell_n} \to \inf_{E_0^*} N_0\} \cup E_0, \eta_{\ell_1} \cup \gamma_0)$$
$$N_0 = \{\eta_{\ell_1,...,\ell_n,s_1,...,s_n} : \eta_{s_1,...,s_n} \in N\} \quad \gamma_0 = \{\eta_{\ell_1,...,\ell_n,s_1,...,s_n} : \eta_{s_1,...,s_n} \in \gamma\}$$
$$E_0 = \{\eta_{\ell_1,...,\ell_n,s_1,...,s_n} \to \eta_{\ell_1,...,\ell_n,s_1,...,s_n,s_{n+1}} : \eta_{s_1,...,s_n} \to \eta_{s_1,...,s_n,s_{n+1}} \in E\} \tag{5.3}$$

Претпоставимо да постоји неколико стаза које садрже трагове који одговарају истој методи s. Када се разматра које позване методе из s су значајне, потребно је испитати све трагове за процедуру s у свим стазама. Ово може бити незгодно имајући у виду да је далеко једноставније преводиоцу да провери један траг када год мора да одлучи шта је „важно" за методу која одговара том чвору. Због тога је потребно дефинисати још једну операцију над стазама трагова.

Операција $\circledast_\eta$ представља операцију спајања стаза трагова (енг. *breadcrumb-trail grafting*). Ова операција качи једну стазу $(N_\xi, E_\xi, \gamma_\xi)$ за чвор η у оквиру друге стазе (N, E, γ). У примеру са слике 5.3 операција спајања додаје локацију позива функције `G.apply` чвору који одговара процедури `foreach` у оквиру стазе са кореном методом `min`. Иако је спајање стабала у овом примеру исправно, није посебно употребљиво на основу примера. У датом програму, метода `G.apply` се никада не позива из контекста који почиње методом `min`.



Слика 5.3: Пример операције спајања стаза трагова

Формално изражено, уколико је дата стаза трагова $\tau = (N, E, \gamma)$, њен траг $\eta_{s_1,...,s_n} \in \gamma$ и стаза трагова $\xi = (N_\xi, E_\xi, \gamma_\xi)$ такви да важи $sub(\eta_{s_1,...,s_n}) = sub(root(\xi))$, односно $s_n = sub(\inf_{E_\xi^*} N_\xi)$, операција спајања стаза $\xi \circledast_{\eta_{s_1,...,s_n}} \tau$ (на основу једначине 5.4) ствара копију стазе τ у којем чвор $\eta_{s_1,...,s_n}$ добија по једно додатно стабло из ξ за сваког потомка чвора $\eta_{s_n,c_2} \in children(root(\xi))$ тако да $\nexists \eta_{s_1,...,s_n,c_2} \in children(\eta_{s_1,...,s_n})$, док су сва подстабла потомака $\eta_c \in children(root(\xi))$ за која $\exists \eta_{s_1,...,s_n,c_2} \in children(\eta_{s_1,...,s_n})$ рекурзивно спојена.

$$(N_\xi, E_\xi, \gamma_\xi) \circledast_{\eta_{s_1,\ldots,s_n}} (N, E, \gamma) \equiv (N \cup N_0, E \cup E_0, \gamma \cup \gamma_0) \quad \textit{где је} \quad \inf_{E^*_\xi} N_\xi = s_n \quad \eta_{s_1,\ldots,s_n} \in \gamma$$
$$N_0 = \{\eta_{s_1,\ldots,s_n,c_2,\ldots,c_m} : \eta_{s_n,c_2,\ldots,c_m} \in N_\xi\} \quad \gamma_0 = \{\eta_{s_1,\ldots,s_n,c_2,\ldots,c_m} : \eta_{s_n,c_2,\ldots,c_m} \in \gamma_\xi\}$$
$$E_0 = \{\eta_{s_1,\ldots,s_n,c_2,\ldots,c_m} \to \eta_{s_1,\ldots,s_n,c_2,\ldots,c_m,c_{m+1}} : \eta_{s_n,c_2,\ldots,c_m} \to \eta_{s_n,c_2,\ldots,c_m,c_{m+1}} \in E_\xi\} \tag{5.4}$$

Ако се усвоји конвенција да важи $(N_\xi, E_\xi, \gamma_\xi) \cup (N, E, \gamma) \equiv (N_\xi \cup N, E_\xi \cup E, \gamma_\xi \cup \gamma)$ и $\inf(N_\xi, E_\xi, \gamma_\xi) \equiv \inf_{E^*_\xi} N_\xi$, тада је могуће прецизније дефинисати операцију спајања стаза трагова на следећи начин:

$$\xi \circledast_{\eta_{s_1,\ldots,s_n}} (N, E, \gamma) \equiv (\xi \odot \langle s_1, \ldots, s_{n-1} \rangle) \cup (N, E, \gamma) \quad \textit{где је} \quad \inf \xi = s_n \quad \eta_{s_1,\ldots,s_n} \in \gamma \tag{5.5}$$

5.2 Шума стаза

Операције. У наставку ће бити дефинисане три операције над шумом стаза трагова, које су неопходне за разумевање дефиниције алгоритма приказаног у секцији 6.3. Ове три операције су спајање шума стаза (енг. *union-grafting*) $\overset{\circledast}{\cup}$, самоспајање стаза (енг. *self-grafting*) $\overset{\circledast}{\nabla}$ и корено самоспајање стаза (енг. *self-root-grafting*) $\overset{\circledast}{\nabla}_{\inf}$. Операција спајања шума стаза формира унију два скупа стаза, операција самоспајања качи једну стазу из скупа на друге стазе у оквиру истог скупа, а операција кореног самоспајања је слична операцији самоспајања, али се качење одвија искључиво у корену стаза.

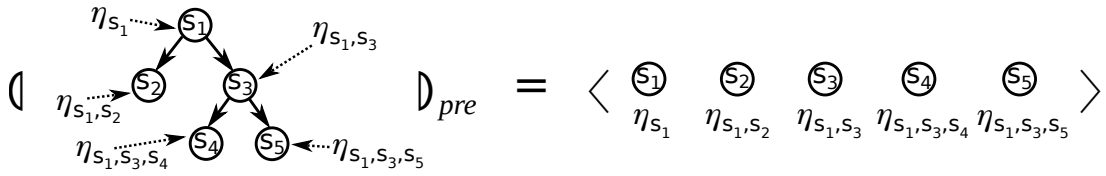
Како би се одредио поредак спровођења операција, биће изабрано специфично уређење чворова унутар стаза, као и уређеност стаза у оквиру шуме. Нека је $(\!|X|\!)_{ord}$ секвенца елемената из скупа X, који су уређени према тоталном поретку *ord*:

$$(\!|X|\!)_{ord} \equiv \langle x_0, \ldots, x_n \rangle : x_i \in X \wedge x_i <_{ord} x_j \Rightarrow i < j \tag{5.6}$$

Следеће, нека је $pre(\tau)$ тотални поредак трагова у стази τ, добијен на основу *preorder* обиласка стазе τ с' лева на десно. Preorder обилазак представља лексикографско уређење стекова позива представљених траговима:

$$\eta_{s_{1,1},\ldots,s_{1,n_1}} <_{pre(\tau)} \eta_{s_{2,1},\ldots,s_{2,n_2}} \equiv s_{1,1} \ldots s_{1,n_1} <_{\text{lex}} s_{2,1} \ldots s_{2,n_2} \tag{5.7}$$

На слици 5.4 приказан је пример одређивања поретка чворова стазе трагова, која садржи пет трагова.

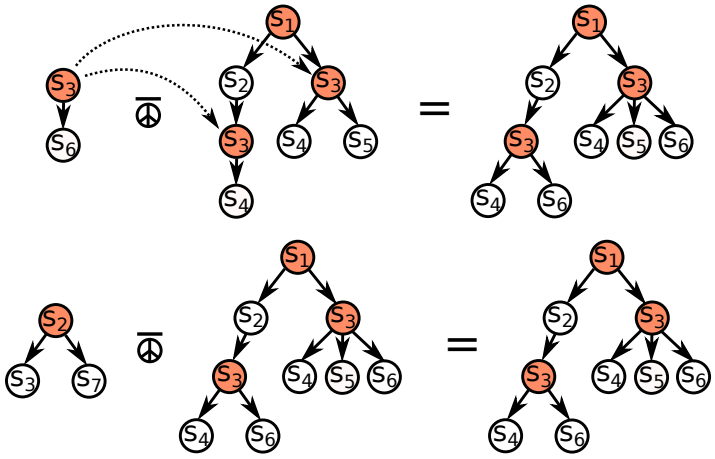


Слика 5.4: Пример одређивања поретка чворова стазе трагова

Операција спајања шума стаза $\overset{\circledast}{\cup}$ се ослања на десно асоцијативну операцију $\overline{\circledast}$ која, за дату стазу који се „качи" $\xi = (N_\xi, E_\xi, \gamma_\xi)$ и циљну стазу $\tau = (N, E, \gamma)$, на којој се

спајање догађа, спроводи спајање стазе ξ на све кандидате тачке спајања у стази τ. Другим речима, операција спајања се спроводи на свим тачкама спајања које се односе на исте потпрограме као и корен стазе ξ. Кандидати-тачке спајања су уређене на основу *preorder* обиласка стазе τ.

Слика 5.5 садржи два примера операције $\overline{\circledast}$. У првом примеру, стаза трагова $s_3 \to s_6$ је припојена циљној стази на обе тачке спајања s_3. У другом примеру, стаза трагова $s_3 \leftarrow s_2 \to s_7$ не може бити припојена циљној стази трагова преко њених тачака спајања из разлога што ниједна тачка спајања циљне стазе не одговара методи s_2.



Слика 5.5: Примери операције спајања стазе ξ на све кандидате-тачке спајања стазе τ

У дефиницији операције $\overline{\circledast}$ користи се помоћна функција g за рекурзивно спајање на свим кандидатима. У једначини 5.8, $pre(N, E)$ је скраћена форма $pre((N, E, \gamma)))$:

$$
\begin{aligned}
&\xi \overline{\circledast}(N, E, \gamma) \equiv g(\xi, (N, E, \gamma), \gamma) \\
&g(\xi, (N, E, \gamma), B) \equiv \begin{cases} \xi \circledast_{\eta_{last}} g(\xi, (N, E, \gamma), B \setminus \eta_{last}) & \text{ако } B \neq \varnothing \wedge s_{m,n_m} = sub(\inf_{E^*_\xi} N_\xi) \\ r(\xi, (N, E, \gamma), B \setminus \eta_{last}) & \text{ако } B \neq \varnothing \wedge s_{m,n_m} \neq sub(\inf_{E^*_\xi} N_\xi) \\ (N, E, \gamma) & \text{у супротном} \end{cases} \\
&\text{где је} \quad (\!|B|\!)_{pre(N,E)} = \langle \eta_{s_{1,1},\ldots,s_{1,n_1}}, \ldots, \eta_{s_{m,1},\ldots,s_{m,n_m}} \rangle \qquad \eta_{last} = \eta_{s_{m,1},\ldots,s_{m,n_m}}
\end{aligned}
\tag{5.8}
$$

На крају, да би се омогућило уређивање скупова стаза, биће дефинисан поредак *lex* над стазама τ_1 и τ_2 као лексикографски поредак над листама формираним на основу *preorder* обиласка чворова стаза τ_1 и τ_2:

$$
(N_1, E_1, \gamma_1) <_{lex} (N_2, E_2, \gamma_2) \equiv (\!|N_1|\!)_{pre(N_1,E_1)} <_{lex} (\!|N_2|\!)_{pre(N_2,E_2)} \tag{5.9}
$$

У једначинама 5.7 и 5.9, дефинисана су два лексикографска поретка за уређивање чворова и стабала, што потом омогућава детерминистичко дефинисање следећих опера-

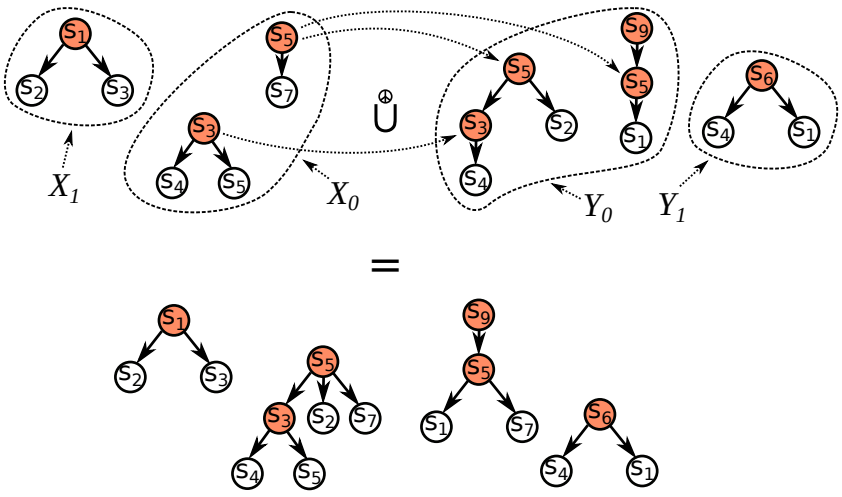
ција. Без губитка општости, и други начини уређивања су могли бити одабрани, али је због једноставности усвојен поредак *lex*.

Операција спајања шума стаза $\overset{\circledast}{\cup}$. Претпоставимо да је циљ спојити шуму стаза X са шумом Y качењем стаза из скупа X на стазе из скупа Y. Скуп стаза X може бити раздвојен у два скупа X_0 и X_1, тако да стазе из скупа X_0 могу бити спојене са стазама из скупа $Y_0 \subseteq Y$, док стазе из скупа X_1 не могу бити спојене са стазама из Y. Операција спајања уније качи сваку стазу $\xi_i \in X_0$ на локације спајања у оквиру стаза $\tau_j \in Y_0$ према лексикографском поретку стаза ξ_i. Резултат спајања се комбинује са остацима $Y_1 = Y \setminus Y_0$ и X_1. У једначини 5.10, $\cup$ је унија два скупа без пресека (енг. *disjoint union*):

$$X \overset{\circledast}{\cup} Y \equiv \begin{cases} X_1 \cup Y_1 \cup \left(\{\xi_1 \overline{\circledast} \ldots \overline{\circledast} \xi_n \overline{\circledast} \tau_1\} \cup \left(X_0 \overset{\circledast}{\cup} (Y_0 \setminus \{\tau_1\})\right)\right) & \text{ако} \quad Y_0 \neq \varnothing \\ X_1 \cup Y_1 & \text{ако} \quad Y_0 = \varnothing \end{cases} \tag{5.10}$$

$$\text{где је} \quad X = X_0 \cup X_1 \quad Y = Y_0 \cup Y_1 \quad (\!|X_0|\!)_{lex} = \langle \xi_1, \ldots, \xi_n \rangle \quad (\!|Y_0|\!)_{lex} = \langle \tau_1, \ldots, \tau_m \rangle$$
$$X_0 = \{\xi \in X : \exists \tau \in Y, \xi \overline{\circledast} \tau \neq \tau\} \quad Y_0 = \{\tau \in Y : \exists \xi \in X, \xi \overline{\circledast} \tau \neq \tau\}$$

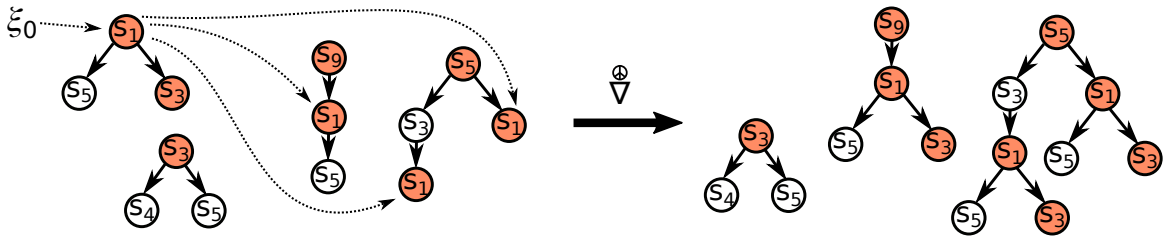
Претходна дефиниција истиче следеће: све стазе $\xi \in X$ које могу бити припојене бар једној стази $\tau \in Y$ биће припојене свим стазама $\tau \in Y$ које задовољавају услов спајања. Све преостале стазе $\xi \in X$ су неизмењене. Резултат операције је унија свих резултујућих стаза (измењених и неизмењених). Пример операције спајања две шуме стаза дат је на слици 5.6.



Слика 5.6: Примери операције спајања шума стаза

Операција самоспајања стаза $\overset{\circledast}{\nabla}$. Операција самоспајања узима један скуп стаза T као улаз и припаја једну од стаза τ_0 из овог скупа другим стазама које припадају скупу T. У једначини 5.11, дефинисано је следеће у контексту операције спајања скупа: ако постоји стаза-кандидат τ_0 у скупу T која може бити припојена другим стазама, тада се примењује операција спајања дате стазе и остатка скупа T. У супротном, резултат операције је оригинални скуп T. Пример извршавања операције приказан је на слици 5.7.

$$\overset{\circledast}{\nabla}\, T \equiv \begin{cases} \{\xi_0\} \overset{\circledast}{\cup} (T \setminus \{\xi_0\}) & \langle \xi_0, \ldots, \xi_n \rangle = (\!|\{\xi \in T : \exists \tau \in T, \xi \neq \tau \wedge \xi \overline{\circledast} \tau \neq \tau\}|\!)_{lex} \\ T & \textit{у супротном} \end{cases} \tag{5.11}$$



Слика 5.7: Пример операције самоспајања стаза

Фиксна вредност fix $\overset{\circledast}{\nabla}$ представља лимит репетитивног извршавања операције самоспајања стаза у оквиру скупа стаза T. У једначини 5.12, нотација $\circ^n$ представља n примена функције:

$$(\mathsf{fix}\ \overset{\circledast}{\nabla})(T) = \lim_{n \to \infty} (\circ^n\ \overset{\circledast}{\nabla})(T) \tag{5.12}$$

Лема 1 *Фиксна вредност* fix $\overset{\circledast}{\nabla}$ *постоји за све скупове стаза трагова* T*. Даље, за све скупове стаза* T*, постоји* $n \in \mathbb{N}$ *такво да је фиксна вредност једнака* n *примена операције самоспајања стаза, тј.,* $(\mathit{fix}\ \overset{\circledast}{\nabla})(T) = (\circ^n\ \overset{\circledast}{\nabla})(T)$*.*

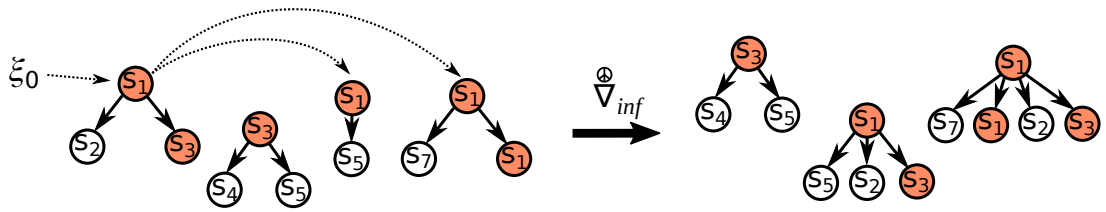
Доказ 1 *Размотрити операцију спајања* $\circledast_{\eta_{s_1,\ldots,s_n}}$*: на основу једначине 5.4. На основу две улазне стазе трагова, применом дате операције добија се једна резултујућа стаза. Даље, на основу једначине 5.8, операција* $\overline{\circledast}$ *такође своди две стазе у једну резултујућу. Дакле, на основу једначине 5.10, резултат операције спајања скупова, која се примењује над два скупа стаза* X *и* Y *је или скуп исте кардиналности у случају да никакво спајање стаза није могло да се реализује, или скуп чија кардиналност износи* $|X_1| + |Y_1| + |Y_0|$*, што је мање од* $|X| + |Y|$*. Самим тим, свака примена операције* $\overset{\circledast}{\cup}$*, односно операције* $\overset{\circledast}{\nabla}$*, производи скуп стаза чија кардиналност је мања или једнака од кардиналности улазних скупова. Испитивањем првог случаја једначине 5.11, јасно је да је резултујући скуп увек мањи захваљујући избору стазе* ξ_0 *тако да је спајање могуће извршити. Тако кардиналност резултујућих скупова стаза секвенце* $(\circ^i\ \overset{\circledast}{\nabla})(T)$ *стрикно монотоно опада. Други случај једначине 5.11 мора бити примењен у неком тренутку (самим тим што величина скупа не може опасти испод* 1*), тако да је могуће пронаћи вредност* n *за које важи* $(\circ^{n+1}\ \overset{\circledast}{\nabla})(T) = (\circ^n\ \overset{\circledast}{\nabla})(T)$*, што имплицира да гранична вредност постоји, и да је фиксна вредност једнака* $(\circ^n\ \overset{\circledast}{\nabla})(T)$*.*

Операција кореног самоспајања $\overset{\circledast}{\nabla}_{\inf}$**.** Лема 1 имплицира да је почевши од било ког скупа стаза T могуће постићи лимит операције у мање од $|T|$ корака. Ипак, постоје патолошки случајеви, код којих величина појединачних стаза у скупу може да нарасте

до $O(2^{|T|})$. За илустрацију ове ситуације биће коришћен скуп стаза који се састоји од $|T|$ међусобно рекурзивних копија Фибоначијеве функције (енг. *Fibonacci function*) `f(n)=f(n-1)+f(n-2)`. Операција самоспајања над овим скупом експоненцијално увећава стазе трагова у сваком кораку. Како би се такав сценарио заобишао, дефинише се операција кореног самоспајања (енг. *self-root-graft operation*) $\overset{\circledast}{\nabla}_{\text{inf}}$, која припаја стазе-кандидате искључиво оним стазама са којима деле корену методу:

$$\overset{\circledast}{\nabla}_{\text{inf}} T \equiv \begin{cases} \{\xi_0 \underset{root(\tau)}{\circledast} \tau : \tau \in R(\xi_0, T)\} \cup (T \setminus R(\xi_0, T)) & \langle \xi_0, \ldots, \xi_n \rangle = (\!|\{\xi \in T : R(\xi, T) \neq \varnothing\}|\!)_{lex} \\ T & \textit{у супротном} \end{cases}$$
$$\textit{где} \quad R((N_\xi, E_\xi, \gamma_\xi), T) = \{(N, E, \gamma) \in T : \tau \neq \xi \wedge \inf_{E^*} N = \inf_{E^*_\xi} N_\xi\} \tag{5.13}$$

Пример извршавања операције кореног самоспајања приказан је на слици 5.8.



Слика 5.8: Пример операције кореног самоспајања стаза

Тачка fix $\overset{\circledast}{\nabla}_{\text{inf}}$ представља лимит репетитивне примене операције кореног самоспајања у оквиру скупа стаза T:

$$(\text{fix}\ \overset{\circledast}{\nabla}_{\text{inf}})(T) = \lim_{n \to \infty} (\circ^n\ \overset{\circledast}{\nabla}_{\text{inf}})(T) \tag{5.14}$$

Лема 2 *Фиксна вредност* $(\text{fix}\ \overset{\circledast}{\nabla}_{\text{inf}})(T)$ *постоји за сваки скуп* T*, и постиже се у коначном броју корака.*

Доказ 2 *На основу Леме 1 закључује се следеће. Како први случај у једначини 5.13 у суштини представља операцију спајања скупа са мањим бројем тачака спајања, лимит се постиже у истом или мањем броју итерација, као у случају операције самоспајања.*

Лема 3 *Свака стаза* (N, E, γ) *у* $(\text{fix}\ \overset{\circledast}{\nabla}_{\text{inf}})(T)$ *има јединствену корену методу, односно* $\inf_{E^*} N$ *је јединствен.*

Доказ 3 *На основу једначине 5.13, фиксна вредност не може бити достигнута све док има стаза* ξ_0 *које су кандидати за спајање. Док постоје две стазе у* $(\circ^i\ \overset{\circledast}{\nabla}_{\text{inf}})(T)$ *које деле корену методу, тада се може применити први случај и постоји кандидат* ξ_0 *који задовољава услов* $R(\xi_0, (\circ^i\ \overset{\circledast}{\nabla}_{\text{inf}})(T)) \neq \varnothing$*, тако да* $(\circ^{i+1}\ \overset{\circledast}{\nabla}_{\text{inf}})(T) \neq (\circ^i\ \overset{\circledast}{\nabla}_{\text{inf}})(T)$*.*

Ради лакшег прегледа најважнијих симбола и операција које се користе за представљање модела у овој секцији у табели 5.1 дати су њихови описи и објашњења.

Табела 5.1: Преглед главних симбола и операција које описују модел

Симбол	**Име**	**Објашњење**
$\mathbb{B}(G, e)$	Скуп стаза трагова (Јед. 5.1)	Скуп коначних стабала $\beta = (N, E)$ таквих да је β подскуп неког стабла позива C, тј. $\beta \subseteq C \in \mathbb{C}(G, e)$.
$\tau(P)$	Анотирана стаза трагова	Стаза трагова β са означеним чворовима (γ), који представљају тачке спајања.
$\mathbb{T}(G, e)$	Скуп анотираних стаза трагова (Јед. 5.2)	Скуп стабала $\tau = (N, E, \gamma)$, где је сваки стаза $\beta = (N, E)$ садржи означене чворове спајања $\gamma \subseteq N$.
$\tau \odot \langle \ell_1, ..., \ell_n \rangle$	Експанзија стазе (Јед. 5.3)	Операција продужава стазу контекстом позивања $\ell_1, ..., \ell_n$, који качи на корен стазе τ.
$\xi \circledast_{\eta_{s_1,...,s_n}} \tau$	Спајање стазе (Јед. 5.4)	Операција која припаја стазу ξ стази τ качењем на чвор $\eta_{s_1,...,s_n}$.
$(\!\|X\|\!)_{ord}$	Уређење скупа X (Јед. 5.6)	Секвенца чланова скупа X уређена је на основу тоталног поретка ord.
$\xi \overline{\circledast} \tau$	Операција спајања на све кандидате (Јед. 5.8)	Операција која спаја стазу трагова ξ на све кандидате тачке спајања стазе трагова τ.
$X \mathbin{\circledcirc\!\!\cup} Y$	Операција спајања шума стаза (Јед. 5.10)	Операција спаја подскуп $X_0 \subseteq X$ стаза из скупа X у скуп Y тамо где је то могуће, а потом враћа унију преосталих стаза које припадају $X \setminus X_0$.
$\overset{\circledast}{\nabla} T$	Операција самоспајања (Јед. 5.11)	Операција бира стазу τ_0 из шуме T, и припаја је другим стазама из исте шуме T на одговарајуће тачке спајања.
$(\mathsf{fix}\ \overset{\circledast}{\nabla})(T)$	Фиксна вредност $\overset{\circledast}{\nabla}$ (Јед. 5.12)	Лимит понављања примене операције самоспајања.
$\overset{\circledast}{\nabla}_{\inf} T$	Операција кореног самоспајања (Јед. 5.13)	Операција бира стазу τ_0 из шуме T, и припаја је другим стазама из исте шуме T уколико обе стазе имају исту корену методу.
$(\mathsf{fix}\ \overset{\circledast}{\nabla}_{\inf})(T)$	Фиксна вредност $\overset{\circledast}{\nabla}_{\inf}$ (Јед. 5.14)	Лимит понављања примене операције кореног самоспајања.

6 Нови алгоритам превођења

Ово поглавље садржи формалну дефиницију алгоритма *PRINC* за унапређење превођења изменом распореда компилације и оптимизације инлајновања на темељу делимично контекстно осетљивих профила [41]. Овде ће најпре бити представљен алгоритам без уласка у детаље, а потом ће он бити декомпонован у целине, чије ће појединости бити објашњене. Формулације приказаних алгоритама су концизне, али се ослањају на апстракције, операције и друге концепте који су у највећем броју случајева уведени у поглављима 2 и 5, док ће у осталим ситуацијама бити прецизно објашњени у овом поглављу. Већина формалних дефиниција илустрована је примерима.

У циљу истицања разлика у односу на претходно решење, у секцији 6.1 ће бити приказан постојећи алгоритам превођења на вишем нивоу апстракције. Секција 6.2 садржи дефиниције улаза и излаза алгоритма. Потом ће бити дат преглед новог алгоритма за превођење и инлајновање у оквиру кога ће, осим саме дефиниције алгоритма, бити приказано извршавање над једноставним улазним програмом. Употребом прецизно дефинисаних операција над уведеним структурама података, које се користе за детекцију често извршаваних делова кода, из поглавља 5, биће објашњена процена учестаности извршавања већих секција кода ради селекције оних делова који су најзначајнији за перформансе целог програма. На крају ће бити приказана и модификација алгоритма за инлајновање, као и преглед главних параметара инлајнера, на које се ослања унапређени алгоритам.

6.1 Постојећи алгоритам превођења

У овој секцији ће бити приказан алгоритам превођења који не укључује измене имплементиране у овом раду. Биће објашњени главни кораци алгоритма без детаља имплементације, а у сврху прављења дистинкције између старог и новог решења. Псеудокод који описује оригинални поступак превођења описан је у наставку у оквиру алгоритма 6.1.

Улаз алгоритма представљају граф позива G, скуп делимично контекстно осетљивих профила П, који се користе током примене оптимизације инлајновања и улазна тачка e програма који се преводи. У општем случају метода које представљају улазне тачке програма може бити више од једне, што не мења логику извршавања алгоритма, па је без губитка општости овде узето да постоји једна почетна метода e. Улаз старог алгоритма за превођење идентичан је улазу новог алгоритма и биће формално описан

Алгоритам 6.1: Постојећи алгоритам за превођење

```
   Input  : call graph G, entry-point e, profile Π
   Output : compilation schedule Σ
1  Σ = ∅; compilationUnits = { e } ;
2  while compilationUnits ≠ ∅ do
3  |  compilationUnits = compilationUnits \ { s_C } : s_C ∈ compilationUnits ;
4  |  C = InlineCold(s_C);
5  |  Σ = Σ ∪ { C } ;
6  |  for s ∈ callees(C) \ (Σ ∪ compilationUnits) do
7  |  |  compilationUnits = compilationUnits ∪ { s };
8  |  end
9  end
```

у следећој секцији. Излаз алгоритма представља валидни распоред компилационих јединица Σ, чија је дефиниција дата у секцији 2.2.

Алгоритам одржава ред метода за превођење, коме се на почетку додаје улазна метода програма. У свакој итерацији дохвата се метода из реда и распоређује за компилацију. Једна од фаза превођења појединачне методе подразумева оптимизацију инлајновања. Метода која се дохвата из реда постаје корен јединице компилације у текућој итерацији, а све методе које се инлајнују у ту методу оформљавају дату компилациону јединицу, према опису у секцијама 2.2, 3.4 и 6.3. Како алгоритам не разликује делове кода према фреквенцији њиховог извршавања, буџет који се додељује методама за превођење је исти за све јединице компилације. Улазни профили се користе у оптимизацији инлајновања, као што је описано у секцији 3.5. Након завршетка превођења неке методе и формирања компилационе јединице C, све методе које се позивају из C, а претходно нису преведене или се не налазе у реду за чекање, додају се у ред. Алгоритам се завршава када у реду више нема метода за превођење, тј. када је целокупан програм преведен (линија 2 алгоритма 6.1).

6.2 Улаз и излаз алгоритма

Улаз алгоритма. Улаз алгоритма састоји се од графа позива формираног за програм који се преводи и скупа делимично контекстно осетљивих профила. Свака метода $s \in S$ неког програма одговара једном чвору у графу позива $G = (S, E \subseteq S \times S)$, а позив методе s_2 из друге методе s_1 у оквиру тог програма одговара усмереној грани $s_1 \to s_2$ овог графа. *Делимично контекстно осетљиви профили* садрже мапирања контекста на број извршавања догађаја на који се тај профил односи. Сваки контекст јесте, заправо, листа локација у програмском коду $\ell_1, \ell_2, \ldots, \ell_n$ које се налазе на стеку позива функција програма. У случају делимично контекстно осетљивих профила контекст не садржи све локације са овог стека, већ само неки одређени суфикс. Чланови контекста $\ell_1, \ell_2, \ldots, \ell_{n-1}$ представљају локације позива у коду са које претходна метода позива наредну на стеку позива, док ℓ_n означава посебну локацију у позваној методи са врха стека на којој се налази догађај за који се прикупља профил. Број извршавања који је у корелацији са одређеним контекстом означава колико пута се дати садржај стека

```
void foreach(int[] xs, int->void f) {
  for (int i = 0; i < xs.length; i++)
    f.apply(xs[i]);
}

void main(int[] args) {
  int min = Integer.MAX_VALUE;
  foreach(args, x -> if (x < m) m = x);
  int max = Integer.MIN_VALUE;
  foreach(args, x -> if (x > m) m = x);
}
```

Листинг 6.1: Поједностављени пример улаза

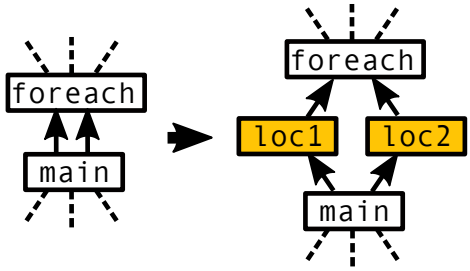
појавио током извршавања програма. Примери графа позива као и улазних профила дати су у секцији 2.3.

Поједностављење улаза алгоритма. Скуп грана у графу позива биће дефинисан као подскуп Декартовог скупа $S \times S$, где S представља скуп метода програма. Ово имплицира да нека метода s_1 може позивати другу методу s_2 са највише једне локације у коду. У реалним програмским кодовима, метода s_1 може садржати више локација позива методе s_2. У наставку ће бити приказано да претходно уведено ограничење јединствене гране за пар чворова није од значаја захваљујући томе што се граф позива са паралелним гранама (енг. *multi-edged call graph*) може свести на једноставнију апстракцију графа позива.

Пример. Програмски код из листинга 6.1 садржи измењену верзију програма приказаног у листингу 2.1. Уместо позива метода `min` и `max` из `main` методе, тела ових метода су ручно уграђена (инлајнована) у тело `main` методе. Самим тим, `main` садржи две локације са које се позива иста метода – `foreach`.

Дискусија. Као што је приказано у примеру, конкретна имплементација алгоритма мора прихватати на улазу граф позива у коме сваки пар чворова може бити повезан произвољним бројем грана. Ипак, то не доводи до значајне модификације претходне дефиницију улаза алгоритма захваљујући томе што се граф позива са паралелним гранама може трансформисати у граф са јединственим гранама између парова чворова. Ово је значајно и због мапирања излаза алгоритма 6.2 у модел графа са паралелним гранама.

Сваки чвор V графа позива са паралелним гранама преводи се у подграф коме је V почетни чвор. За сваку грану $V \rightarrow_i U$ која означава позив методе репрезентоване чвором U из методе репрезентоване чвором V на локацији i, умеће се виртуелни чвор



Слика 6.1: Трансформација улазног мултиграфа

$V_{i,U}$ и одговарајуће усмерене гране $V \to V_{i,U}$ и $V_{i,U} \to U$. Ова трансформација одговара примени издвајања сваке локације позива у засебну методу која служи као мост (енг. *bridge method*). На слици 6.1 приказано је додавање два виртуелна чвора `loc1` и `loc2` у методу `main` приказану у листингу 6.1.

Распоред превођења који је резултат примене алгоритма компилације (алгоритам 6.2) може укључивати виртуелне чворове у нотацији одговарајућих грана $V \to V_{i,U}$ и $V_{i,U} \to U$. Како сваки чвор $V_{i,U}$ показује на тачно један оригиналан чвор U, сваку грану $V \to V_{i,U}$ могуће је заменити граном $V \to_i U$. Слично, како на сваки виртуални чвор $V_{i,U}$ показује тачно један оригиналан чвор V, сваку грану $V_{i,U} \to U$ могуће је заменити граном $V \to_i U$. Овако трансформисан граф распореда компилације може садржати више грана на свакој локацији i у случају виртуелних позива метода. Различити чворови U тада представљају конкретне имплементације датог виртуелног позива.

Закључак који је се изводи је да сваки улазни граф позива може бити поједностављен и трансформисан у граф у коме су свака два чвора повезана највише једном усмереном граном. У наставку ће бити коришћен поједностављен модел у формализацији алгоритма због једноставности, али је показано да се и коришћењем таквог модела решава проблем који је еквивалентан конкретном проблему компилације.

Излаз алгоритма. Излаз алгоритма превођења представља валидан распоред превођења и мапирање компилационих јединица са буџетом компилације одвојеним за превођење те јединице. Дефиниција валидног распореда превођења, као и примери неколико различитих распореда, валидних и невалидних, приказани су у секцији 2.2.

6.3 Алгоритам *PRINC*

У овој секцији ће бити представљен нови алгоритам за унапређење распореда превођења и оптимизације инлајновања – *PRINC* [41]. Најпре ће бити приказана дефиниција алгоритма уз издвајање главних корака. Ради лакшег разумевања алгоритма, поред дефиниције алгоритма 6.2, биће илустрован и рад алгоритма *PRINC* на улазном примеру. Након тога ће сваки издвојени корак алгоритма бити прецизније дефинисан и објашњен на примерима у циљу бољег разумевања комплетног алгоритма.

Опис алгоритма *PRINC*. Алгоритам користи граф позива и скуп делимично контекстно осетљивих профила да би закључио који делови програма се често извршавају, како би се на основу тога више оптимизационих напора инвестирало у превођење тих секција кода. Како би се ово спровело, алгоритам одваја компилационе јединице у два скупа – скуп често извршаваних („врућих”) и ређе извршаваних („хладних”) компилационих јединица. У оквиру овог истраживања мапирање компилационих јединица на буџет компилације је бинаран, тј. ако је нека компилациона јединица означена као често извршавана, за њено превођење се одваја већи буџет, док за све оне јединице које су означене као хладне, буџет остаје исти. За перформансе програма је кључно да преведени код који се често извршава буде добро оптимизован јер се, са једне стране, у њему проводи највише времена у току извршавања програма, а са друге стране, подједнака оптимизација целокупног кода доводи до незанемарљивог увећања генерисаног кода, а такође утиче и на време потребно за превођење. Исправном идентификацијом често

Алгоритам 6.2: Нови алгоритам за превођење

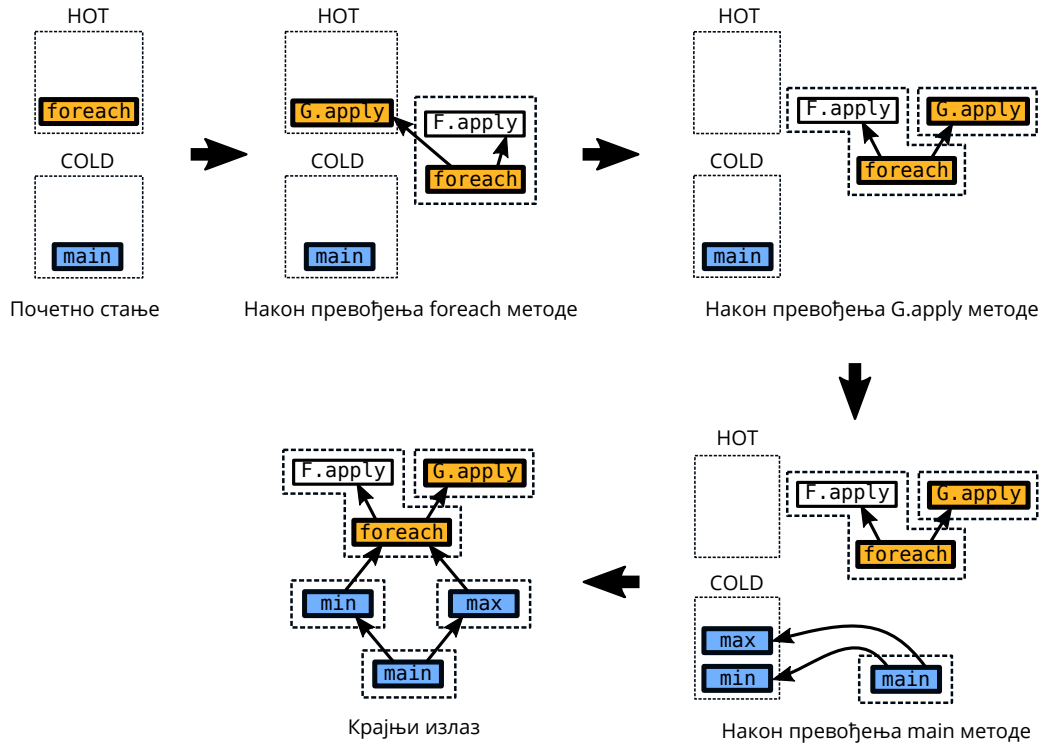
```
   Input  : call graph G, entry-point e, profile Π
   Output : compilation schedule Σ, budget B
 1 Σ = ∅; B = ∅; cold = { e } ;
 2 hot = DetectHot(G, Π);
 3 while hot ≠ ∅ do
 4 |   hot = hot \ { s_C } : s_C ∈ hot ;
 5 |   C = InlineHot(s_C);
 6 |   Σ = Σ ∪ { C }; B = B ∪ { C };
 7 |   cold = cold \ { s_C } ;
 8 |   for s ∈ callees(C) \ (Σ ∪ hot) do
 9 |   |   if IsHot(C) then hot = hot ∪ { s };
10 |   |   else cold = cold ∪ { s };
11 |   end
12 end
13 while cold ≠ ∅ do
14 |   cold = cold \ { s_C } : s_C ∈ cold ;
15 |   C = InlineCold(s_C);
16 |   Σ = Σ ∪ { C } ;
17 |   for s ∈ callees(C) \ (Σ ∪ cold) do
18 |   |   cold = cold ∪ { s };
19 |   end
20 end
```

извршаваних делова кода и њиховом оптимизацијом, увећање генерисаног кода које се односи на често извршаване методе програма не утиче значајно на укупно увећање извршног кода програма. Како буџет компилације директно утиче на количину примене оптимизација на појединачне компилационе јединице, ово чини мотивацију за раздвајање буџета компилације.

Предложени алгоритам одржава два реда, један за често извршаване јединице компилације и један за све преостале (ређе извршаване). На почетку, алгоритам бира иницијални скуп често извршаваних метода и смешта их у предодређени ред. Овај ред ће у наставку текста бити референсан као *H peg*, док ће ред у који се смештају ретко извршаване јединице бити референсан као *C peg*. Затим се из H реда итеративно узима по једна метода која постаје корен компилационе јединице у оквиру које се спроводи инлајновање док се не формира комплетна јединица компилације. Ако, након примене оптимизације инлајновања у оквиру формиране компилационе јединице, преостану позиви ка другим методама које претходно нису додате у одговарајући ред, тада алгоритам смешта сваку од таквих метода у један од два реда. Најпре се распоређују често извршаване методе за компилацију, а тек након што су све овакве методе распоређене и цео H ред је испражњен, прелази се на компилацију преосталих метода. Скуп ретко извршаваних метода у највећем броју случајева укључује и почетне методе програма. Хладне методе се распоређују за компилацију на сличан начин све док се ред не испразни.

Пример извршавања. Рад алгоритма биће приказан на примеру компилације са слике 6.2. Често извршаване методе на слици су означене жутом бојом и смештају се у

оивичен ред са називом „Hot”, док су ретко извршаване методе означене плавом бојом и смштају се у ред „Cold”. На почетку, алгоритам смешта методу `foreach` у листу често извршаваних метода (ред H), а методу `main` у ред C. Као што је већ речено, ради се о улазној методи програма, а она се извршава само једном. Затим се, према алгоритму, `foreach` метода уклања из реда, у њу се инлајнује позив `F.apply`, а потом се, на основу преосталог буџета оптимизације инлајновања, доноси одлука да није могуће инлајновати и позив `G.apply`. Алгоритам, ипак, има информацију да је позив преостале методе `G.apply` значајан за перформансе програма. Другим речима, овај позив је означен је као често извршаван, тако да ће `G.apply` бити додата у ред често извршаваних метода. Након што се тај ред испразни, тј. све често извршаване методе се компајлирају, алгоритам распоређује прву методу из реда хладних, тј. `main`. У конкретном примеру, на основу одлука оптимизације, ниједна метода неће бити инлајнована у `main`, па ће `min` и `max` бити додате у ред C. Битно је нагласити да алгоритам никада неће инлајновати значајну компилациону јединицу, тј. често извршавану у неку другу ретко извршавану, тј. хладну – често извршаване компилационе јединице су већ у потпуности формиране. У овом примеру то значи да `foreach` метода неће бити инлајнована ни у `min` нити у `max` методу.



Слика 6.2: Пример распореда превођења

Дискусија. Претходни опис алгоритма изоставља неколико битних „детаља” које је потребно разјаснити. На почетку, потребан је скуп конкретних корака који морају бити имплементирани како би се пронашао скуп значајних метода. Након тога, методе морају бити проширене тако да формирају компилационе јединице применом конкретне полисе инлајнера. Након формирања често извршаваних компилационих јединица, све методе које нису инлајноване у оквиру тих јединица, а које се позивају из њих, морају

се разврстати између два реда метода. Другим речима, све такве методе алгоритам мора да класификује као често извршаване или ретко извршаване и распореди их за компилацију у складу са тим.

У алгоритму 6.2 претходни кораци издвојени су у методама DetectHot, InlineHot, InlineCold и IsHot, респективно. Функционисање ових метода ослања се на наменски дефинисане структуре података које чине модел, а пре свега стазе трагова. *Стаза трагова* је структура података базирана на стаблима, која мапира методе s и њихове контексте са листом често извршаваних метода које позива s из истог контекста. Нека метода се сматра често извршаваном уколико процењено време проведено извршавањем само те методе (без времена проведеног извршавањем метода које она позива) превазилази неки унапред одређени фиксни проценат ψ времена проведеног извршавањем целокупног програма, или, уколико та метода транзитивно позива неку често извршавану методу, до границе одређене буџетом превођења. Сврха ове структуре података је да усмерава превођење у графу позива неког програма ка оним деловима кода у којима је проведено највише времена током извршавања. Структура података стаза трагова је формално дефинисана у секцији 5.1. Остатак овог поглавља садржи детаље ових издвојених метода. Самим тим, појединости алгоритма су издвојене у три главне компоненте:

1. DetectHot: метода која формира стазе трагова на основу улаза алгоритма, тј. графа позива G и скупа делимично контекстно осетљивих профила P, (биће описана у секцији 6.4).

2. InlineHot и InlineCold: измењени алгоритам оптимизације инлајновања који, на основу информација о често извршаваним деловима кода из стаза трагова, усмерава одлуке инлајнера (биће описано у секцији 6.5).

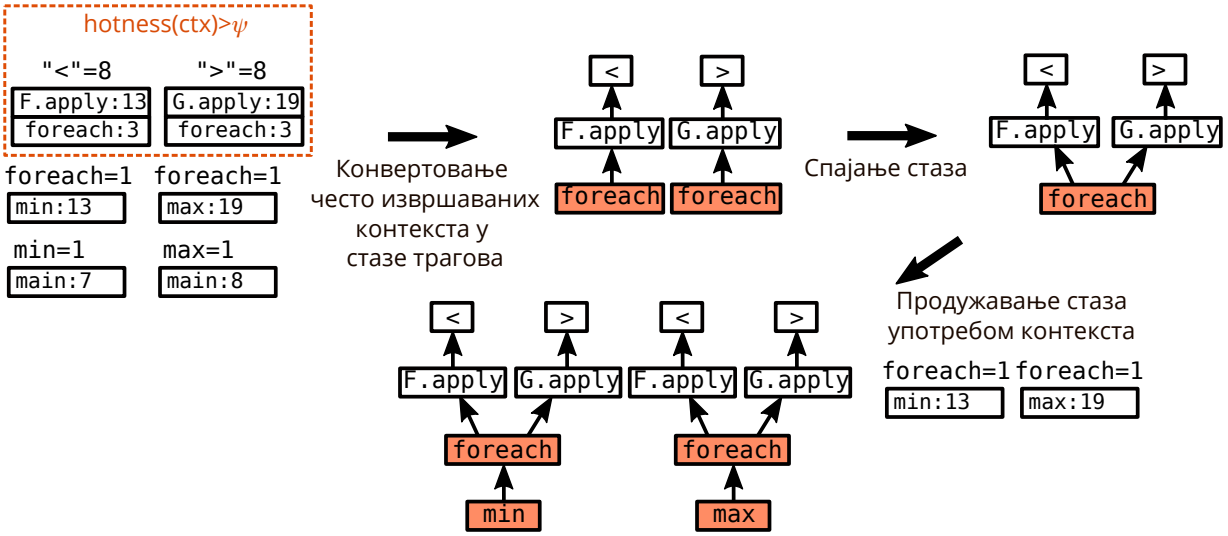
3. IsHot: метода која, након формирања неке компилационе јединице, раздваја преостале методе које се позивају из ње на често извршаване и ретко извршаване на основу информација из стаза трагова (биће дефинисана у секцији 6.6).

Табела 5.1 сумира кључне симболе који су уведени до овог тренутка, а који су директно коришћени у описима алгоритама у овој секцији. Ова табела укључује и симболе који ће у наставку бити коришћени како би се дефинисао алгоритам детекције често извршаваних делова кода.

6.4 Алгоритам за детекцију често извршаваног кода

Утврђено је да код из алгоритма 6.2 раздваја граф позива на регионе према учестаности њихових извршавања. „Врући" региони графа позива састоје се од често извршаваних делова кода, заједно са контекстом довољно великим да укључи све релације неопходне за оптимизације кода. Улога уведених структура стаза трагова у овом раду је да ограничи регионе често извршаваних делова кода.

Стазе које оивичавају регионе често извршаваног кода, морају бити конструисане на основу профила који одговарају извршавању програма који се преводи у оптимизовану



Слика 6.3: Поступак детекције често извршаваног кода

форму. Уколико би потпуно контекстно осетљиви профили били доступни, да би се формирале стазе трагова довољно је трансформисати оне контексте који се на стеку позива функција појављују више пута од дефинисане граничне вредности и потом спојити добијене стазе. Како улаз алгоритма чине делимично контекстно осетљиви профили, овакав поступак није довољан како би се маркирала значајна количина често извршаваног кода у циљу побољшања перформанси. Најчешће извршавани контексти садржани у парцијалним профилима су често превише кратки да би се повезале регионе значајних делова кода тако да се ова информација може искористити у оптимизацијама. Због тога се иницијално формиране стазе морају повећавати све док покривени делови стабла позива не буду довољно велики како би оптимизације које следе имале довољно информација и могле да унапреде перформансе програма. Самим тим, алгоритам мора да превазиђе неколико изазова: одабир релевантног скупа иницијалних контекста, избор контекста који доприносе хеуристици приликом спекулативне експанзије стазе, избор момента терминирања примене операција продужавања $\odot$ и ширења $\circledast$ стазе. Ова компонента алгоритма је у овом раду означена као *детекција често извршаваних делова кода* (енг. *hot-code detection*).

Опис алгоритма. Алгоритам за детекцију често извршаваних делова кода на почетку врши избор скупа иницијалних контекста и на основу њих формира објекте стаза трагова. Формиране стазе се спајају када год је то могуће, како би се груписали већи региони кода. Након овога, следећи кораци се понављају док се не испуни услов завршетка алгоритма:

1. бира се најзначајнија стаза из скупа,
2. изабрана стаза се експандује оним контекстима, из којих, када се корена метода стазе позива, стаза описује значајан фрагмент кода за перформансе целог програма
3. експандоване стазе се или припајају некој другој стази у скупу, или се враћају у истом облику у скуп, уколико спајање није могуће.

Пример извршавања. На слици 6.3 је приказан поступак детекције често извршаваног кода на примеру. Почетни скуп профила, који се на слици налази на левој страни је делимично контекстно осетљив, тако да су контексти неких профила дужине $n = 1$, а неких дужине $n = 2$. На почетку алгоритам бира оне профиле код којих број појављивања контекста на стеку позива функција премашује предефинисану граничну вредност ψ. У примеру са слике биће узето да су то профили са контекстима `foreach`→`F.apply`→`‘<‘` и `foreach`→`G.apply`→`‘>‘`. Изабрани контексти се потом трансформишу у почетни скуп стаза трагова, тј. стабала. У следећем кораку, свака стаза ξ се припаја чворовима $\eta_{s_1,\ldots,s_n}$ других стаза уколико је метода која одговара чворовима $\eta_{s_1,\ldots,s_n}$ иста као и метода која одговара корену стазе ξ, тј. $sub(root(\xi)) = sub(\eta_{s_1,\ldots,s_n})$. У датом примеру, прва стаза са кореном методом `foreach`, која позива `F.apply` качи се на другу стазу са истом кореном методом, али која у овом случају позива `G.apply`.

Алгоритам затим врши претрагу над профилима како би пронашао могуће позиваоце методе `foreach` и утврђује да су методе `min` и `max` најчешћи позиваоци у коду. На основу тога, стаза са кореном који одговара методи `foreach` се продужава употребом свих корисних контекста који се односе на честе позиваоце. Претходне две операције, спајање и експанзија, понављају се итеративно све док алгоритам не одреди да је потрошен буџет за даљу модификацију стабла. У конкретном примеру, алгоритам престаје са радом након стварања две стазе чији корени одговарају `min` и `max` методама.

Значајно је приметити да претходни пример садржи довољно дугачке контексте који омогућавају статичкој анализи да закључи да се `G.apply` никада неће позвати из процедуре `min` и слично, да се `F.apply` никада неће позвати из `max`. Другим речима, формиране стазе трагова нису увек угњеждене у минимално стабло позива. Алгоритам предложен у овом раду не уклања недостижне позиве у фази детекције често извршаваног кода зато што преводиоци углавном примењују оптимизације које уклањају такве позиве након инлајновања позваних метода у једну компилациону јединицу [112, 113, 114, 115, 116]. Штавише најсавременији (енг. *state-of-the art*) алгоритми за инлајновање користе анализу тока података (енг. *dataflow analyses*) како би поједноставили стабло позива пре него што се инлајновање догоди [117, 118, 119, 5]. Другим речима, у наредним фазама се обезбеђује да се не спроведе инлајновање позива `G.apply` у контекст процедуре `min`, као ни инлајновање позива `F.apply` у контекст процедуре `max` упркос томе што су обе информације садржане у скупу стаза. Поједностављивање стаза није неопходно у фази детекције често извршаваних делова кода захваљујући алгоритму за инлајновање на које се ослања предложено решење, а који одсеца гране стаза трагова које се не могу реализовати касније у процесу.

Формална дефиниција алгоритма. Након описа алгоритма на вишем нивоу и демонстрације рада на улазном примеру, биће дата и формална дефиниција на основу приказаног псеудокода алгоритма 6.3. Процедура DetectHot на почетку бира скуп значајних (често извршаваних) контекста из профила П и, као што је речено, на основу њих формира почетни скуп стаза трагова T_0. У алгоритму, овај поступак приказан је на линији 1. Конкретни кораци за креирање скупа стаза апстраховани су методом InitialTrails. Како стазе у скупу T_0 одговарају профилима и нису претпроцесиране, најчешће је већ након креирања почетних стаза могуће спојити неке од њих са другим

Алгоритам 6.3: DetectHot метода

```
   Input  : program call graph G, profile Π
   Output : hot roots H, trail set T
 1 T_0 = InitialTrails(Π);
 2 T = (fix ⍟∇_inf)(T_0);
 3 F = ∅;
 4 while ¬ DetectionDone(T, F) do
 5 |   τ = TopTrail(T \ F);
 6 |   Γ = CallingContexts(τ, Π, G);
 7 |   T_{τ,Γ} = { τ ⊙ c : c ∈ Γ, Accept(τ ⊙ c) };
 8 |   if T_{τ,Γ} = ∅ then F = {τ} ⍟∪ F ;
 9 |   else
10 |   |   T = T \ {τ};
11 |   |   T = T_{τ,Γ} ⍟∪ T;
12 |   end
13 end
14 H = { sub(root(τ)) : τ ∈ T };
```

стазама из скупа, као што је и био случај у претходном примеру. Спајање над иницијалним скупом стаза представљено је операцијом fix $\overset{⍟}{\nabla}_{\text{inf}}$ на линији 2 у алгоритму. Оно подразумева итеративно узимање једне стазе из скупа T_0 и припајање те стазе свим преосталим стазама којима је то могуће на основу операције кореног самоспајања (*self-root-graft operation* $\overset{⍟}{\nabla}_{\text{inf}}$ из једначине 5.13). Резултат ове операције је скуп стаза T.

Алгоритам потом креира празан скуп стаза F, у коме ће се по завршетку детекције налазити стазе у финалној форми. Све док услов DetectionDone не постане испуњен, алгоритам репретитивно бира стазу τ на основу постављене полисе TopTrail апстраховане на линији 5 и позива методу CallingContexts која идентификује скуп Γ контекста позива методе која одговара корену стазе τ – $sub(root(\tau))$ на линији 6. Стаза трагова τ се експандује употребом сваког од контекста $c \in \Gamma$ како би се добили дужи контексти $\tau \odot c$. Алгоритам задржава само оне продужене стазе $\tau \odot c$ које задовољавају услов дефинисан Accept предикатом (линија 7). Прихваћене стазе чине скуп експандованих стаза $T_{\tau,\Gamma}$. Уколико је овај скуп празан, то значи да стаза τ није могла бити даље продужена употребом било ког од контекста позива њене корене методе, па ће над њом и скупом коначних стаза F бити примењена операција спајања шума стаза $\overset{⍟}{\cup}$ (*union-grafting operation*), дефинисана у једначини 5.10 (линија 8). У супротном, иста операција се примењује, али над скуповима $T_{\tau,\Gamma}$ и T јер се продужене стазе могу потенцијално експандовати даље у наредним итерацијама алгоритма. За сваку стазу $\tau \in T$ неопходно је да се односи на јединствену корену методу $sub(root(\tau))$ у том скупу. Другим речима ниједне две стазе из скупа T не смеју имати исту корену методу. Ово се постиже тако што алгоритам примењује операцију спајања када год је то могуће. Самим тим се стазе са истим кореним методама спајају у једну. Спајање $T_{\tau,\Gamma}$ и T је тако дефинисано операцијом спајања шума стаза $\overset{⍟}{\cup}$. Скуп често извршаваних метода H изводи се директно из скупа T на линији 14.

6.4.1 Стратегије детекције често извршаваног кода

Алгоритам 6.3 у коме је приказан предложени поступак проналажења често извршаваних делова кода укључује неколико метода које дефинишу поступак детекције. То су методе InitialTrails, DetectionDone, TopTrail, CallingContexts и Accept. Свака комбинација различитих имплементација ових метода зове се полиса детекције често извршаваног кода (енг. *hot-code-detection policy*). Различите полисе резултују различитим понашањем алгоритма, а у наставку ће бити приказана конфигурација изабрана у овом раду.

Метода InitialTrails. Метода InitialTrails формира почетни скуп стаза трагова на основу профила Π. Како би у овај скуп ушли само они профили који описују делове кода значајне за извршавање целокупног програма, у једначини 6.1 је дефинисана константа $\psi \in [0, 1]$. Допринос засебног профила h_I у односу на фреквенцију извршавања свих делова кода мора бити бар ψ како би био одабран за конструкцију иницијалног скупа стаза трагова. Ради постизања што бољих перформанси, тражи се најбоља вредност константе ψ. Поступак проналажења ове вредности приказан је у секцији 8.7.

$$\begin{aligned} \textsc{InitialTrails}(\Pi) \equiv \{(N, E, \gamma) : (\langle \ell_1, \dots, \ell_n \rangle, h_I) \in \Pi \wedge \frac{h_I}{\sum\limits_{(L,h)\in\Pi} h} > \psi\} \\ \textit{где је} \quad N = \{\eta_{\ell_1}, \dots, \eta_{\ell_1,\dots,\ell_n}\} \quad E = \{\eta_{\ell_1} \to \eta_{\ell_1,\ell_2}, \dots, \eta_{\ell_1,\dots,\ell_{n-1}} \to \eta_{\ell_1,\dots,\ell_n}\} \quad \gamma = \{\eta_{\ell_1}\} \end{aligned} \tag{6.1}$$

Метода DetectionDone. Ова метода представља услов за прекидање алгоритма детекције, који ће бити испуњен када су све стазе трагова из скупа T достигле свој финални облик (оне не могу бити даље продужаване или спајане), тј. пребачене су у скуп стаза F:

$$\textsc{DetectionDone}(T, F) \equiv T \setminus F = \varnothing \tag{6.2}$$

Услов за заустављање поступка детекције подразумева да скуп F монотоно расте, као и да скуп T у неком тренутку престане са растом. Обе ставке су осигуране избором предиката Accept, што ће бити описано нешто касније.

Метода CallingContexts. Корен сваке стазе трагова одговара кореној методи неке компилационе јединице, која може бити продужена употребом неког контекста из кога та метода може бити позвана у програмском коду. Скуп свих могућих контекста позива за дату методу може бити одређен на основу анализе профила Π применом следећег поступка. Потребно је идентификовати подскуп $\Pi|_c$ из скупа профила Π који се односе на специфичну локацију c позива дате методе. За конкретну стазу τ, скуп *callerProfiles* одређен је као подскуп улаза из скупа профила $\Pi|_c$, чији се контексти завршавају методом s_n, и из којих се позива метода која одговара корену стазе $root(\tau)$

у графу позива G. У једначини 6.3 $\eta_1 \to \eta_2 \in G$ где је $G = (N, E)$ је скраћено за $\eta_1 \to \eta_2 \in E$.

$$\begin{aligned} &\mathit{callerProfiles}((N, E, \gamma), \Pi, G) \equiv \{(\langle s_1, \ldots, s_n \rangle, h) : (\langle s_1, \ldots, s_n \rangle, h) \in \Pi|_c \wedge s_n \to \inf_{E^*} N \in G\} \\ &\textsc{CallingContexts}(\tau, \Pi, G) \equiv \{L : (L, h) \in \mathit{callerProfiles}(\tau, \Pi, G)\} \end{aligned} \tag{6.3}$$

Укупно време проведено у делу кода представљеном стазом трагова τ изражено је сумом времена проведеног у тој стази када је она позвана из сваког контекста из кога је могуће позвати корену методу стазе τ. Када се продужава стаза τ, идеја је задржати оне продужене стазе $\tau \odot c$ за које су коришћени контексти c за које важи да се у делу програма описаним стазом τ проводи значајна количина времена када је она позвана из контекста c. Значајно је напоменути да информације о времену проведеном у делу кода који стаза описује када је позвана из неког контекста нису укључене у профиле, тј. да профили садрже искључиво бројеве извршавања за појединачне тачке програма. Управо због тога, један од значајних доприноса рада јесте естимација процента извршавања неке стазе када је она позвана из специфичног контекста, тј. процена проведеног времена у продуженој стази трагова $\tau \odot c$.

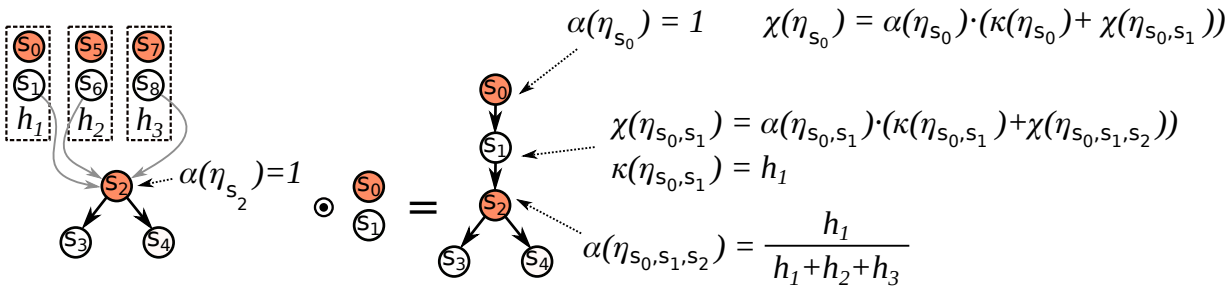
Из наведеног разлога, метода CallingContext додатно израчунава *фактор удела* (енг. *attenuation factor*) $a_c(\tau) \in [0, 1]$ за сваки контекст позивања $c = \langle s_1, \ldots, s_n \rangle$, који представља апроксимацију удела броја извршавања стазе τ када се она позива из контекста c у односу на укупан број позивања ове стазе. У једначини 6.4 бројилац је број извршавања једног засебног контекста $c = \langle s_1, \ldots, s_n \rangle$ који позива $root(\tau)$, а именилац сума извршавања свих контекста L који позивају корену методу стазе τ ($root(\tau)$).

$$a_{s_1,\ldots,s_n}(\tau) \equiv \frac{h_{s_1,\ldots,s_n}}{\sum_{(L,h) \in \mathit{callerProfiles}(\tau,\Pi,G)} h} \quad \textit{где је} \quad (\langle s_1, \ldots, s_n \rangle, h_c) \in \Pi \tag{6.4}$$

Фактор $a_c(\tau)$ се користи ради процене времена проведеног у делу кода ограниченим продуженом стазом. Важно је нагласити да се ради само о естимацији на основу делимично контекстно осетљивих профила. Такође, бројачи h_c у бројиоцу и h у имениоцу разломка, не садрже заправо информацију о времену проведеном у неком делу кода, већ колико пута се тај део кода извршио. Иако се ове две информације могу разликовати када се неки део кода позива мали број пута, али се у њему проводи доста времена, у највећем броју случајева се показује да доступни бројачи служе као добра апроксимација (енг. *proxy*) за количину проведеног времена у датим позивима. За илустрацију израчунавања фактора $a_c(\tau)$ дат је пример са слике 6.4.

На слици 6.4, стази $s_3 \leftarrow s_2 \rightarrow s_4$, која је приказана на левој страни, одговарају контексти позивања $s_0 \to s_1$, $s_5 \to s_6$ и $s_7 \to s_8$, чији бројеви извршавања су респективно h_1, h_2 и h_3. Фактор удела за ова три различита контекста који одговарају методи s_2 израчунати су према једначини 6.4.

Како би се израчунао укупан број извршавања дела кода описаног неком стазом, потребно је сабрати бројеве извршавања свих трагова (чворова стазе), али се то су-



Слика 6.4: Процена времена извршавања дела кода обухваћеног стазом

мирање мора извршити тежински употребом фактора удела. За ову сврху, у тези су дефинисане три функције: функција удела тачке спајања α (енг. *graft-point attenuation*), која бележи фактор удела када се стаза продужава, функција учестаности извршавања трага κ, која бележи број извршавања засебног чвора стабла, и функција укупног броја извршавања стазе χ. На слици, фактор удела корена $\alpha(\eta_{s_2})$ стазе $s_3 \leftarrow s_2 \rightarrow s_4$ је 1, али након продужавања стазе, удео стазе се смањује и постаје $^{h_1}/_{h_1+h_2+h_3}$. Следеће, број извршавања појединачног чвора κ за чвор η_{s_0,s_1} наслеђује се од контекста позива h_1. Број извршавања стазе χ рачуна се рекурзивно сумирањем броја извршавања припадајућих потомака и датог чвора и множењем добијене вредности са фактором удела α.

6.4.2 Фреквенција извршавања стазе трагова

Наредне методе служе да рангирају стазе трагова на основу очекиване учестаности њиховог извршавања. Стазе са вишом фреквенцијом извршавања имају већу вероватноћу да буду разматране за експанзију и треба их продужавати употребом више различитих контекста позивања. *Фреквенција стазе трагова* је функција $\chi : \mathbb{T}(P) \rightarrow \mathbb{R}_0^+$ која мапира стазу на неку ненегативну реалну вредност. Пре дефинисања функције χ биће дефинисане претходно поменуте функције α и κ.

Дефиниција 1 Удео тачке спајања *стазе трагова* τ *је функција* $\alpha_\tau : N \rightarrow [0, 1]$ *која мапира сваки чвор на реалну вредност између* 0 *и* 1. *Улога* α_τ *је да умањи фреквенцију неких делова стазе када се она екстендује употребом неког контекста позивања. За сваку стазу* $\tau = (N, E, \gamma)$, $\alpha_\tau(\eta) = 1$ *за све чворове који нису тачке спајања, тј.* $\eta \notin \gamma$. *За тачке спајања* $\eta \in \gamma$, *алгоритам инкрементално конструише* α_τ *када се формира стаза* τ *према следећим правилима:*

- *За све стазе који су иницијално формиране на основу скупа профила* $(\langle \ell_1, \ldots, \ell_n \rangle, h) \in \Pi$ *(према једначини 6.1), алгоритам поставља* $\alpha_\tau(\eta_{\ell_1}) = 1$. *Другим речима, удео фреквенције нових стаза је* 1.
- *За сваку стазу* $\tau = (N, E, \gamma)$ *која је добијена применом операције спајања* $\xi \circledast_{\eta_{s_1,\ldots,s_n}} \rho$ *(на основу једначине 5.4), где* $\xi = (N_\xi, E_\xi, \gamma_\xi)$ *и* $\rho = (N_\rho, E_\rho, \gamma_\rho)$: *ако* $\eta \in \gamma_\rho$, *тада* $\alpha_\tau(\eta) = \alpha_\rho(\eta)$; *у супротном ако* $\eta_{s_n,d_2,\ldots,d_m} \in \gamma_\xi$, *тада важи* $\alpha_\tau(\eta_{s_1,\ldots,s_n,d_2,\ldots,d_m}) = \alpha_\xi(\eta_{s_n,d_2,\ldots,d_m})$. *Односно, удео се наслеђује на основу улаза операције.*

- *За сваку стазу трагова $\tau = (N, E, \gamma)$ која је формирана применом операције продужавања $\rho \odot c$, где је $\rho = (N_\rho, E_\rho, \gamma_\rho)$ и $c = \langle \ell_1, \dots, \ell_n \rangle$, алгоритам израчунава фактор удела $a_c(\rho)$ за сваки контекст c (према дефиницији из једначине 6.4), који је одређен у* CallingContexts *процедури, на линији 6 алгоритма 6.3. Затим, за корен који је у исто време и тачка спајања важи $\alpha_\tau(\inf_{E^*} N) = 1$; за чвор који се претходно налазио у корену $\alpha_\tau(\eta_{\ell_1,\dots,\ell_n,\text{суб}(\inf_{E^*_\rho} N_\rho)}) = a_c$; а за све остале чворове $\eta_{s_1,\dots,s_m} \in \gamma_\rho$, $\alpha_\tau(\eta_{\ell_1,\dots,\ell_n,s_1,\dots,s_m}) = \alpha_\rho(\eta_{s_1,\dots,s_m})$ Односно, α_τ тачке спајања која је претходно била корен поставља се на вредност a_c, а удео преосталих тачака спајања остаје непромењен.*

Дефиниција 2 Фреквенција трага *неке стазе трагова $\tau = (N, E, \gamma)$ је функција $\kappa_\tau : N \to \mathbb{N}_0$, која мапира сваки чвор на одговарајући процењени број извршавања. Алгоритам конструише κ_τ када је τ креиран на следећи начин:*

- *За све стазе креиране на основу улаза у профилима $(\langle \ell_1, \dots, \ell_n \rangle, h) \in \Pi$ (видети једначину 6.1), фреквенција чвора на највећој дубини је $\kappa_\tau(\eta_{\ell_1,\dots,\ell_n}) = h$, и $\kappa_\tau(\eta) = 0$ за све друге чворове.*

- *За сваку стазу $\tau = (N, E, \gamma)$ која је конструисана након операције спајања $\xi \circledast_{\eta_{s_1,\dots,s_n}} \rho$ (видети једначину 5.4), где $\xi = (N_\xi, E_\xi, \gamma_\xi)$ и $\rho = (N_\rho, E_\rho, \gamma_\rho)$: ако је $\eta_{s_n,c_2,\dots,c_m} \in N_\xi$ и $\eta_{s_1,\dots,s_n,c_2,\dots,c_m} \in N_\rho$, тада је $\kappa_\tau(\eta_{s_1,\dots,s_n,c_2,\dots,c_m}) = \kappa_\xi(\eta_{s_n,c_2,\dots,c_m}) + \kappa_\rho(\eta_{s_1,\dots,s_n,c_2,\dots,c_m})$; ако је $\eta_{s_n,c_2,\dots,c_m} \in N_\xi$ и $\eta_{s_1,\dots,s_n,c_2,\dots,c_m} \notin N_\rho$, тада $\kappa_\tau(\eta_{s_1,\dots,s_n,c_2,\dots,c_m}) = \kappa_\xi(\eta_{s_n,c_2,\dots,c_m})$; иначе $\kappa_\tau(\eta) = \kappa_\rho(\eta)$. Другим речима, фреквенције припојених чворова су сумиране где је то било могуће, а на преосталим местима наслеђене.*

- *За сваку стазу $\tau = (N, E, \gamma)$ насталу продужавањем $\rho \odot c$, где $\rho = (N_\rho, E_\rho, \gamma_\rho)$ и $c = \langle \ell_1, \dots, \ell_n \rangle$: чворови који проистичу из ρ наслеђују фреквенције, односно $\kappa_\tau(\eta_{\ell_1,\dots,\ell_n,s_1,\dots,s_n}) = \kappa_\tau(\eta_{s_1,\dots,s_n})$. Новоформираним чворовима $\kappa_\tau(\eta_{\ell_1,\dots,\ell_i})$ фреквенција има вредност 0.*

Дефиниција 3 *Фреквенција стазе χ се затим дефинише као рекурзивна сума свих трагова у стази трагова τ, где је свако подстабло тежински укључено на основу фактора умањења тачке спајања α_τ, која представља корен датог стабла:*

$$\begin{gathered}
\chi(\tau) \equiv \alpha_\tau(\inf_{E^*} N) \cdot \left(\kappa_\tau(\inf_{E^*} N) + \sum_{\tau_c \in subtrees(\tau)} \chi(\tau_c) \right) \quad \text{где је} \quad \tau = (N, E, \gamma) \\
subtrees(\tau) = \{(N_{\eta_c}, E_{\eta_c}, \gamma_{\eta_c}) : \inf_{E^*} N \to \eta_c \in E\} \quad N_{\eta_c} = \{\eta_x \in N : \eta_c \leq_{E^*} \eta_x\} \\
E_{\eta_c} = \{\eta_x \to \eta_y \in E : \eta_x, \eta_y \in N_{\eta_c}\} \qquad \gamma_{\eta_c} = \{\eta_x \in \gamma : \eta_x \in N_{\eta_c}\}
\end{gathered} \tag{6.5}$$

Предикат TopTrail. Алгоритам се ослања на учестаност извршавања појединачних профила како би одредио које стазе су најперспективније, тј. које стазе треба продужити уколико је то могуће. Фреквенција извршавања стазе χ уводи тотални поредак (енг. *total order*) *фреквентнији* (енг. *hotter*) над скупом свих стаза $\mathbb{T}$. Уколико

је фреквенција две стазе идентична, одлучује се на основу лексикографског поретка те две стазе τ_1 и τ_2 (једначина 5.9):

$$\tau_1 <_{hotter} \tau_2 \equiv \chi(\tau_1) > \chi(\tau_2) \vee (\chi(\tau_1) = \chi(\tau_2) \wedge \tau_1 <_{lex} \tau_2) \tag{6.6}$$

Метода TopTrail тада једноставно узима априоритетнију стазу, тј. прву стазу из скупа T на основу поретка *фреквентности*:

$$\textsc{TopTrail}(T) \equiv \tau_0 \quad \textit{тако да} \quad \langle \tau_0, \ldots, \tau_n \rangle = (\!|T|\!)_{hotter} \tag{6.7}$$

Предикат Accept. Како усвојена полиса алгоритма увек спроводи операције над стазом трагова која се најчешће извршава, чиме се она увећава, пожељно је ипак процесирати и стазе које су мање фреквентне, али такође и мање по величини. Из ових разлога, предикат Accept мора спречити даљу експанзију најфреквентнијих стаза онда када оне обухвате превише кода. Да би се проценила величина стаза, у једначини 6.8 је дефинисана метрика ν која сумира величину кода свих метода које припадају стази.

$$\nu(N, E, \gamma) \equiv \sum_{\eta_{s_1,\ldots,s_n} \in N} codeSize(s_n) \tag{6.8}$$

Корисно је ограничити и количину рекурзије у стази трагова, па се овде дефинише и функција ζ, која израчунава суму чланова $2^{recursionDepth}$ за све чворове, помножену малом, експериментално утврђеном константом k_{rec} (у следећој једначини, $\delta_{x,y}$ је *Kronecker* делта функција, која узима вредност 1 када је $x = y$, а вредност 0 у супротном):

$$\zeta(N, E, \gamma) \equiv k_{rec} \cdot \sum_{\eta_{s_1,\ldots,s_n} \in N} 2^{recursionDepth(s_1,\ldots,s_n)} - 1 \quad \textit{где је} \quad recursionDepth(s_1, \ldots, s_n) = \sum_{i \in \{1,\ldots,n-1\}} \delta_{s_i,s_n} \tag{6.9}$$

Нека је релативна фреквенција изражена као број извршавања стазе $\chi(\tau)$ подељена са укупним временом проведеним у програму, односно сумом свих улаза у профилима које се односе на фреквенције извршавања делова кода из профила П. Продужена стаза се прихвата (према дефиницији из једначине 6.10) уколико је њена релативна фреквенција, умањена пеналима за рекурзију $\zeta(\tau)$, већа од функције *граничне вредности* (енг. *threshold*), која зависи од величине стазе τ. Функција граничне вредности је приказана на графику на слици 6.5. Гранична вредност је изражена малом, експериментално утврђеном константом k_h уколико је величина стазе мала $\nu(\tau)$. Дакле, мале стазе ће у скоро сваком случају бити продужене. Након што величина стазе премаши вредност $k_s + k_h \cdot (k_b - k_s)$, гранична вредност расте линеарно. Самим тим, стаза веће величине може бити експандована уколико покрива део кода који се веома често извршава, али временом, како се величина стазе приближава константи k_b, вероватноћа експанзије се приближава нули.

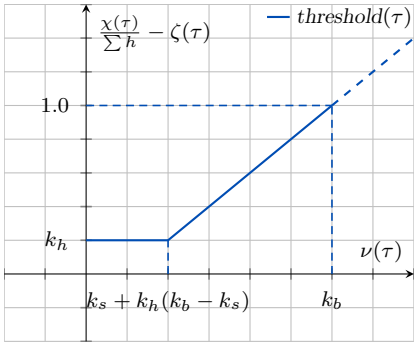
$$\textsc{Accept}(\tau) \equiv \frac{\chi(\tau)}{\sum\limits_{(L,h)\in\Pi} h} - \zeta(\tau) > \mathit{threshold}(\tau) \quad \textit{где је} \quad \mathit{threshold}(\tau) = \max\left(k_h, \frac{\nu(\tau) - k_s}{k_b - k_s}\right) \tag{6.10}$$

У табели 6.1 налазе се кључни симболи и методе, који су део алгоритма за детекцију често извршаваних делова кода и приказани су у секцији 6.4.

Табела 6.1: Преглед главних симбола и функција коришћених у секцији 6.4

Симбол	**Назив**	**Објашњење**
Π	Профили	Скуп улазних профила. Појединачан профил је пар (L, h) који чини контекст позива $L = \ell_1, \ldots, \ell_n$ и број његовог извршавања h.
$\Pi\|_c$	Профили локације позива	Подскуп улаза у Π који се односе на извршавање локација позива.
ψ	Гранична вредност извршавања	Вредност изнад које се број извршавања сматра честим.
$\textsc{InitialTrails}(\Pi)$	Полиса иницијалног скупа стаза (Јед. 6.1)	Формира се почетни скуп стаза трагова на основу профила Π.
$\textsc{DetectionDone}(\mathrm{T}, \mathrm{F})$	Полиса терминирања (Јед. 6.2)	Одлучује да ли се завршава детекција често извршаваног кода.
callerProfiles(τ, Π, G)	Скуп профила позивалаца (Јед. 6.3)	Подскуп профила из скупа $\Pi\|_c$ који позивају корену процедуру стазе трагова τ у графу позива G.
$\textsc{CallingContexts}(\tau, \Pi, G)$	Скуп контекста позивања (Јед. 6.3)	Враћа контексте који припадају скупу *callerProfiles*(τ, Π, G), тј. исте улазе профила, али без бројева извршавања.
$a_{\ell_1,\_,\ell_n}(\tau)$	Фактор удела (Јед. 6.4)	Процењени удео извршавања стазе τ када је стаза τ позвана из специфичног контекста позивања $\ell_1, \ldots, \ell_n$.
$\alpha_\tau(\eta)$	Удео тачке спајања (Деф. 1)	Мапира сваки траг η на вредност удела у опсегу $[0, 1]$.
$\kappa_\tau(\eta)$	Број извршавања трага (Деф. 2)	Мапира сваки чвор стазе η на број извршавања тог трага.
$\chi(\tau)$	Број извршавања стазе (Јед. 6.5)	Процењен број извршавања кода који покрива целокупну стазу τ.
$(\!\|T\|\!)_{hotter}$	Скуп стаза уређен према броју извршавања (Јед. 6.6)	Секвенца стаза из скупа T, уређена на основу релације *hotter*.
$\textsc{TopTrail}(T)$	Полиса најприоритетније стаза (Јед. 6.7)	Враћа најчешће извршавану стазу из скупа T.
$\nu(\tau)$	Величина стазе (Јед. 6.8)	Процена величине кода који обухвата стаза τ.
$\zeta(\tau)$	Пенал за рекурзију (Јед. 6.9)	Пенали за рекурзивне позиве метода у стази τ.
$\textsc{Accept}(\tau)$	Полиса прихватања (Јед. 6.7)	Одлучује да ли треба задржати стазу τ након експанзије.

Лема 4 *Лева страна израза* $\frac{\chi(\tau)}{\sum\limits_{(L,h)\in\Pi} h} - \zeta(\tau)$ *из једначине 6.10 је асимптотски мања од десне стране израза граничне вредности* threshold(τ) *за било који граф позива* G *и скуп профила* Π.



Слика 6.5: Функција граничне вредности *threshold*

Доказ 4 *За неки граф позива* G*, свако стабло позива (на основу једначине 2.7) је или коначно или бесконачно. Ако су стабла позива коначна, тада је величина стазе ограничена (према једначини 5.2), чиме се намеће горња граница за* $\chi(\tau)$ *на левој страни израза. Ако су стабла позива бесконачна, тада величина стазе није ограничена, али стабла позива морају садржати рекурзивне позиве. Фреквенција стазе* $\chi(\tau)$ *и њена величина* $\nu(\tau)$ *расту линеарно са сваким рекурзивним позивом, али пенали* $\zeta(\tau)$ *расту експоненцијално. Како десна страна израза не садржи пенале, она има асимптотски већу вредност од леве стране.*

Теорема 1 *Када се извршава у комбинацији са полисама за детекцију често извршаваних делова кода из једначина 6.1, 6.2, 6.3, 6.7 и 6.10, алгоритам 6.3 терминира за било који улазни граф позива* G *и скуп профила* Π.

Доказ 5 *У наставку ће бити размотрено да ли кораци који се понављају у алгоритму 6.3 терминирају. Операција* fix $\overset{\oplus}{\nabla}_{\inf}$ *терминира у коначном броју корака, као што је већ показано у леми 2.*

Дакле, слично је потребно установити и за `while` *петљу на линији 4. Како би се показало да се ова петља на крају завршава, потребно је утврдити да ће израз* $T \backslash F = \varnothing$ *из једначине 6.2 на крају бити испуњен. Размотримо скуп* $T \setminus F$*. У свакој итерацији петље бира се једна стаза* τ*, и креира се скуп* стриктно већих *стаза продужавањем стазе* τ *преко њених контекста позивања на линији 7 алгоритма. Стаза* τ *се надаље или смешта у скуп* F *и више се не разматра као најприоритетнија стаза, или се искључује из скупа* T *на линији 10. На крају остаје скуп* $T \setminus F$ *који не садржи стазу* τ*, али уместо тога може садржати* подскуп стаза које су стриктно веће од τ*. Ако се петља изврши довољан број пута, тада ће* Accept *предикат из једначине 6.10 у неком тренутку добити вредност false за све стазе у скупу* $T \setminus F$*, као последица леме 4. Дакле, тада* $T \setminus F$ *престаје да расте. Штавише, ово узрокује да скуп* F *у неком*

тренутку почне строго да расте, имајући у виду да се свака стаза τ из скупа T изабере у некој итерацији у методи TopTrail*, и премешта у скуп F. Као закључак, јасно је да ће услов* DetectionDone *бити задовољен, чиме може да се тврди да алгоритам терминира у сваком случају.*

6.5 Модификација алгоритма за инлајновање

Након што детекција често извршаваног кода из алгоритма 6.3 креира скуп стаза T који одговара често извршаваним компилационим јединицама, алгоритам за инлајновање, који утиче на формирање засебне компилационе јединице, може да искористи информације из скупа стаза T. Неки алгоритми за инлајновање одржавају *структуру стабла позива* [5, 120, 10], која се на неким местима зове *план инлајновања* (енг. *inlining plan*) или *стабло инлајновања*, док преостали алгоритми махом раде директно над графом позива [78, 77, 86, 121]. Одлуке око ширења компилационе јединице преко графа позива или структуре стабла позива се често доносе на основу анализе користи и цене (*cost-benefit analysis*) [10, 77, 122, 80, 117, 78, 123].

Иако се може тврдити да анализа односа користи и цене у оквиру већине инлајнера може бити унапређена употребом информација о фреквенцији извршавања у оквиру стаза трагова, у овој секцији биће показано како је конкретан алгоритам за инлајновање, који је коришћен у оквиру компајлера *Graal* [5], проширен ових информацијама. Овај алгоритам за инлајновање одржава структуру стабла инлајновања (дефинисано у једначини 6.11), које представља коначно стабло угњеждено у оквиру неког стабла позива C, као што је дефинисано у једначинама 2.4 и 2.7.

$$\mathbb{I}(P) \equiv \{(N, E) : |N| < \aleph_0 \wedge E \subseteq N \times N \wedge \exists C \in \mathbb{C}(P), (N, E) \lessdot C\} \tag{6.11}$$

Процес инлајновања одвија се по следећим корацима.

1. Стабло инлајновања се шири (експандује) док инлајнер не донесе одлуку да је довољна количина стабла позива покривена.

2. Инлајнер одлучује који делови стабла инлајновања ће бити *инлајновани* у неку компилациону јединицу.

3. Инлајнер оптимизује стабло инлајновања одсецањем неких грана стабла, односно, покушава да пронађе друго стабло инлајновања угњеждено у оквиру тренутног стабла.

Ови кораци се понављају док се не задовољи услов завршетка алгоритма и они су сумирани у оквиру алгоритма 6.4.

Током фазе ширења, инлајнер репетитивно бира чвор који је лист стабла инлајновања, и потом додаје потомке тог чвора. Алгоритам 6.5 приказује појединачни корак ширења стабла инлајновања – почевши од корена, спушта се до чвора потомка са највишим приоритетом φ све док не пронађе неки чвор $\eta_{s_1,\ldots,s_n}$ који нема потомке, тј. лист.

Алгоритам 6.4: GraalVM inlajnovanje [5]

Input : program call graph G, entry-point e
Output : IR u of the root compilation unit

$\theta = (\{\eta_e\}, \varnothing)$;
while $\neg$ InliningDone(G, θ) **do**
  θ = Expand(G, θ);
  θ = Inline(θ);
  θ = Optimize(θ);
end
$\eta_u = \inf_{E_\theta^*} N_\theta$ *gde je* $\theta = (N_\theta, E_\theta)$;
u = GetIR(η_u);

Алгоритам 6.5: Корак Expand [5]

Input : call graph (N, E), inlining tree θ
Output : expanded inlining tree θ

$\eta = \inf_{E^*} N$ *gde je* $\theta = (N_\theta, E_\theta)$;
while $\{\eta_c : \eta \to \eta_c \in E_\theta\} \neq \varnothing$ **do**
  $\eta = \arg\max_{\eta \to \eta_c \in E_\theta} \varphi(\eta_c)$
end
$\eta_{s_0,\dots,s_n} = \eta$;
$N_c = \{\eta_{s_0,\dots,s_n,s_{n+1}} : \eta_{s_n} \to \eta_{s_{n+1}} \in E\}$;
$E_c = \{\eta_{s_0,\dots,s_n} \to \eta_{s_0,\dots,s_n,s_{n+1}} : \eta_{s_n} \to \eta_{s_{n+1}} \in E\}$;
$\theta = \theta \cup (N_c, E_c)$;

Ако тај чвор - лист не садржи ниједну локацију позива (енг. *callsite*), тада се његов приоритет (φ) поставља на $-\infty$. У супротном, приоритет тог листа се дефинише као количник његовог бенефита (*benefit*) и величине кода (*codeSize*), где је бенефит повезан са фреквенцијом позивања. Приоритет φ неког унутрашњег чвора једнак је приоритету чвора потомка η_c са највећим приоритетом φ, умањеним за вредност функције пенала која зависи од величине одговарајућег подстабла. Како би се обезбедило да се ређе позиване методе које су близу корену компилационе јединице не занемаре, користе се пенали како би се смањила вероватноћа ширења великих подстабала која се често извршавају [5]:

$$\varphi_{\text{GraalVM}}(\eta, \theta, G) \equiv \begin{cases} -\infty & \eta_{s_n} \to \eta_{s_{n+1}} \notin E_G \\ benefit(\eta) \,/\, codeSize(\eta) & \eta_{s_1,\dots,s_n,s_{n+1}} \notin N_\theta \wedge \eta_{s_n} \to \eta_{s_{n+1}} \in E_G \\ \max\limits_{\eta \to \eta_c \in E_\theta} \varphi_{\text{GraalVM}}(\eta_c, \theta, G) - penalty(\eta) & \textit{у супротном} \end{cases}$$
$$\textit{где је} \quad \eta = \eta_{s_1,\dots,s_n} \quad \theta = (N_\theta, E_\theta) \quad G = (N_G, E_G) \tag{6.12}$$

Модификације које се описују у овој секцији односе се на процедуру InlineHot из алгоритма 6.2. Процедура InlineCold одговара неизмењеној верзији инлајнера у оквиру компајлера *Graal*.

Модификације корака експанзије. Оптимизација инлајновања има ограничен буџет за ширење стабла инлајновања, тако да се инлајновање ослања значајно на ширење најзначајнијих делова стабла инлајновања, тј. оних делова који доносе највећи бенефит. Због тога се наводи функција приоритета φ тако да се повећава вероватноћа за експандовање чворова стабла инлајновања који одговарају чворовима неке стазе трагова. Разлог лежи у томе што је вероватније да ови чворови транзитивно позивају често извршавани код, који треба да буде инлајнован у оквиру те компилационе јединице. Модификација (једначина 6.13) описана у овом раду дефинише нову функцију приоритета $\varphi_{\text{InlineHot}}$, која множи подразумевану вредност приоритета са константом k_{bonus} када је чвор стабла инлајновања могуће упарити са чвором неке стазе трагова. Вредност константе k_{bonus} је експериментално утврђена. Захваљујући увођењу пенала у функцију приоритета φ_{GraalVM}, бонус само наводи експанзију око често извршаваног кода, који је репрезентован стазама, али не спречава истраживање делова стабла позива који су означени као ретко извршавани.

$$\varphi_{\text{InlineHot}}(\eta_{s_1,\ldots,s_n},\theta,G) \equiv \varphi_{\text{GraalVM}}(\eta_{s_1,\ldots,s_n},\theta,G) \cdot \begin{cases} k_{bonus} & \exists(N_\tau,E_\tau,\gamma_\tau)\in T, \eta_{s_m,\ldots,s_n}\in N_\tau \\ 1 & \textit{у супротном} \end{cases} \tag{6.13}$$

Модификације буџета. Када се преводи нека јединица компилације која покрива често извршавани код, предложени алгоритам мења параметре инлајнера како би повећао буџет који је доступан за инлајновање и тиме омогућио инлајновање веће количине кода него што је то иначе омогућено. Инлајнер користи следећу функцију граничне вредности, која може да заустави ширење у зависности од величине ν целокупног стабла инлајновања, као што је дефинисано у једначини 6.8:

$$\textit{benefit}(\eta)\,/\,\textit{codeSize}(\eta) \geq e^{\nu(\theta)/r} \tag{6.14}$$

Затим, је потребно ограничити количину експандованог кода који ће заправо бити инлајнован. У те сврхе, инлајнер користи следећу функцију граничне вредности, на основу које се одлучује да ли код процедуре η која се позива из корена јединице компилације стабла θ треба директно уградити у ту јединицу компилације:

$$\textit{benefit}(\eta)\,/\,\textit{codeSize}(\eta) \geq t_1 \cdot 2^{(\textit{codeSize}(\textit{root}(\theta))+\textit{codeSize}(\eta))/t_2} \tag{6.15}$$

Како би се повећала величина често извршаваних јединица компилације, и као последица, побољшале перформансе целокупног програма, потребно је пронаћи најбоље вредности кључних параметара инлајнера. У оквиру истраживања спроведени су експерименти проналажења најбољих вредности параметара r, t_1 и t_2, а резултати су приказани у секцији 8.3.

Модификације оптимизације инлајновања ређе извршаваних делова кода. Иако алгоритам у највећој мери задржава идентичан поступак инлајновања ретко извршаваних делова кода, који нису кључни за перформансе програма, ипак постоји једна

модификација. Алгоритам спречава инлајновање било које методе која се често извршава и налазе се у скупу H из алгоритма 6.3, и претходно је преведена као таква, у неку јединицу компилације која се ретко извршава. Ово се постиже на нивоу оба фазе инлајновања. У фази ширења стабла инлајновања приоритет позива процедура $s_n \in H$ поставља се на $-\infty$ (описано у једначини 6.16). Додатно, INLINE корак из алгоритма 6.4 модификован је тако да никада не инлајнује позив ка некој методи $s_n \in H$.

$$\varphi_{\text{INLINECOLD}} \equiv \begin{cases} -\infty & s_n \in H \\ \varphi_{\text{GRAALVM}}(\eta_{s_1,\dots,s_n}, \theta, G) & \textit{иначе} \end{cases} \tag{6.16}$$

6.6 Класификација преосталих често извршаваних позива

Након спровођења оптимизације инлајновања током превођења неке често извршаване процедуре, у општем случају преостаће неки неинлајновани позиви у датој јединици компилације. Неки од ових позива нису инлајновани јер, на основу хеуристике, нису означени као значајни, док за инлајновање неких других, значајних позива, није било довољно буџета. Такви преостали значајни позиви треба даље да буду преведени са повећаним буџетом.

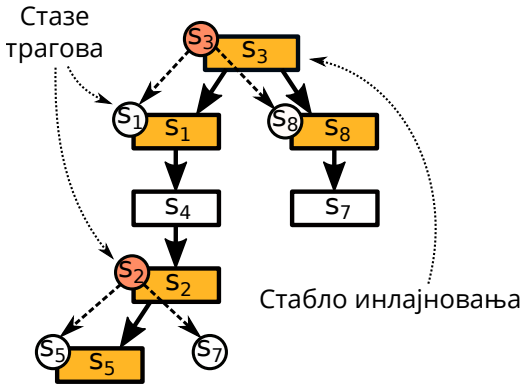
Метода ISHOT из алгоритма 6.2 одлучује за преостале позиве компилационе јединице, на који начин треба да буду компајлиране циљне методе тих позива. Другим речима, свака позивана метода треба да буде распоређена и један од два реда – H ред или C ред. Позивана метода чији чвор $\eta_{s_1,\dots,s_m,\dots,s_n}$ у стаблу инлајновања одговара некој стази трагова $\eta_{s_m,\dots,s_n}$ смешта се у H ред за компилацију.

У датом примеру са слике 6.6 приказани су стабло инлајновања, чији су чворови представљени правоугаоницима и стазе, чији су чворови представљени круговима. Упарени чворови стазе $s_1 \leftarrow s_3 \rightarrow s_8$ са стаблом инлајновања обојени су жутом бојом, док су преостали, неупарени чворови обојени белом бојом. Чвор s_4 није упарен ни са једним стазом, али његова позвана метода s_2 може бити упарена са другом стазом $s_5 \leftarrow s_2 \rightarrow s_7$. Такве позиве класификујемо као значајне, нпр. ако метода s_5 не буде инлајнована, биће надаље компајлирана са повећаним буџетом.

Најпре ће бити дефинисана *операција упаривања стаза* (енг. *trail-matching operation*) $\downarrow$ на следећи начин. На основу датог скупа стаза T и чвора стабла инлајновања $\eta_{s_1,\dots,s_m,\dots,s_n}$, операција проналази ону стазу која садржи најдужи суфикс секвенце позива $s_1, \dots, s_m, \dots, s_n$.

$$T \downarrow \eta_{s_1,\dots,s_m,\dots,s_n} \equiv \underset{(N_\tau,E_\tau,\gamma_\tau)\in T, \exists \eta_{s_m,\dots,s_n}\in N_\tau}{\arg\max} n - m \qquad \textit{тако да} \quad \{(N_\tau, E_\tau, \gamma_\tau) \in T : \exists \eta_{s_m,\dots,s_n} \in N_\tau\} \neq \varnothing \tag{6.17}$$

У најбољем случају, резултат операције упаривања стаза је иста стаза τ, којом почиње тренутна значајна компилациона јединица која одговара методи s_1, тј. $\inf_{E_\tau^*} N_\tau = \eta_{s_1}$.



Слика 6.6: Упаривање стабла инлајновања са стазом трагова

У супротном, упаривање стабла инлајновања и оригиналне стазе τ је прекинуто, што показује да су неки од позива у секвенци $s_1, \ldots, s_m, \ldots, s_n$ мање значајни, а неки чворови стабла инлајновања, који се у стаблу налазе негде након места прекида упаривања, упарени су са стазом који почиње чвором η_{s_m}. Ово значи да је позив $s_{m-1} \to s_m$ упућен из мање значајне методу у методу која се често извршава. У крајњем стаблу инлајновања, позив се сматра значајним ако и само ако постоји упарена стаза:

$$\textsc{IsHot}(\eta) \equiv \exists \tau \in T, \tau = T \downarrow \eta \tag{6.18}$$

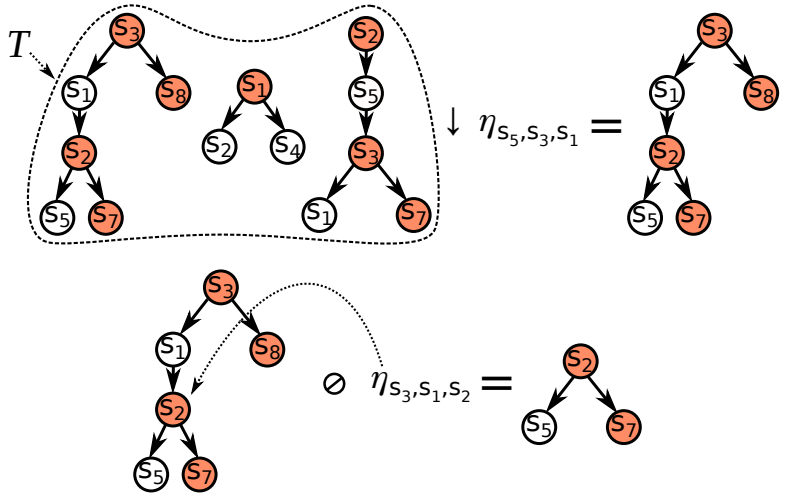
Како би се обезбедило да метода InlineHot из секције 6.5 ради, додатно је потребно повезати стазу са сваком значајном јединицом компилације. За почетни скуп значајних метода H из алгоритма 6.3, ово повезивање је праволинијско пошто свака метода $s \in H$ одговара стази $\tau \in T$. Ипак, значајан позив методе s можда нема директно одговарајућу стазу и због тога, уводи се *операција пресецања стазе* (енг. *trail-cut operation*).

Операција пресецања стазе. На основу стазе $\tau = (N_\tau, E_\tau, \gamma_\tau)$ и њеног трага $\eta_{s_1,\ldots,s_m} \in N_\tau$, операција пресецања стазе $\oslash$ формира нову стазу која се састоји искључиво од стабла које почиње са $\eta_{s_1,\ldots,s_m}$:

$$\begin{aligned}(N_\tau, E_\tau, \gamma_\tau) \oslash \eta_{s_1,\ldots,s_m} \equiv (N, E, \gamma) \quad \textit{где је} \quad \eta_{s_1,\ldots,s_m} \in N_\tau \\ N = \{\eta_{s_m,\ldots,s_n} : \eta_{s_1,\ldots,s_m,\ldots,s_n} \in N_\tau\} \quad \gamma = \{\eta_{s_m,\ldots,s_n} : \eta_{s_1,\ldots,s_m,\ldots,s_n} \in \gamma_\tau\} \\ E = \{\eta_{s_m,\ldots,s_n} \to \eta_{s_m,\ldots,s_n,s_{n+1}} : \eta_{s_1,\ldots,s_m,\ldots,s_n} \to \eta_{s_1,\ldots,s_m,\ldots,s_n,s_{n+1}} \in E_\tau\}\end{aligned} \tag{6.19}$$

Операције упаривања стазе и пресецања стазе су илустроване у примеру са слике 6.7. Најдужа упарена стаза у скупу T за контекст позивања $\langle s_5, s_3, s_1\rangle$ је крања лева стаза са слике зато што ова стаза садржи најдужи суфикс s_5, s_3, s_1. Треба приметити да резултат операције $T \downarrow \eta_{s_5,s_3,s_1}$ није најдеснија стаза из скупа T зато што $\langle s_2, s_5, s_3, s_1\rangle$ није суфикс од $\langle s_5, s_3, s_1\rangle$ као што се захтева према једначини 6.17. Резултујућа стаза је затим пресечен на десној страни слике, на чвору η_{s_3,s_1,s_2}, што ствара стазу $s_5 \leftarrow s_2 \to s_7$.

Стаза трагова позиване процедуре. Стаза који одговара значајном позиву $\eta_{s_1,\ldots,s_n}$ одређена је комбинацијом операција упаривања стаза и пресецања стаза. На почетку,



Слика 6.7: Операције упаривања и пресецања стаза

проналази се упарена стаза за $\eta_{s_1,\ldots,s_n}$ у скупу стаза T, а затим се сече стаза на подстаблу које одговара секвенци позива $s_1, \ldots, s_n$:

$$\mathit{calleeTrail}(T, \eta_{s_1,\ldots,s_m,\ldots,s_n}) \equiv (T \downarrow \eta_{s_1,\ldots,s_m,\ldots,s_n}) \oslash \eta_{s_m,\ldots,s_n} \quad \textit{где је} \quad \eta_{s_m,\ldots,s_n} \in (T \downarrow \eta_{s_1,\ldots,s_m,\ldots s_n}) \tag{6.20}$$

Значајна јединица компилације чија корена метода одговара неком позиву методе η из друге значајне компилационе јединице, увек је асоцирана са стазом одређеним изразом $\mathit{calleeTrail}(T, \eta)$.

У табели 6.2 приказани су најзначајнији симболи и функције из секција 6.5 и 6.6.

Табела 6.2: Преглед главних симбола и функција из секција 6.5 и 6.6

Симбол	**Назив**	**Значење**
$\mathbb{I}(P)$	Стабло инлајновања (Једн. 6.11)	Структура података коју одржава инлајнер на основу програма P.
φ_{GraalVM}	GraalVM функција приоритета чвора (Једн. 6.12)	Функција која рачуна приоритет чворова коришћењем хеуристике у GraalVM [5].
$\varphi_{\text{InlineHot}}$	Функција приоритета чвора InlineHot (Једн. 6.13)	Функција која увећава приоритет чвора уколико је он маркиран као значајан.
$\varphi_{\text{InlineCold}}$	Функција приоритета чвора InlineCold (Једн. 6.16)	Функција која модификује приоритет чвора стабла инлајновања, уколико је он означен као хладан.
$T \downarrow \eta_{s_1,\ldots,s_m,\ldots,s_n}$	Операција упаривања стазе (Једн. 6.17)	Операција која проналази стазу у скупу стаза T који садржи најдужи суфикс секвенце позива $\eta_{s_1,\ldots,s_m,\ldots,s_n}$.
$\tau \oslash \eta_{s_1,\ldots,s_m}$	Операција пресецања стазе (Једн. 6.19)	Операција која формира нову стазу која се састоји само од подстабла стаза τ почевши од чвора $\eta_{s_1,\ldots,s_m}$.

6.7 Преглед параметара

У табели 6.3, излистани су главни параметри ψ, k_{bonus} и k_{rec}, уведени у предложеном алгоритму. Константа ψ представља граничну вредност коју број извршавања из неког улаза у профилима мора да премаши како би био укључен у иницијални скуп често извршаваних контекста програма, посматрано у односу на укупно време проведено у програму, према дефиницији из једначине 6.1. Константа k_{rec} служи као умножак у једначини 6.9 за израчунавање пенала за рекурзију у стази. k_{bonus} је константа којом се множи подразумевана вредност приоритета чвора у стаблу инлајновања за сваки чвор који може бити упарен са неком стазом, на основу једначине 6.13. За све константе су експерименталним путем тражене набоље вредности, тј. вредности које када се користе у алгоритму, доводе до набољих перфомансi програма. Резултати ових експеримената презентовани су у секцији 8.7. Треба имати у виду да различити преводиоци и алгоритми инлајновања користе другачије међурепрезентације и шеме оптимизација и да хеуристике анализе користи и цене које користе могу изгледати другачије. Самим тим је очекивано да ће бити потребно да се проналазе најбоље вредности параметара у сваком специфичном окружењу.

Табела 6.3: Преглед главних параметара уведених у алгоритму *PRINC*

Симбол	**Назив**	**Значење**
ψ	Праг броја извршавања (Јед. 6.1)	Гранична вредност изнад које се број извршавања сматра значајним.
k_{rec}	Константа рекурзије (Јед. 6.9)	Константа која се користи за граничавање количине рекурзије у стази.
k_{bonus}	Бонус ширења (Јед. 6.13)	Бонус који се користи за ширење чвора стабла инлајновања упареног са стазом.

Секција такође садржи кључне параметре који су већ укључени у инлајнер и они су приказани у табели 6.4. Ови параметри су подешавани како би се показало да имплементирани алгоритам не показује боље перформансе само над одабраним подскупом вредности параметара. Резултати ових експеримената су приказани у секцији 8.3. Иако је примарни циљ експеримента одређивање најбољих вредности параметара, које се затим постављају као подразумеване, демонстрирано је и да нови алгоритам ради добро и са осталим вредностима параметара.

Табела 6.4: Преглед главних параметара у постојећем инлајнеру

Симбол	**Назив**	**Значење**
r	Expansion inertia base (Јед. 6.14)	Параметар који диктира количину ширења стабла позива, које спроводи инлајнер.
t_1	Relative benefit coefficient (Јед. 6.15)	Параметар који усмерава граничну вредност користи на основу које се доноси одлука да ли неку методу треба инајновати.
t_2	Base target spending (Јед. 6.15)	Параметар који ограничава буџет доступан за оптимизацију инлајновања.

7 Детаљи имплементације

Алгоритам за измену распореда превођења и унапређење оптимизације инлајновања *PRINC* приказан је на апстрактном нивоу у поглављу 6. Уз прилагођавање конкретном окружењу, такав алгоритам може бити имплементиран унутар већине модерних преводилаца. У оквиру овог истраживања, предложени алгоритам је имплементиран као саставни део AOT компајлера за *GraalVM Native Image* [14] у циљу евалуације над конкретним улазним програмима. Прецизније говорећи, додата је фаза за анализу улазних профила, а потом је резултат примене ове фазе употребљен како би се побољшале одлуке постојеће оптимизације инлајновања и изменила постојећа имплементација реда за компилацију у оквиру конкретног AOT компајлера.

Ово поглавље садржи описе структура података дефинисаних у поглављу 5 из перспективе имплементације у секцији 7.1, као и неке специфичне детаље имплементације у секцији 7.2. Приказани имплементациони детаљи раздвојени су од апстракције алгоритма из разлога што могу варирати у односу на конкретан компајлер коме се описани алгоритам прилагођава. Превођење и извршавање програма употребом предложеног алгоритма биће приказани у секцијама 7.3 и 7.4 на примеру бенчмарка Mnemonics.

Улаз имплементације. Улаз имплементираног решења је затворени универзум метода и типова, који је настао као резултат Points-to анализе у оквиру поступка формирања извршне датотеке Native Image. Главни кораци конструкције извршног фајла, укључујући и анализу представљени су у секцији 3.2. Универзум је представљен класом `HostedUniverse`. Такође, улаз чине и скуп улазних метода програма, од којих је свака представљена класом `HostedMethod`, као и профили, тј. скуп парова локација у програмском коду и целобројних вредности. Овакав улаз директно одговара улазу алгоритма 6.2, дефинисаном у поглављу 6.2. Универзум садржи граф позива, који одговара дефиницији из секције 2.1, као и скуп достижних класа и њихових поља.

7.1 Структуре података за анализу кода

У овој секцији приказана је имплементација главних структура података уведених у овом раду, на основу којих је извршена детекција често извршаваних делова кода. Како би се објаснила имплементација написана у програмског језику Java, у наставку ће бити дефинисана најважнија поља сваке од ових структура података, а биће и приказане и њихове кључне методе.

7.1.1 Структура података `Breadcrumb`

У Листингу 7.1 је приказана имплементација структуре података траг (`Breadcrumb`), тј. чвора који припада анотираној структури података стаза трагова, према дефиницији 5.2. Чвор одговара једној методи програма. Информација о методи на коју се чвор односи смештена је у пољу `method`. Класа садржи и референцу на родитељски чвор у пољу `parent`, који се односи на методу позиваоца у текућем стаблу, тј. стази трагова. Поље `callsites` представља мапу која повезује позиције у коду, тј. индексе бајткода са листом чворова који одговарају методама које могу бити позване на тим локацијама. Вредности у мапи су листе метода, уместо појединачних метода, како би се сместиле све могуће имплементације виртуелних метода које могу бити позване на датим локацијама. Сваки траг садржи `GraftPoint` објекат, који означава тачку спајања уколико се тај чвор може користити у операцији спајања, као што је описано у секцији 5.1. У супротном, ово поље има вредност `NULL`. `GraftPoint` објекат садржи поље `attenuation`, које се односи на функцију удела тачке спајања (*graft-point attenuation*) одговарајуће стазе трагова. Функција је формално дефинисана у секцији 6.4.1.

```
class Breadcrumb {
    HostedMethod method;
    Map<Integer, Breadcrumb[]> callsites;
    Breadcrumb parent;
    GraftPoint graftPoint;
}
class GraftPoint {
    double attenuation;
}
```

Листинг 7.1: Структуре података које описују траг

7.1.2 Структура података `Trail`

У листингу 7.2 приказана је конкретна имплементација структуре података стаза трагова (`Trail`). Формална дефиниција ове структуре података, за анотиране стазе трагова дата је у једначини 5.2 у секцији 5.1.

Стаза трагова је стабло чији чворови представљају трагове. Корени траг, садржан у пољу `root` класе, одговара методи јединице компилације на коју се стаза односи. Сви други чворови стабла имају тачно један родитељски чвор и произвољан број потомака, као што је приказано у листингу 7.1. Стаза трагова садржи мапу `grafts`, која пресликава методе у скуп чворова-трагова у стаблу који одговарају датој методи и садрже `GraftPoint` објекте. Другим речима, то су сви они чворови на којима се може спровести операција качења друге стазе трагова чији корени траг одговарају истој методи. Ова мапа се користи како би се оптимизовала операција спајања стазе трагова, која је прецизно дефинисана једначином 5.4. Додатно, како би се моделовале бројне функције стазе трагова, дефинисане у секцији 6, сваком објекту стазе додељен је и објекат класе `Metrics`. Ова класа садржи следећа поља: `hotness`, које представља функцију фреквенције стазе $\chi(\tau)$ из једначине 6.5; `size`, које се односи на функцију величине

```
class Trail {
    Breadcrumb root;
    Map<HostedMethod, Breadcrumb[]> grafts;
    Metrics metrics;

    static Trail createInitial(Profile profile);
    Trail expand(Profile profile);
    Trail graft(Trail graftee);
}
class Metrics {
    long hotness;
    int size;
    Map<HostedMethod, Integer> recursion;
}
class Profile {
    Location[] context;
    long value;
}
class Location {
    HostedMethod method;
    int bci;
}
```

Листинг 7.2: Структуре података које описују стазу трагова

стазе $\nu(\tau)$ из једначине 6.8; `recursion`, које мапира сваку методу са њеном највећом дубином рекурзије у датој стази, и која се користи како би се ограничило инлајновање рекурзивних позива на основу пенала $\zeta(\tau)$ дефинисаног у једначини 6.9.

Листинг 7.2 такође садржи методе које имплементирају три главне операције дефинисане за стазе трагова. Ове операције су формално дефинисане у секцији 5.1. Метода `createInitial` одговара операцији продужавања празне стазе неким контекстом позива $(\varnothing, \varnothing, \varnothing) \odot \langle \ell_1, \ldots, \ell_n \rangle$, на основу дефиниције 5.3. Ова метода као улаз добија објекат `Profile`, који се састоји од контекста позива и одговарајуће нумеричке вредности `value`. Њен резултат је нови објекат стазе трагова, код кога сваки траг одговара тачно једној локацији из улазног контекста позива и налазе се у одговарајућој релацији предак-потомак. Нумеричка вредност из улазног профила користи се ради постављања иницијалне вредности фреквенције извршавања те стазе (`hotness` поље). Величина стазе и дубина рекурзије се рачунају на основу улазног контекста. Корен резултујуће стазе трагова постаје нова тачка спајања, на којој је омогућено качење друге стазе и чији се фактор умањења (`attenuation`) се поставља на 1.0.

Метода `expand` спроводи операцију продужавања стазе трагова над `this` објектом и имплементира операцију $\odot$ из једначине 5.3. Улазни аргумент `profile` методе садржи контекст којим се врши експанзија и нумеричку вредност из профила која одговара датом контексту. За сваку локацију контекста, у методи се креира по један чвор-траг, а потом се ажурира мапа `grafts`, слично поступку приликом конструкције нове стазе трагова.

Метода `graft` имплементира операцију спајања стазе трагова $\circledast$ из једначине 5.4. Метода добија стабло `graftee`, које припаја стаблу `this`. Ова операција може бити

спроведена само над оним чворовима стабла `this` који су анотирани као тачке спајања и чије методе одговарају кореној методи стазе трагова `graftee`. Мапа `graftee` омогућава ефикасну селекцију датог скупа чворова. Како је могуће да постоји више таквих чворова у истом стаблу, операција припајања једне стазе може се применити на више места у оквиру стазе трагова `this`. За сваки чвор над којим се примењује операција, стаза трагова `graftee` се копира и качи на одговарајућу локацију. Метрике резултујућег стабла се ажурирају на основу метрика припојеног стабла. Укупна учестаност извршавања резултујућег стабла формира се сумирањем фреквенције извршавања стабла `graftee` помножене фактором умањења `attenuation` чвора стабла над којим се операција извршила и фреквенције стабла `this` пре примене операције. Израчунавање фреквенције извршавања резултујуће стазе трагова приказано је у једначини 6.5. Преостале две метрике `size` и `recursion` ажурирају се увећањем величине резултујуће стазе трагова величином стазе која је прикачена, као и увећањем дубине рекурзије сваке методе чворова додате стазе, тако да су задовољене једначине 6.8 и 6.9.

7.1.3 Структура података `TrailSet`

Током извршавања алгоритма 6.3 метода DetectionDone позива се на линији 4, а метода TopTrail на линији 5. Текућа стаза трагова τ додаје се у скуп финализованих стаза F на линији 8, а уклања из скупа T на линији 10. Операција спајања шума стаза на линији 11 се извршава вишеструко. Употребом уобичајене имплементације скупа, свака од ових операција би имала сложеност $O(n)$, где је n укупан број стабала у скупу. Како би се умањио трошак израчунавања, имплементација користи наменску структуру за скуп стаза (`TrailSet`), која је приказана листингом 7.3.

```
class TrailSet {
    PriorityQueue<Trail> queue;
    HashSet<Trail> finalized;
    HashMap<HostedMethod, Set<Trail>> candidates;

    boolean isFinalized();
    Trail popHottest();
    int unionGraft(Trail graftee);
    void graftToRoots(Trail trail);
}
```

Листинг 7.3: Структура података која описује шуму стаза трагова

Класа садржи приоритетни ред стабала `queue`, хеш скуп финализованих стабала `finalized` и хеш мапу `candidates`, која мапира сваку методу на скуп стабала која садрже бар један анотирани чвор над којим може бити извршена операција спајања стабла, а који одговара датој методи. Стабла која нису финализована смештена су у реду `queue`, а финализована у хеш скупу `finalized`, па је према нотацији у листингу 6.3, $T \equiv$ `queue` $\cup$ `finalized` и $F \equiv$ `finalized`.

Структура података шума стабала укључује неколико метода. Метода `isFinalized` проверава да ли је ред `queue` празан, чиме се метода DetectionDone из једначине 6.2

извршава у $O(1)$ сложености. Метода `popHottest` користи `queue` да пронађе најчешће извршавану стазу трагова у сложености $O(\log n)$, што се даље користи како би се имплементирала полиса TopTrail. Поредак стаза је утврђен метриком `hotness` стабала према једначини 6.5.

Мапа `candidates` омогућава ефикасно проналажење свих кандидата-чворова за операцију спајања у свим стаблима која припадају скупу. Временска сложеност ове операције је константна, $O(1)$, и зависи од величине резултујућег скупа стабала. Време потребно да се операција спајања шума $\overset{\circledast}{\cup}$ изврши, према једначини 5.10 одређено је искључиво величином резултујућих стабала. Хеш скуп `finalized` се користи како би се дохватио коначни скуп стаза трагова (скуп H из алгоритма 6.3).

7.2 Детаљи имплементације продужавања контекста

Претходне секције описују како имплементација алгоритма у конкретном окружењу *Native Image* одговара формализацији описаној у поглављу 6.3. Ипак, изостављени су неки детаљи имплементације зависни од самог окружења, попут одређивања контекста позивања, израчунавања фреквенције извршавања контекста позивања и одређивања иницијалног скупа контекста (полиса CallingContexts). У овој секцији биће приказани овакви детаљи, прецизније како се израчунавају фреквенције извршавања локација директних и виртуелних позива метода, и како се процењује фреквенција извршавања појединачних чворова-трагова у стаблу. Такође, биће приказани детаљи имплементације одређивања контекста позива и фактора умањења за сваку од метода програма.

7.2.1 Одређивање контекста позивања

Како би се одредио скуп директних позивалаца неке стазе трагова, тј. њене корене методе, $\mathit{directCallers}(\tau)$, имплементација описана у овом раду претпроцесира међурепрезентацију метода у универзуму `HostedUniverse`. Резултат је мапа која пресликава сваку методу у скуп локација са којих она може бити позвана, било да је реч о директним или индиректним позивима. Исто мапирање се користи и како би се имплементирала CallingContexts полиса из секције 6.4.1.

7.2.2 Одређивање фреквенције позива виртуелних метода

За сваки контекст позива који се односи на позив виртуелне методе, *Native Image PGO* подаци мапирају сваку конкретну методу на број позива те методе:

$$\pi(\ell_1 \dots \ell_n) = \{(s, N) : s \in \mathit{implementations}(\mathit{target}(\ell_n))\} \tag{7.1}$$

У једначини 7.1 $\mathit{target}(\ell_n)$ је метода основне класе, која одговара локацији ℓ_n позива виртуелне методе, а *implementations* је функција која одређује скуп метода које представљају конкретне имплементације одговарајуће методе базне класе. Зависно од случаја

коришћења, овакав профил има две интерпретације. При креирању скупа почетних стаза трагова, фреквенција извршавања трага који се креира на основу профила виртуелних позива формира се сумирањем броја позива сваке конкретне методе која одговара тој локацији позива. Разлог за овакву одлуку лежи у томе да свака имплементација базне методе утиче на укупан број извршавања позива, а самим тим и стазе трагова које он представља.

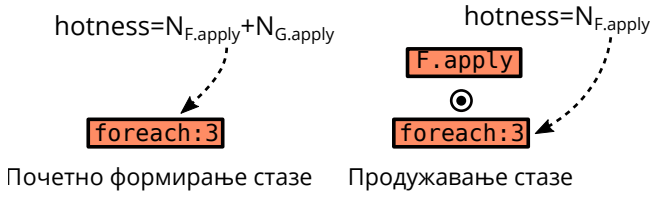
$$h_{\ell_1,\ldots,\ell_n} = \sum_{(s,N)\in\pi(\ell_1\ldots\ell_n)} N \tag{7.2}$$

Приликом продужавања стазе трагова, чији корени траг одговара методи s, употребом контекста који се односи на позив виртуелне методе, од свих могућих имплементација, користи се број позива конкретне методе s:

$$h_{\ell_1,\ldots,\ell_n|s} = N \quad \textit{тако да} \quad (s,N) \in \pi(\ell_1,\ldots,\ell_n) \tag{7.3}$$

Дакле, приликом креирања почетног скупа стаза на линији 1 алгоритма 6.3, једначина 7.2 одређује фреквенцију извршавања локације виртуелног позива. Приликом продужавања најприоритетније стазе трагова на линији 7 у алгоритму 6.3, једначина 7.3 одређује фреквенцију извршавања локације виртуелног позива.

Пример. За илустрацију претходних једначина биће искоришћен пример из листинга 2.1, (који је у раду коришћен за илустровање већине кључних целина и појмова), специфично контекст позива `foreach:3` са слике 7.1. Овај контекст се односи на позив виртуелне методе `apply`. Метода `apply` је основна метода, а одговарају јој конкретне имплементације `F.apply` и `G.apply` из примера. У конкретном примеру, овај контекст се користи за формирање иницијалне стазе трагова, чијој фреквенцији извршавања доприносе обе метрике (фреквенције извршавања метода `F.apply` и `G.apply`). Са друге стране, ако се стаза трагова која се састоји искључиво од трага `F.apply` продужи контекстом позива `foreach:3`, тада искључиво број извршавања конкретне методе `F.apply` доприноси фреквенцији извршавања стазе имајући у виду да се метода `G.apply` не може позвати из дате стазе.



Слика 7.1: Одређивање фреквенције позива виртуелних метода

7.2.3 Одређивање фреквенције директних позива метода

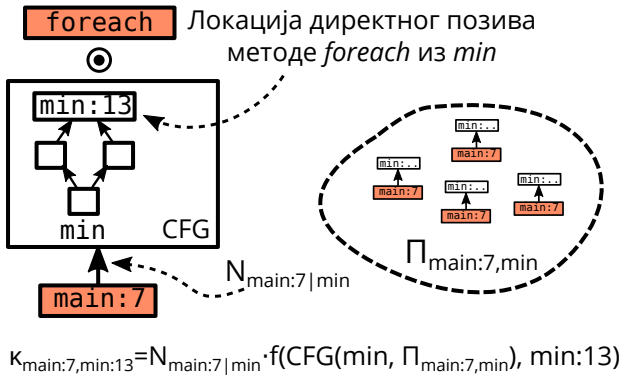
За разлику од виртуелних позива, профили који проистичу из *Native Image* инструментације не садрже фреквенције директних позива метода. Фреквенција извршавања

директног позива на локацији ℓ_n, која припада контексту позивања $\ell_1, \ldots, \ell_{n-1}, \ell_n$ мора бити израчуната на основу вероватноћа извршавања грана контролних структура који припадају методи s_n, која одговара последњој локацији ℓ_n контекста позивања. Да би се израчунала ова фреквенција, најпре је потребно одредити скуп $\Pi_{\ell_1,\ldots,\ell_{n-1},s_n|B}$ свих појединачних профила $\ell_1, \ldots, \ell_{n-1}, \ell'_n$, који се завршавају локацијом ℓ'_n унутар методе s_n таквих да се локација ℓ'_n односи на инструкцију кондиционала. Затим се, коришћењем `ControlFlowGraph` класе преводиоца *Graal* [124] формира граф тока контроле за методу s_n, у којем се релативне фреквенције (у односу на методу s_n) извршавања базичних блокова израчунавају на основу учестаности извршавања грана кондиционала из профила $\Pi_{s_n|B}$. Релативна фреквенција базичног блока који садрже директан позив ℓ_n се потом множи са бројем извршавања методе s_n како би се добио број позива из задатог контекста позивања:

$$h_{\ell_1,\ldots,\ell_{n-1},\ell_n|s} = N_{\ell_1,\ldots,\ell_{n-1}|s_n} \cdot f(CFG(s_n, \Pi_{\ell_1,\ldots,\ell_{n-1},s_n|B}), \ell_n) \tag{7.4}$$

У једначини 7.4, CFG враћа граф тока контроле који одговара методи s_n на основу профила контролних структура $\Pi_{s_n|B}$, f одређује фреквенцију базичног блока који садржи локацију ℓ_n у датом графу, релативну у односу на s_n методу, а $N_{\ell_1,\ldots,\ell_{n-1}|s_n}$ представља број извршавања методе s_n када се она позива из контекста $\ell_1, \ldots, \ell_{n-1}$. Овако израчуната фреквенција позива користи се у оба случаја коришћења – при креирању иницијалног скупа стаза трагова и приликом продужавања стаза.

Пример. За илустрацију једначине 7.4, биће коришћен пример из листинга 2.1. У фокусу је контекст позивања `main:7,min:13`, који се односи на директан позив методе `foreach`. Како би се израчунала фреквенција директног позива методе `foreach` на локацији `main:7,min:13`, потребно је одредити скуп профила $\Pi_{\texttt{main:7,min}|B}$, који се односи на гранање и одговара контекстима облика `main:7,min:X`, где се `min:X` односи на било коју локацију у методи `min`. Профили који се односе на контролне структуре у скупу $\Pi_{\texttt{main:7,min}|B}$ се потом користе како би се формирао граф тока контроле методе `min`, и мапирање између базичних блокова и њихових релативних фреквенција извршавања у односу на методу `min`. Ово мапирање се користи како би се израчунала релативна фреквенција $f(CFG(\texttt{min}, \Pi_{\texttt{main:7,min}|B}), \texttt{min:13})$ базичног блока који садржи



Слика 7.2: Одређивање фреквенције директних позива метода

локацију `min:13`, на којој се налази директан позив, а потом помножила вредношћу $N_{\texttt{main:7,min|min}}$, која представља број извршавања методе `min` када се она позива из контекста `main:7`. Резултат је укупан број директних позива позива методе са локације `main:7,min:13`.

7.2.4 Процена фактора умањења контекста позивања

Као што је претходно објашњено, у имплементацији приказаној у овом раду профили не садрже информације о директним позивима. Тако, израчунавање фактора умањења, дефинисано у једначини 6.4 мора бити прилагођено да укључи информације о директним позивима у имениоцу:

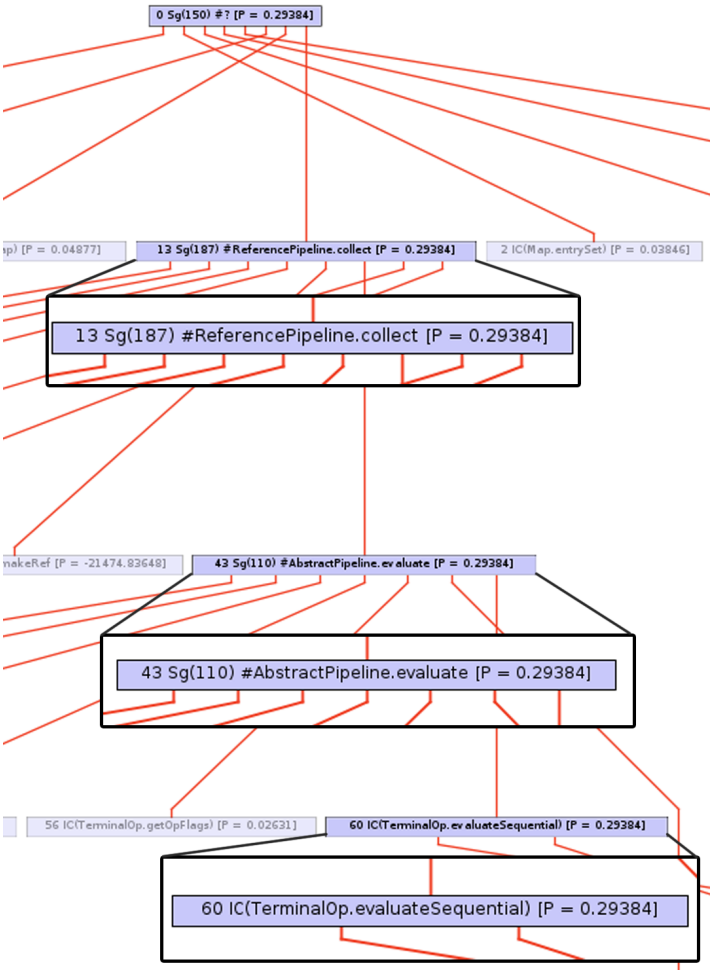
$$a_{\ell_1,\ldots,\ell_n}(\tau) \equiv \frac{h_{\ell_1,\ldots,\ell_n|\tau}}{\sum_{L\in directCallers(\tau)} h_{L|\tau} + \sum_{(L,h_{L|\tau})\in callerProfiles(\tau)} h_{L|\tau}} \tag{7.5}$$

У једначини 7.5, именилац садржи суму фреквенција извршавања $h_{L|\tau}$ контекста L који директно позивају стазу трагова τ (према једначини 7.4) и фреквенција $h_{L|\tau}$ контекста L који индиректно позивају стазу трагова τ (према једначини 7.3). Фреквенције извршавања директних позива метода и фактори умањења су кеширани у циљу побољшања перформанси.

7.3 Пример компилационе јединице

Рад предложеног алгоритма приказан је на једноставном показном примеру у поглављу 6. У овој секцији биће приказан фрагмент извршавања на репрезентативном програму. Изабран је бенчмарк Mnemonics из скупа бенчмарка Renaissance [44]. За дати бенчмарк, алгоритам идентификује 12 често извршаваних метода за компилацију, од којих је једна `org.renaissance.jdk.streams.MnemonicsCoderWithStream.encode`. У наставку ће бити приказан процес превођења методе `encode` уз посебан фокус на упаривање структуре стазе трагова са датим стаблом инлајновања.

На слици 7.3 налази се део стабла инлајновања методе `encode`, формираног у оквиру алгоритма за инлајновање у компајлеру *Graal* и визуелно приказаног употребом алата *Ideal Graph Visualizer (IGV)* [125]. Неколико чворова стабла је увећано и истакнуто ради побољшања читљивости њихових садржаја. Садржај сваког чвора састоји се од јединственог идентификатора чвора, низа карактера који означава тип чвора (нпр. `Sg`), величине методе коју дати чвор стабла инлајновања представља (изражено у броју чворова графовске репрезентације те методе), имена методе и приоритета за ширење датог чвора стабла инлајновања (у угластим заградама), респективно. Чвор у корену одговара методи `encode`, а његови директни потомци одговарају скупу метода које се позивају из методе `encode`. Једна од таквих метода које се позивају је и `ReferencePipeline.collect` метода, која даље слично има скуп позваних метода (са слике нпр. `AbstractPipeline.evaluate`). Надаље, један потомак чвора који одговара методи `evaluate` је чвор за методу `TerminalOp.evaluateSequential`. Ови чворови представљају најчешће

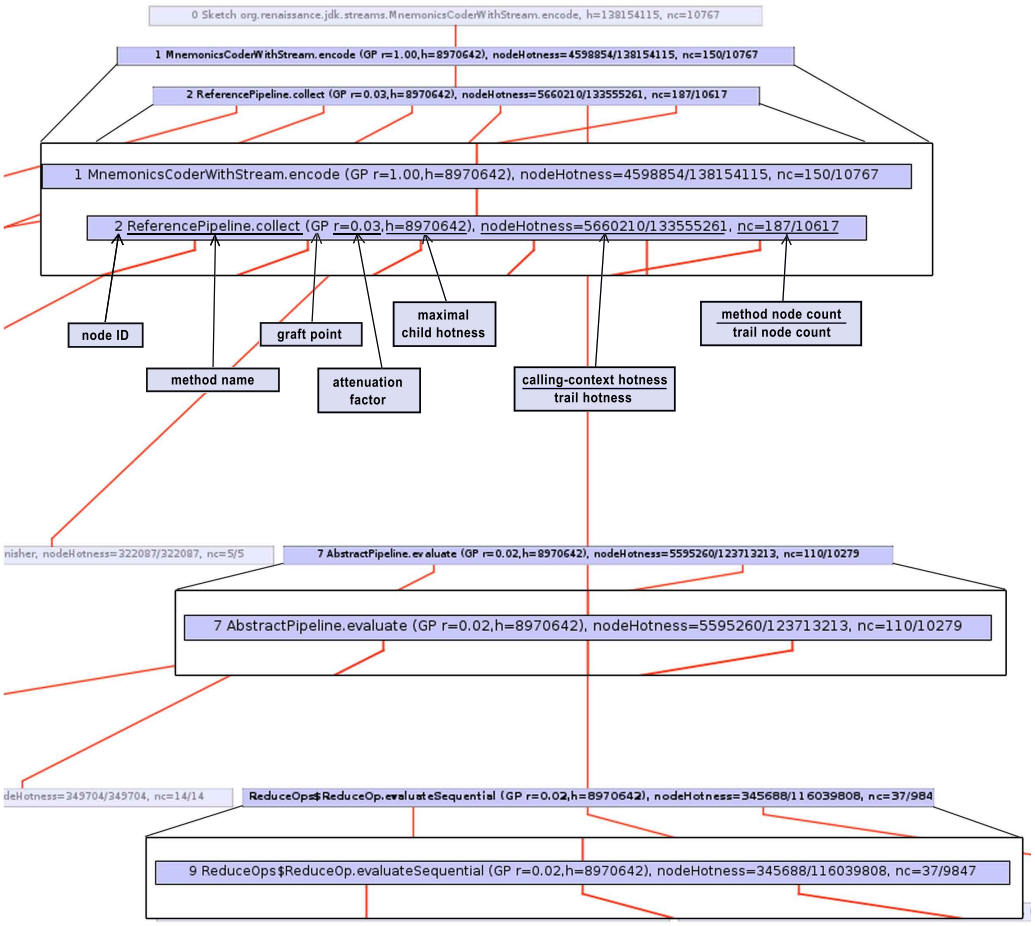


Слика 7.3: Део стабла инлајновања за методу `encode` из бенчмарка Mnemonics

извршаване делове стазе трагова и воде до методе која саджи главну петљу операције JDK `Stream`.

Следеће, биће разматрана стаза који одговара често извршаваној јединици компилације методе `encode`. Слика 7.4 садржи чворове стазе који директно одговарају истакнутим чворовима стабла инлајновања са слике 7.3. Истакнути чворови са слике 7.4 приказују неке од релевантних метрика попут фактора удела a_c, учестаности извршавања трага κ, учестаности извршавања кода представљеног неким подстаблом χ и величину ν изражену у броју чворова у графовској међурепрезентацији. Стаза садржи све истакнуте чворове стабла инлајновања у истом поретку позивалац-позвани. Чворови стабла инлајновања који су мапирани на чворове стаза имају увећан приоритет током фазе истраживања стабла.

На слици 7.3, приоритет ширења свих чворова који су припадају стази је $P = 0.29384$, што је приметно виша вредност у односу на окружујуће чворове, чији приоритет је између 0.026 и 0.049. Виша вредност је последица измењене вредности приоритета операције ширења φ из једначине 6.13. На овај начин се форсира инлајнер да приоритизира ширење дуж одговарајућег ланца позива и да истражи стабло позива дубље око овог региона често извршаваног кода. Важно је истаћи да се ширење осталих делова стабла



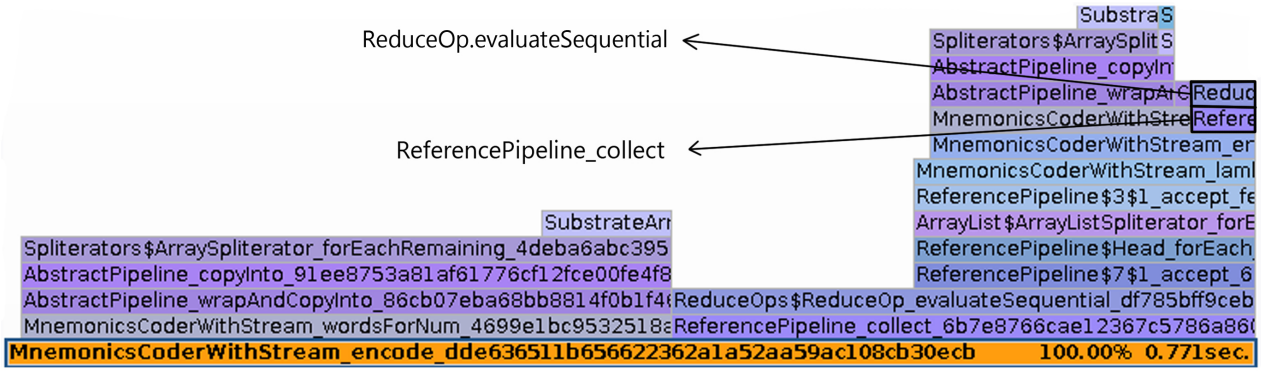
Слика 7.4: Део стазе трагова који одговара методи `encode` из бенчмарка Mnemonics

инлајновања и даље одвија пошто функција *penalty* из једначине 6.12 смањује количину истраживања подстабала која постану сувише велика.

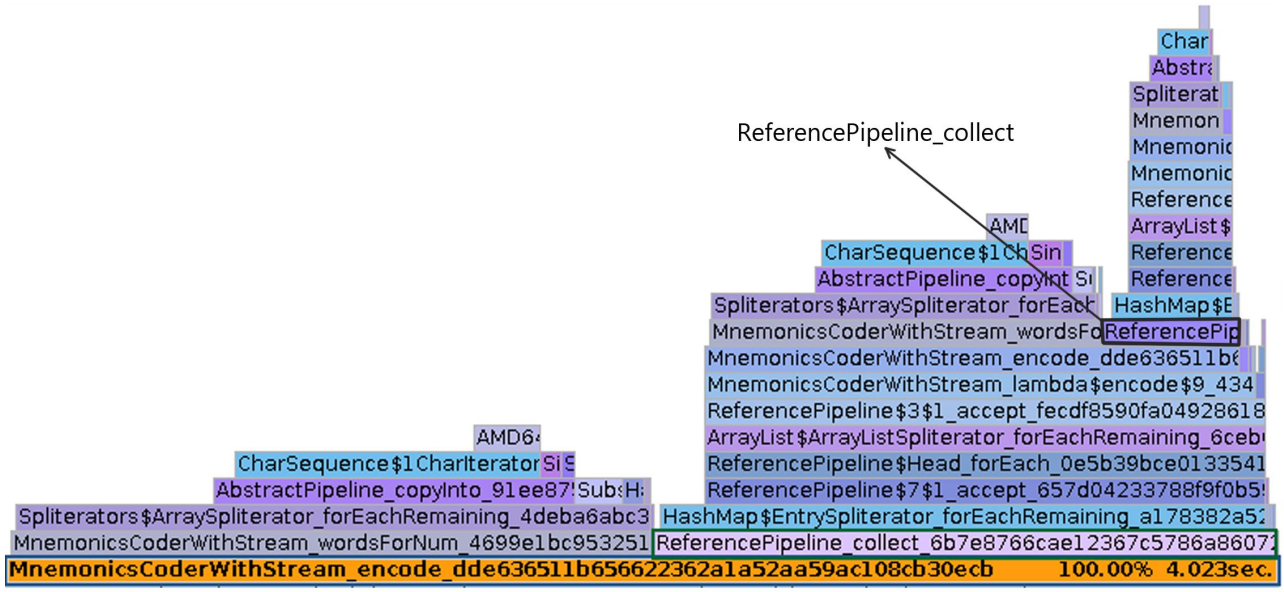
7.4 Пример превођења

У оквиру ове секције биће приказана анализа утицаја измењеног распореда превођења јединица компилације на примеру бенчмарка Mnemonics. Циљ секције је демонстрирати рад алгоритма на реалном примеру, и у те сврхе је изабран исти бенчмарк који је коришћен у претходној секцији. Анализа алгоритма ће бити спроведена над деловима програмског стека реконструисаним на основу потпуно контекстно осетљивих профила прикупљених уз помоћ екстерног алата за праћење извршавања програма. Стекови извршавања бенчмарка Mnemonics су визуелно представљени кроз графике. Слике 7.5 и 7.6 садрже најзначајније делове ових графика за AOT преведени бенчмарк Mnemonics, са и без примене алгоритма *PRINC*, респективно.

На основу овог примера могуће је илустровати ефекте две специфичне функционалности предложеног алгоритма:



Слика 7.5: Део графика извршавања који одговара бенчмарку Mnemonics преведеног уз примену алгоритма *PRINC*



Слика 7.6: Део графика извршавања који одговара бенчмарку Mnemonics преведеног уз примену постојећег алгоритма за превођење и инлајновање

- идентификација често извршаване јединица компилације и увећавање њиховог буџета за превођење,
- измена распореда превођења која као последицу има измењени скуп јединица компилације.

У оба случаја приказано је како ове функционалности утичу на бенчмарк истичући релевантне оквире (енг. *frame*) на стековима позива.

Како би се видели ефекти увећања буџета за инлајновање у често извршаваним јединицама компилације, треба размотрити стек позива са леве стране на слици 7.5, у којем је метода `ArraySpliterator.forEachRemaining` идентификована као значајна компилациона јединица. Нови алгоритам за инлајновање спроводи доста агресивније инлајновање у овој јединици компилације – све позване методе се инлајнују у `forEachRemaining`,

осим исечка кода (енг. *snippet*) `arraycopy`, као што ће бити објашњено у секцији 8.9. Са друге стране, без употребе новог алгоритма, неколико позиваних метода као што је `AbstractPipeline.copyInto` (као и њихових позиваних метода) остају као засебне компилационе јединице у графику извршавања, као што је приказано на слици 7.6 у оквиру стека са леве стране.

Ефекти измене редоследа превођења видљиви су на датом примеру тако што се скуп јединица компилације разликује између графика на сликама 7.5 и 7.6. У графику извршавања који током превођења користи нови алгоритам за инлајновање и распоред превођења (слика 7.5), јединица компилације која одговара методи `ReferencePipeline.collect` на десном стеку позива има једну преосталу методу коју позива (која није инлајнована) и то је метода `ReduceOp.evaluateSequential`. У подразумеваној верзији алгоритма, ова метода `evaluateSequential` не представља засебну компилациону јединицу и, уместо тога, инлајнована је у оквиру методе `ReferencePipeline.collect`. То узрокује да приликом превођења методе `ReferencePipeline.collect`, буџет превођења буде потрошен пре инлајновања `HashMap$EntrySpliterator.forEachRemaining` методе. Овај пример је само један од показатеља како погрешно усмерен буџет инлајновања може утицати на перформансе програма када је реч о значајним секцијама кода.

8 Евалуација

У оквиру евалуације биће приказани релевантни експерименти који потврђују допринос овог истраживања. У секцији 8.1 описани су услови спровођења експеримената и тестови коришћени у евалуацији. Секција 8.2 садржи поређење перформанси предложеног алгоритма за AOT превођење наспрам неколико постојећих решења, са нагласком на претходни алгоритам AOT превођења уз коришћење неизмењеног (*state-of-the-art*) инлајнера из компајлера *Graal* [5]. Како је утврђено да је инлајнер јако параметризован, да би се потврдило да перформансе не могу бити значајно поправљене једноставном модификацијом вредности параметара инлајнера, спроведен је експеримент који варира вредности значајних параметара. Главни резултати, као и најбоље вредности параметара за оба алгоритма приказани су у секцији 8.3. С' обзиром да је наглашено како је од значаја минимизовати промену величине генерисаног кода, у секцији 8.4 демонстрирано је у којој мери примењене оптимизације утичу на величину преведеног кода. Цена превођења огледа се између осталог и у трајању самог процеса компилације, па секција 8.5 садржи поређење времена превођења програма уколико се користи постојећа шема и у случају коришћења новог алгоритма за превођење.

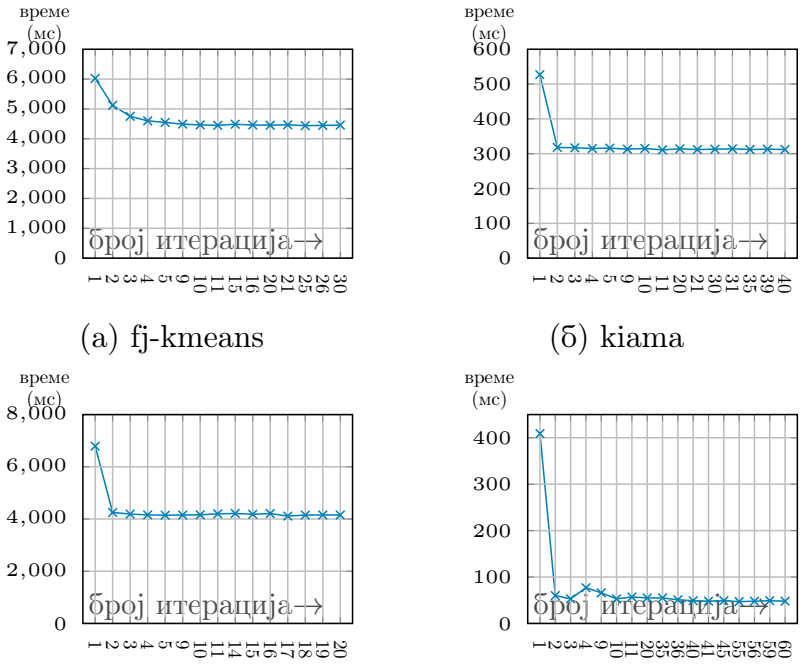
У секцији 8.6 евалуиран је утицај просечне дужине контекста у профилима са перформансама преведеног програма. Затим, у секцији 8.7 анализирано је како различите компоненте и параметри имплементираних хеуристика у предложеном алгоритму утичу на перформансе програма. Секција 8.8 приказује утицај процеса профилисања на процес превођења програма и као и на извршавање наменски преведених програма. Важно је истаћи да у оквиру JIT преводиоца постоје одређени механизми који се разликују у односу на AOT превођење, тако да се разлике у перформансама између програма преведених са ова два преводиоца не могу свести искључиво на оптимизацију инлајновања. Евалуација ће бити закључена прегледом неких од ових разлика у секцији 8.9.

8.1 Методологија екперимената

У наставку је дат опис спровођења експеримената. Коришћен је процесор Intel Xeon E5-2699v3 са 18 језгара. Током извршавања експеримената, функција која дозвољава повећање фреквенције такта процесора за време повећаног оптерећења (енг. *turbo boost*) је искључен, а језгра су покретана са фреквенцијом од 2.3GHz. Да би резултати били поуздани, за сваки добијени податак спроведено је 5 независних мерења над различитим процесима који укључују покретање извршног кода добијеног превођењем преводиоцем *GraalVM Native Image* (*Native Image 20.3.0-dev* (Java 8, commit: 7955c628b5c) са под-

разумеваним механизмом ослобађања меморије GC), или покретањем Java виртуелне машине (JVM) [126]. У оквиру мерења сваке инстанце, због постизања стабилних разултата, програм је покретан предефинисани број пута N. Број N је за сваки програм одабран тако да стабилно време извршавања буде постигнуто пре завршетка 60% укупног броја итерација програма. Време извршавања се сматрало стабилним уколико је укупно време извршавања итерација премашило 20 секунди и варијација перформанси резултата не прелази задати праг [127]. Време постизања стабилних резултата (енг. *startup time*) за *GraalVM Native Image* је брже од времена потребног да се постигне стабилно време извршавања на JVM.

На слици 8.1 се може видети да се стабилни резултати постижу већ након неколико итерација приликом AOT превођења са преводиоцем *Native Image*. Разлог лежи у томе што извршавању главне main методе програма претходи свега неколико почетних корака приликом иницијализације хипа извршног кода који се одвијају током извршавања програма [14]. Одређени број понављања сваког бенчмарка (програма) преведеног *Native Image* преводиоцем или превођеног и извршаваног на Java виртуелној машини дат је табеларно у оквиру табеле 8.1. Крајњи забележени резултати формирани су упросечавањем добијених времена за последњих 40% понављања бенчмарка за свако од појединачних 5 мерења. Резултати у већини експеримената садрже и стандардно одступање, осим у експериментима попут мерења промене величине генерисаног кода, у којима су резултати изузетно стабилни и одступање није значајно.



(а) fj-kmeans (б) kiama

(в) mnemonics (г) scrabble

Слика 8.1: Време постизања стабилних перформанси са преводиоцем *Native Image*

Табела 8.1: Број итерација потребан за постизање стабилног стања бенчмарка

Бенчмарк	**Native Image**	**JVM**
h2	10	16
fj-kmeans	20	40
mnemonics	20	60
par-mnemonics	20	60
philosophers	15	30
reactors	10	10
rx-scrabble	80	150
scala-stm-bench	30	60
scrabble	60	250
apparat	10	20
kiama	40	120
scalac	40	120
scaladoc	40	120
scalap	120	250
scalariform	40	120
tmt	12	16

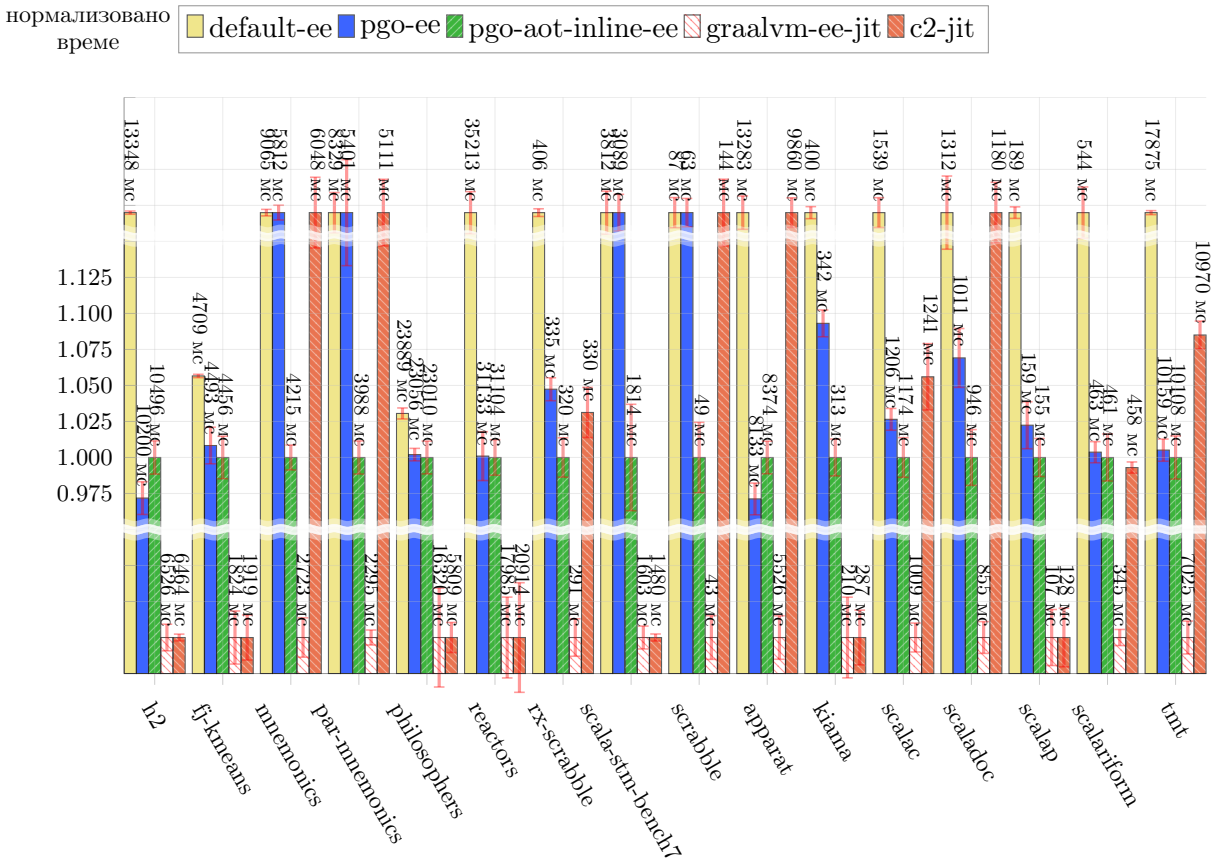
8.1.1 Опис тестова

Током тестирања коришћено је 16 бенчмарка из DaCapo [42], Scalabench [43] и Renaissance [44] скупа бенчмарка које је преводилац *Native Image* могао да преведе у тренутку извођења експеримената. Опис коришћених тестова је следећи. Део бенчмарка из Renaissance скупа тестова над којима су вршени експерименти (MNEMONICS, PAR-MNEMONICS и SCRABBLE) манипулишу подацима употребом механизма Java 8 Streams. RX-SCRABBLE и SCRABBLE тестови решавају идентичан проблем, али се први ослања на *RxJava* радни оквир (енг. *framework*) у реализацији. Бенчмарк PHILOSOPHERS решава познати конкурентни проблем филозофа који вечерају (енг. *the dining philosophers problem*) коришћењем *ScalaSTM* радног оквира. Овај радни оквир је коришћен и у оквиру SCALA-STM-BENCH7 бенчмарка [128]. REACTORS бенчмарк имплементира слање порука (енг. *message-passing*), а FJ-KMEANS бенчмарк спроводи познати *K-means* алгоритам употребом *Fork/Join* оквира. DaCapo бенчмарк H2 извршава скуп трансакција над базама података. Коришћено је 7 бенчмарка из скупа Scalabench. APPARAT бенчмарк оптимизује датотеке са посебним екстензијама, док KIAMA извршава посао процесирања језика. Бенчмарци SCALAC, SCALADOC и SCALAP представљају преводилац за Scala програмски језик, генератор документације за Scala програмски језик и декодер дела информација из .class датотеке (енг. *pickled data*), респективно. SCALARIFORM бенчмарк садржи код за форматирање програма написаних у програмском језику Scala, а бенчмарк TMT представља алат за анализу необележеног кода (енг. *unlabeled-code analysis*).

8.2 Поређење перформанси релевантних алгоритама за инлајновање

У оквиру главног експеримента пореде се времена извршавања програма у пет конфигурација од интереса. Свака конфигурација укључује другу комбинацију виртуелне машине, преводиоца и алгоритма за инлајновање. Три конфигурације односе се на покретање native image извршног кода (конфигурације *default-ee*, *pgo-ee*, *pgo-aot-inline-ee*). Конфигурација означена именом *default-ee* је основна и током превођења програма не користи профиле за унапређење оптимизација. Конфигурација под именом *pgo-ee* користи постојећу шему употребе профила у оптимизацијама, која је детаљније описана у секцији 3.5.2 и најрелевантнија је за поређење са алгоритмом предложеним у овом раду. Перформансе новог алгоритма за унапређење редоследа превођења и оптимизације инлајновања на основу профила – *PRINC* [41] представљене су конфигурацијом *pgo-aot-inline-ee*. У овом екперименту фиксиране су вредности параметара инлајнера у свим конфигурацијама. Четврта конфигурација (*graalvm-ee-jit*) односи се на JIT превођење употребом преводиоца *Graal Enterprise* и виртуелне машине *HotSpot*, док последња конфигурација *c2-jit* представља, такође JIT превођење употребом виртуелне машине *HotSpot*, али подразумеваним сервер (C2) компајлером.

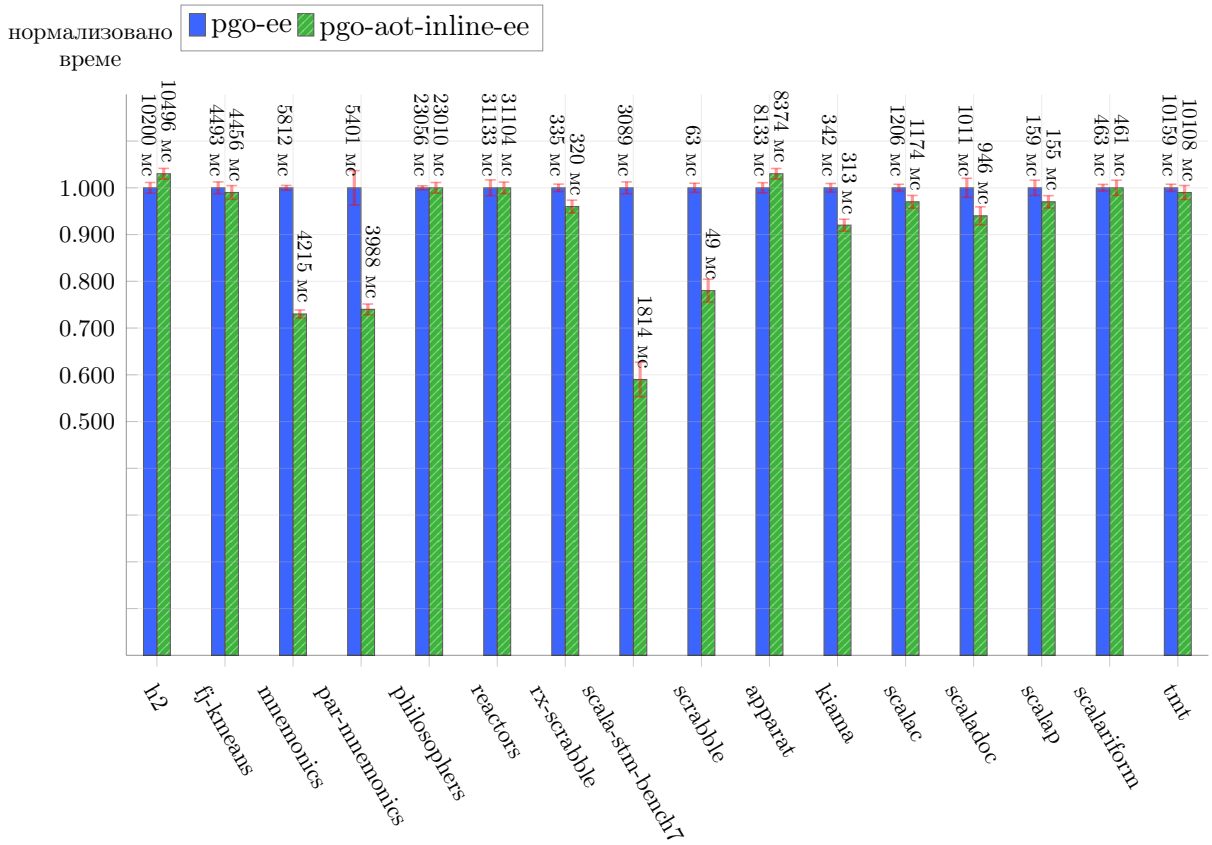
Резултати овог експеримента су приказани на слици 8.2. На x-оси су приказани



Слика 8.2: Време извршавања бенчмарка (нижа вредност је боља)

бенчмарци над којима је спровођен главни експеримент. График садржи пет колона за сваки бенчмарк, тако да свака колона презентује резултате бенчмарка који се покрећу једном од пет конфигурација. Сви резултати су нормализовани у односу на *pgo-aot-inline-ee* конфигурацију и приказани на y-оси. На y-оси фокусиран је опсег око оптималних вредности како би се нагласио утицај нове хеуристике за измену распореда превођења и унапређење оптимизације инлајновања на основу профила у поређењу са постојећом имплементацијом (*pgo-ee* конфигурација).

У поређењу са *default-ee* конфигурацијом, алгоритам *PRINC* побољшава перформансе бенчмарка до 55%. Прецизније изражено, три бенчмарка су убрзана за више од 50%, време извршавања два бенчмарка је унапређено између 40% и 50%, једног бенчмарка преко 35%, пет бенчмарка у опсегу $20 - 30\%$, три бенчмарка између 10% и 20%, а преостала два су убрзана до 10%.



Слика 8.3: Време извршавања бенчмарка за конфигурације *pgo-aot-inline-ee* и *pgo-ee* (нижа вредност је боља)

Разлика перформанси конфигурација *pgo-ee* и *pgo-aot-inline-ee* је, као што је истакнуто, најзначајнија зато што директно приказује утицај новог алгоритма у односу на претходну употребу профила у процесу компилације. Из тог разлога, перформансе бенчмарка које одговарају овим двема конфигурацијама додатно су издвојене на слици 8.3. Време извршавања бечмарка је на y-оси приказано од нуле, па су резултати нормализовани у односу на *pgo-ee* конфигурацију како распон подеока на yоси не би био значајно виши од 1.0, што график чини прегледнијим. Предложени алгоритам показује најзначајније побољшање над следећим бенчмарцима: SCALA-STM-BENCH7, MNEMONICS,

PAR-MNEMONICS и SCRABBLE. Њихово време извршавања је скраћено за 40%, 27%, 26% и 22%, респективно. KIAMA и SCALADOC бенчмарци су унапређени у опсегу од 5% до 10%, док је опсег убрзања бенчмарка RX-SCRABBLE, SCALAC и SCALAP између 2.5% и 5%. Пет бенчмарка је убрзано око 1%, а на два бенчмарка (H2 и APPARAT) уочено је успорење за мање од 3%.

8.3 Утицај параметара инлајнера на перформансе програма

Како би се показало да је поређење перформанси конфигурација фер, спроведен је низ експеримената који укључује варијацију вредности параметара који имају утицај на фазе експанзије (ширења стабла позива) и инлајновања у оквиру оптимизације инлајновања током компајлирања методе и добијени резултати су анализирани. Процес тражења најбољих вредности за ове параметре биће у наставку реферисан и као процес подешавања (енг. *tuning*). Подешавање параметара инлајнера представља итеративни процес чији је циљ проналазак комбинације вредности параметара који омогућавају најбоље перформансе програма преведених употребом таквог инлајнера.

8.3.1 Подешавање вредности параметара инлајнера

Процес проналажења најбољих вредности се спроводи над параметрима r, t_1 и t_2 из једначина 6.14 и 6.15. Ови параметри директно утичу на количину примене оптимизације инлајновања у оквиру компилационих јединица – *expansion inertia base-value* (параметар r) утиче на количину истраживања, тј. ширења стабла позива коју спроводи инлајнер, док *relative benefit coefficient* (параметар t_1) и *base target spending* (параметар t_2) утичу на одређивање границе бенефита (енг. *benefit threshold*), на основу које се доноси одлука да ли метода треба да буде инлајнована. На овај начин се, ефективно, ограничава буџет расположив за оптимизацију инлајновања [5]. Када параметар r узима веће вредности, инлајнер врши истраживање већег дела стабла позива. Веће компилационе јединице биће формиране у случају да параметар t_1 има мање вредности док параметар t_2 узима неку већу вредност.

Вредности сваког од параметара r, t_1 и t_2 су подешаване засебно у оквиру овог истраживања по следећем принципу. За проналажење најбоље вредности неког параметра P, на почетку се преостали параметри поставе на своје подразумеване вредности, које су претходно подешаване за постојећи инлајнер у оквиру компајлера *Graal* [5]. Затим се користи варијанта Simplex алгоритма [129, 130] како би се одредио опсег у оквиру кога се налази оптимална вредност параметра који се тренутно подешава P. За то време, вредности свих других параметара су фиксиране. Почевши од иницијалне вредности параметра, P_0, испитују се вредности $P_0 \pm \epsilon \cdot 2^i$ у сваком кораку i све док се не пронађе опсег који садржи тачку прегиба, такву да важи $[P_0 - \epsilon \cdot 2^j, P_0 + \epsilon \cdot 2^i]$ такво да постоји k за које $P_0 \pm \epsilon \cdot 2^k$ има бољу вредност у односу на границе узетог опсега. За одређивање критеријума „боље вредности" коришћена је функција геометријске средине над вред-

ностима перформанси свих бенчмарка. Затим је опсег подељен на 10 до 15 подједнаких делова, и тражене су оптималне вредности (енг. *optimal fitness*) у оквиру ових делова.

Најбоље вредности параметра r у оквиру подинтервала приказане су на x-оси на слици 8.4. док су вредности преосталих параметара фиксирани. Иако би процес проналажења најбољих вредности сваког параметра могао бити унапређен применом вишедимензионог подешавања свих параметара, такав процес је значајно компликованији. Комплексност се огледа у томе што је за сваки резултат (енг. *datapoint*) у оквиру сваког бенчмарка потребно два пута превести програм у извршни код (изградити native image извршни фајл) – код који наменски спроводи инструментацију и оптимизовани извршни код. Симултано подешавање више параметара је значајно скупље и захтева више времена извршавања на машинама него што је било доступно аутору у оквиру инфраструктуре.

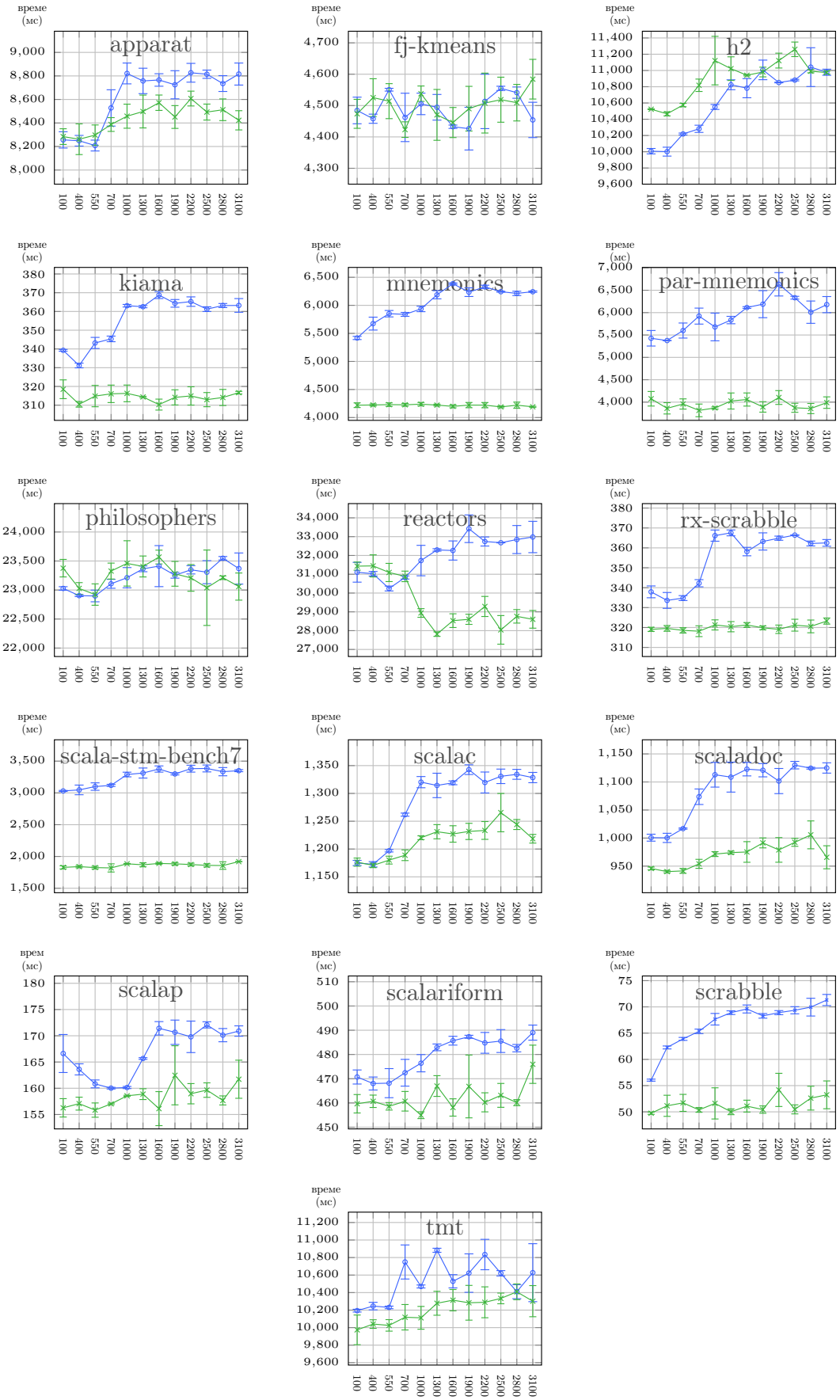
Услови процеса подешавања вредности параметара подразумевају исту методологију и инфраструктуру описану у секцији 8.1. За сваку вредност параметра спроведено је 5 засебних мерења у одвојеним инстанцама процеса. Покретане су две најрелевантније конфигурације за истраживање, које подразумевају превођење програма употребом *GraalVM Native Image* преводиоца. На основу процеса подешавања, изабране су следеће вредности параметара које су коришћене као подразумеване у оквиру новог алгоритма за инлајновање: $r = 550$, $t_1 = 0.0002$ и $t_2 = 300$.

8.3.2 Параметар *Expansion-Inertia Base Value*

Параметар *expansion-inertia base value* директно утиче на количину ширења стабла позива у фази истраживања (експанзије) пре него што инлајнер одлучи које ће од истражених делова стабла позива заправо инлајновати. Овај параметар одговара вредности r из једначине 6.14. У току истраживања експериментално је показано да је ово параметар који има највише утицаја на инлајнер и то је разлог зашто су његови ефекти на перформансе бенчмарка посебно приказани у овој секцији. Од интереса су пре свега перформансе конфигурација *pgo-ee* и *pgo-aot-inline-ee*.

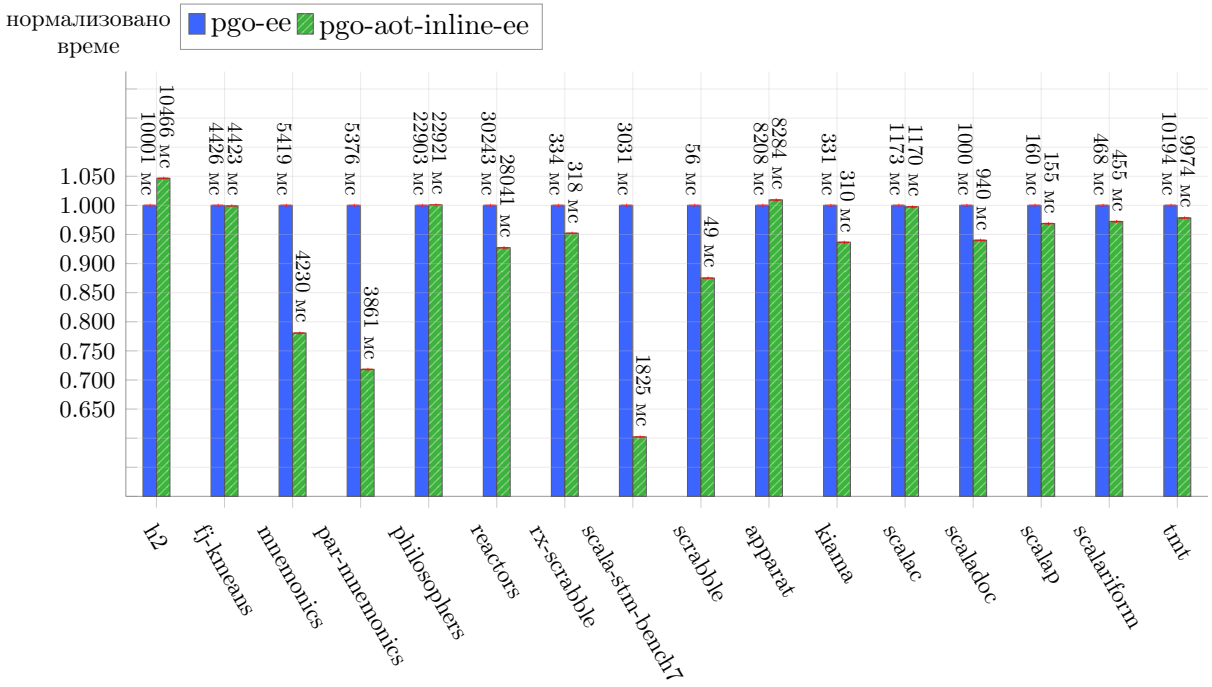
На слици 8.4 приказана су поређења ове две конфигурације засебно на сваком бенчмарку за значајне вредности овог параметра. Вредности параметра су поређане на x-оси, док се на y-оси налази време извршавања неког бенчмарка изражено у милисекундама. Приказане перформансе су анализиране како би се утврдила најбоља вредност за подразумевану вредност параметра. За већину бенчмарка обе конфигурације показују најбоље перформансе када параметар узима вредност између 400 и 550 са неколико изузетака када се боље перформансе постижу уколико је вредност параметра између 1000 и 1300. Перформансе новог инлајнера предложеног у овом раду показују се бољим у случају 11 бенчмарка за све вредности параметра. Од преосталих 5 бенчмарка, нови инлајнер показује боље перформансе на REACTORS бенчмарку за већину вредности параметра. За бенчмарке APPARAT, FJ-KMEANS и PHILOSOPHERS перформансе су углавном сличне за већину вредности параметра, док је погоршање перформанси уочено једино на H2 бенчмарку.

Спроведен је још један експеримент у циљу поређења перформанси две конфигурације, тако да се у свакој користи најбоља вредност параметра добијена на основу



Слика 8.4: Тражење најбоље вредности параметра *Expansion-Inertia Base Value* (*pgo-aot-inline-ee* (×) и *pgo-ee*(○))

претходног експеримента. Ова вредност не мора бити нужно иста за обе конфигурације јер су инлајнери независно подешавани и то за сваки бенчмарк понаособ. Резултати су приказани на слици 8.5. Бенчмарци су приказани на x-оси, а на y-оси је приказано нормализовано време извршавања и нижа вредност означава боље перформансе. Референтна вредност за поређење је 1.0 и односи се на *pgo-ee* конфигурацију, која не користи нови алгоритам за компајлирање и инлајновање. Конфигурација *pgo-aot-inline-ee*, која приказује резултате алгоритма *PRINC* даје боље перформансе на 11 бенчмарка и приближно једнаке перформансе на 3 бенчамрка. Ови независно подешавани резултати су релевантни за AOT компилацију зато што *offline* превођење различитих програма омогућава и подешавање параметара за сваки програм независно. Оваква парадигма није толико применљива на *online*, тј. JIT компилацију.

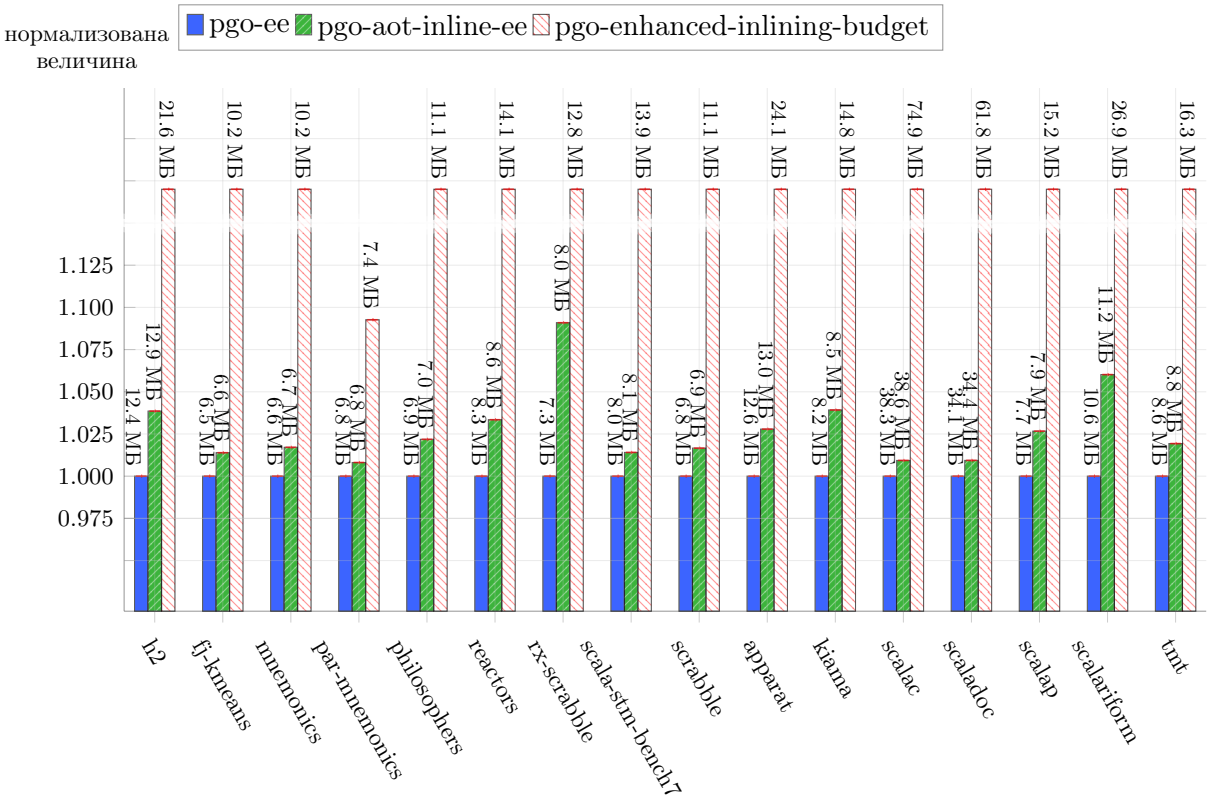


Слика 8.5: Време извршавања бенчмарка за оптималне вредности параметра *Expansion-Inertia Base Values* (нижа вредност је боља)

8.4 Утицај алгоритма на величину преведеног кода

Осим постизања бољих перформанси извршног кода, важан фактор у процени успешности примењеног решења је и цена алгоритма. Већ је претходно у раду наглашено да се цена агресивније примене оптимизације инлајновања доминантно огледа у увећању генерисаног извршног кода. У овој секцији је приказана евалуација предложеног алгоритма у контексту увећања генерисаног кода. Циљ овог експеримента је да покаже, пре свега, да повећањем буџета компилације искључиво често извршаваних компилационих јединицама не долази до значајног увећања укупног компајлираног кода, као што је и тврђено у секцији 6. На слици 8.6 приказано је поређење величине компајлираног кода за конфигурације *pgo-aot-inline-ee* и *pgo-ee*. Осим тога, како би се приказала разлика

између селективне примене појачаног инлајновања и глобалне примене ове оптимизације, на датој слици приказана и величина компајлираног кода за конфигурацију *pgo-enhanced-inlining-budget-ee*. Ова конфигурација подразумева примену идентичног буџета за инлајновање који је примењен над често извршаваним компилационим јединицама у оквиру новог алгоритма, али над свим компилационим јединицама без изузетка. Циљ је приказати цену коју би примена алгоритма за унапређење оптимизације инлајновања, али без детекције значајних делова кода, имала на величину изгенерисаног кода.



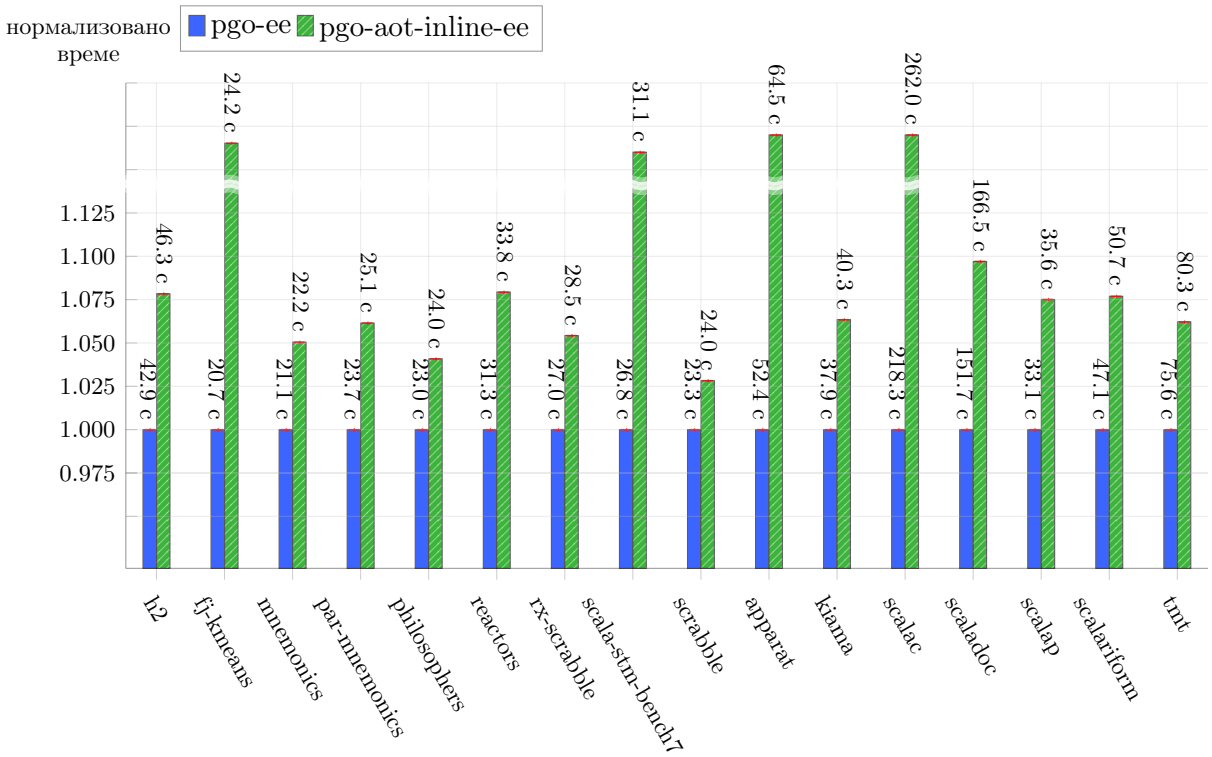
Слика 8.6: Величина компајлираног кода (нижа вредност је боља)

На x-оси приказани су бенчмарци, док је на y-оси приказана величина преведеног кода. Свака колона у оквиру графика представља резултат једног бенчмарка преведеног употребом једне од три наведене конфигурације. Резултати су нормализовани у односу на *pgo-ee* конфигурацију, која спроводи најмању количину инлајновања. Употребом инлајнера предложеног у овом раду величина преведеног кода је увећана, у односу на постојећи алгоритам инлајновања, између 0.8% на примеру PAR-MNEMONICS бенчмарка, и 9% за RX-SCRABBLE бенчмарк. Ипак, у случају чак 10 од 16 бенчмарка, ово увећање износи највише 2.5%. Са друге стране, када се буџет оптимизације спроводи глобално, увећање величине кода иде и до 2.5 пута. Најмање увећање кода примећено за конфигурацију *pgo-enhanced-inlining-budget-ee* износи око 9% за бенчмарк PAR-MNEMONICS, на коме је примећено најмање увећање и приликом селективне примене повећаног буџета оптимизације. Ипак, просечно повећање за конфигурацију *pgo-enhanced-inlining-budget* је око 2 пута, што је доста више у поређењу са новим инлајнером. Закључак који се изводи је да увећање у опсегу од 0.8% до 9%, постигнуто применом имплементираног

решења у оквиру *pgo-aot-inline-ee* конфигурације, није значајно и да је прихватљиво у пракси.

8.5 Утицај алгоритма на време превођења

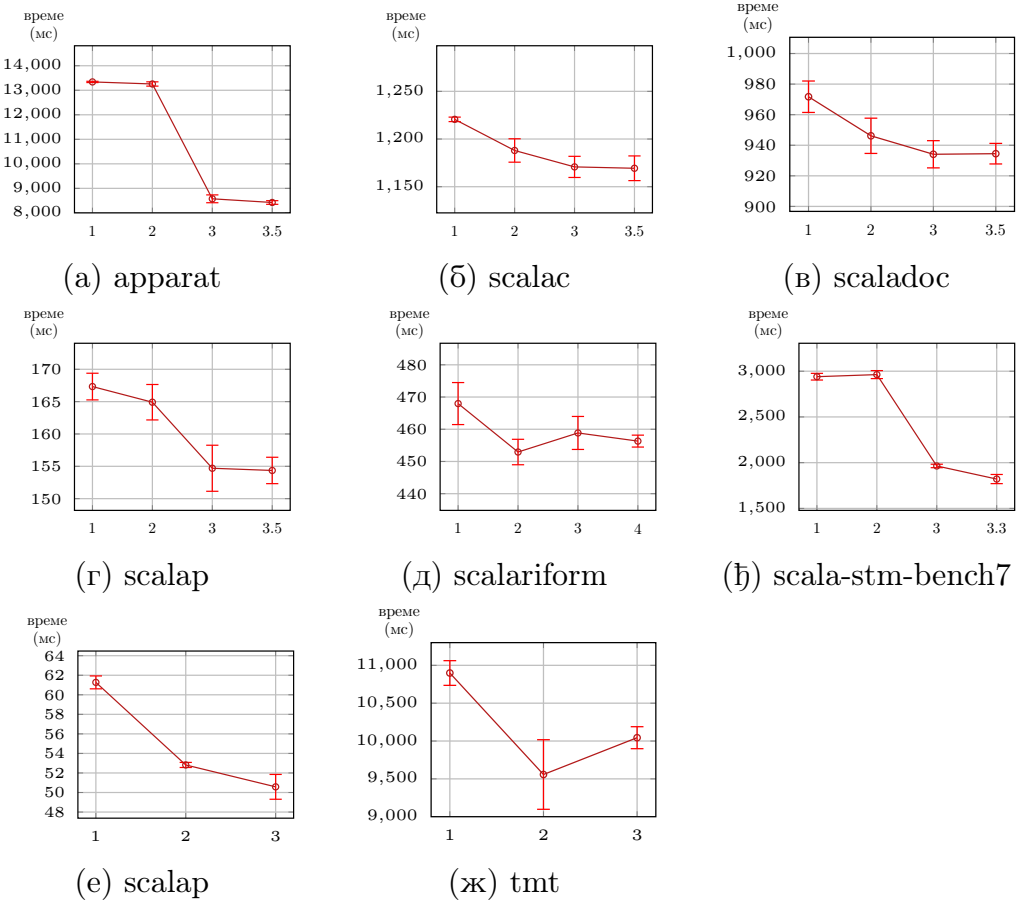
Осим у погледу увећања количине компајлираног кода, цена алгоритма који се укључује као део постојеће шеме компилације огледа се и у утицају на време потребно да се програм преведе тако модификованом шемом. Иако се само превођење одвија пре времена извршавања, и због тога није кључно да увећање трајања превођења (енг. *compile-time overhead*) буде минимално, у овој секцији се показује да је промена времена превођења, узрокована употребом предложеног алгоритма, разумна и као таква практично примењива. На слици 8.7 приказано је поређење времена потребног за превођење програма за конфигурације *pgo-aot-inline-ee* и *pgo-ee*. На x-оси налазе се бенчмарци, а на y-оси приказано је нормализовано време превођења. Свака колона представља време превођења једног бенчмарка када је коришћена једна од две наведене конфигурације превођења. Резултати су нормализовани у односу на *pgo-ee* конфигурацију. Спроведени експеримент показује да нови алгоритам доводи до повећања времена компилације до 23%, што је примећено на бенчмарку APPARAT. Над већином бенчмарка (12 од 16 бенчмарка) време превођења је увећано доста мање и то између 2.8% и 10%.



Слика 8.7: Време превођења (нижа вредност је боља)

8.6 Утицај дужине контекста на перформансе

Један од циљева предложеног алгоритма је да искористи контекстно осетљиве профиле како би унапредио одлуке које се доносе у оквиру оптимизације инлајновања. Дужина контекста представља број локација садржаних у оквиру контекста једног профила. Са порастом дужине контекста углавном расте и прецизност информације садржане у том профилу. Варирајући просечну дужину контекста у профилима, у оквиру ове секције показује се да постоји узрочно последична веза између дужине контекста и перформанси новог алгоритма за инлајновање. Како би се постигла измена дужине контекста, примењен је поступак ограничавања „дубине" примене оптимизације инлајновања у оквиру генерисања наменског извршног кода за профилисање. Истовремено је примењено и контролисање буџета оптимизације инлајновања до тренутка доласка до засићења, тј. немогућности генерисања дужих контекста у профилима. Један од разлога за достизање лимита просечне дужине профила лежи у виртуелним позивима, који у највећем броју случајева не могу бити инлајновани у оквиру извршног кода намењеном профилисању услед већег броја конкретних метода које могу бити позване на датој локацији, а за које не постоји никаква информација о стварној фреквенцији позивања.

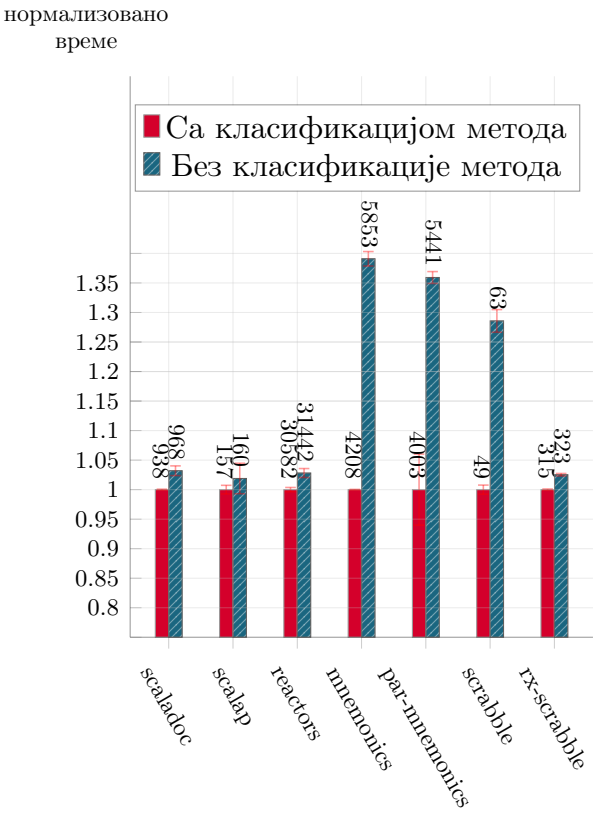


(а) apparat (б) scalac (в) scaladoc

(г) scalap (д) scalariform (ђ) scala-stm-bench7

(е) scalap (ж) tmt

Слика 8.8: Утицај дужине контекста на време извршавања бенчмарка (нижа вредност је боља)



Слика 8.9: Транзитивно превођење често позиваних метода као значајних јединица компилације (ниже вредности су боље)

Слика 8.8 приказује подскуп бенчмарка на којима је уочен значајан утицај промене дужине контекста. Анализа резултата показује нешто мањи утицај на преостале бенчмарке, па њихови графици нису представљени у раду. Сваки приказани график садржи податке о перформансама када просечна дужина контекста у профилима варира између 1 и најдуже просечне дужине која је постигнута за парцијалне контексте за конкретан бенчмарк. Горња граница варира у односу на то који бенчмарк је у питању, али је у случају већине бенчмарка постигнута максимална просечна дужина између 3 и 4 локације. Бенчмарци APPARAT и SCALA-STM-BENCH7 показују највеће варијације перформанси у односу на дужину контекста. За просечну дужину контекста између 3 и 3.5, на овим бенчмарцима постижу се боље перформансе за чак $35 - 40\%$ у односу на превођење потпомогнуто контекстно неосетљивим профилима (код којих су контексти дужине 1). Што се тиче преосталих приказаних бенчмарка, SCRABBLE демонстрира убрзање од 18%, TMT 12%, SCALAP око 8%, док се за остале бенчмарке уочава убрзање између 4% и 5% када се користе дужи контексти у профилима.

8.7 Утицај хеуристике инлајнера на перформансе

У оквиру експеримента који ће бити представљен у овој секцији, анализиран је утицај појединих фрагмента новог алгоритма за превођење и инлајновање на перформансе. У оквиру сваког експеримента, презентују се резултати на подскупу бенчмарка на којима је најуочљивији утицај анализираних компоненти алгоритма.

8.7.1 Транзитивно превођење често позиваних метода са увећаним буџетом за инлајновање

Након што је нека метода компајлирана, све преостале неинлајноване методе које се позивају из ове методе, постају корене методе скупа компилационих јединица које се следеће преводе. Овакав начин превођења ће надаље бити референсан као „транзитивно компајлирање". Неинлајноване позиване методе су или означене као често извршаване и преводе се са већим буџетом или се, у супротном, преводе са подразумеваним буџетом. Детаљи одлучивања су већ претходно објашњени у секцији 6.6. У експерименту описаном у овој подсекцији пореди се предложени алгоритам са варијацијом истог алгоритма у којој је IsHot полиса из једначине 6.18 увек замењена са IsHot = $\bot$.

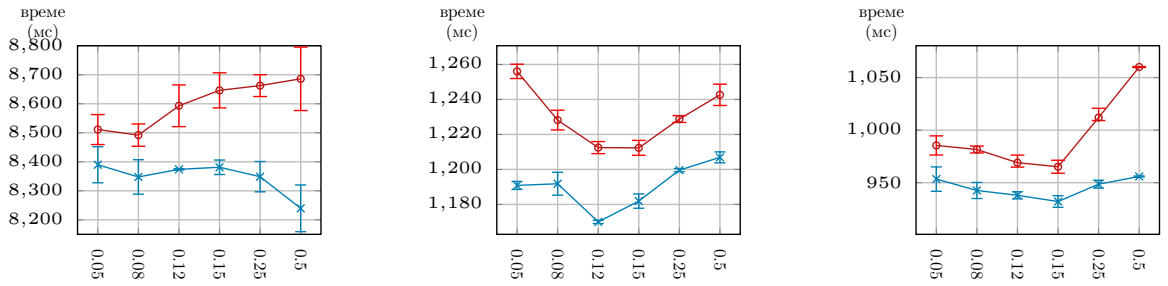
На слици 8.9 црвене колоне односе се на алгоритам у својој неизмењеној форми, док плаве колоне приказују перформансе алгоритма када се сви преостали позиви након формирања компилационе јединице третирају као ретко извршавани. Приказани су резултати за неколико бенчмарка за које је примећено да ова компонента алгоритма самостално показује значајнији утицај на перформансе програма. Резултати су нормализовани у односу на основну верзију алгоритма, у којој се преостали позиви класификују према учестаности извршавања. Опсег од интереса је истакнут на y-оси графика. Побољшање перформанси које се остварује применом IsHot хеуристике из једначине 6.18 је у опсегу између 22% и 28% за све бенчмарке који доминантно користе библиотеку *Java Streams* док је у случају осталих бенчмарка забележено убрзање до 5%.

8.7.2 Ширење стазе трагова

Предложени алгоритам итеративно шири стазе трагова, почевши од иницијалног скупа стаза употребом могућих контекста позивања. Детаљи овог процеса претходно су објашњени у секцији 6.4.1, а CallingContexts полиса задужена за одређивање скупа кандидата-контекста дефинисана је у једначини 6.3. Како би се измерио утицај експанзије стаза трагова на алгоритам, у овом експерименту ширење је онемогућено променом полисе на следећи начин: CallingContexts = $\varnothing$. Искључивањем операције ширења стаза из алгоритма, скуп често извршаванх компилационих јединица садржи само методе које представљају коренове контекста позива иницијално одабраних профила.

На слици 8.10 приказано је поређење времена извршавања бенчмарка неизмењеног алгоритма *PRINC*, у коме постоји операција ширења стаза трагова (резултати су приказани плавом кривом) и модификованог алгоритма у коме је експанзија стаза трагова искључена (резултати су приказани црвеном кривом). График истовремено презентује процес одређивања најбоље вредности прага (ψ из једначине 6.1) за одабир контекста који се иницијално сматрају често извршаваним (енг. *hot-context threshold tuning process*).

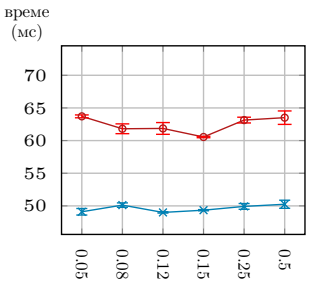
На основу датог графика закључује се да независно од избора вредности прага ψ време извршавања бенчмарка Apparat, Kiama, Scalariform, Scalac, Scaladoc и Scrabble је увек стриктно боље уколико је експанзија укључена. Поређењем најбољих вредности на обе криве са графика показано је да се време извршавања бенчмарка побољшава до 20% захваљујући овој компоненти алгоритма.



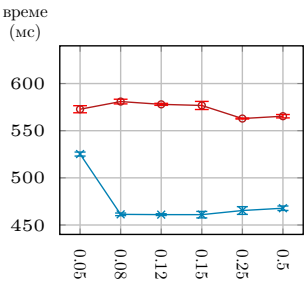
(а) apparat

(б) scalac

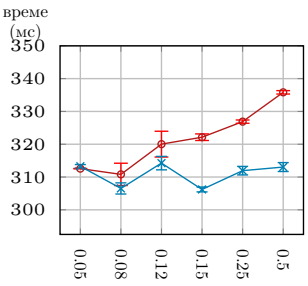
(в) scaladoc



(г) scrabble



(д) scalariform



(ђ) kiama

Слика 8.10: Тражење најбоље вредности прага за укључивање у иницијални скуп често извршаваних контекста (са операцијом експанзије $\times$ и без операције експанзије $\circ$; ниже вредности су боље)

8.7.3 Величина почетног скупа често извршаваних контекста

Константа ψ из једначине 6.1 представља однос времена које треба да буде проведено у делу кода представљеним контекстом из неког профила и укупног времена извршавања програма да би се дати профил сматрао често извршаваним. Сви профили у којима се проводи довољна количина времена биће укључени у иницијални скуп често извршаваних профила. У овом одељку показује се да постоји опсег оптималних вредности константе ψ када је активна компонента алгоритма ширења стаза трагова. Са друге стране, када је експанзија онемогућена, оптимална вредност константе налази се у много ужем опсегу, који се мења зависно од бенчмарка.

Слика 8.10 садржи графике подскупа бенчмарка на којима се најбоље уочава утицај варирања вредности константе ψ. На x-оси налази се скуп вредности које су узете за константу, а на y-оси време извршавања бенчмарка када се у алгоритму користи конкретна вредност константе ψ. Виша вредност константе, тј. виша вредност прага подразумева мањи број профила укључених у иницијални скуп, а самим тим и мањи број компилационих јединица иницијално означених као често извршаване. Повећањем вредности прага преко одређене границе, задати однос проведеног времена у делу кода везаним за неки контекст у односу на укупно време извршавања постаће недостижан, и, као последица, ниједна компилациона јединица неће бити компајлирана са већим буџетом. Ово је еквивалент *pgo-ee* конфигурације.

Експерименти ради утврђивања оптималне вредности константе ψ вршени су на два начина – у комбинацији са операцијом ширења стаза трагова и када је експанзија искључена. У случају када операција експанзије није активна, оптимална вредност константе

ψ лежи у опсегу између 0.05% и 0.5% и утиче на перформансе и до 10%. На већини бенчмарка смањење броја често извршаваних компилационих јединица испод извесног прага негативно утиче на перформансе, али такође и претерано смањење вредности прага узрокује велики број различитих стаза трагова, односно лошу детекцију често извршаваних региона кода, што доводи до лошијих перформанси. Најбоље перформансе се, у случају онемогућавања операције ширења, за већину бенчмарка постижу када константа узима вредност између 0.12% и 0.15%. За неке бенчмарке се, ипак, попут бенчмарка Kiama показује да имају више користи од већих скупова иницијално одабраних често извршаваних профила. Са друге стране, када се користи компонента експанзија стаза, најбоље изабране вредности константе ψ налазе се у опсегу између 0.08% и 0.25% за већину бенчмарка, одакле се изводи закључак да је алгоритам у овој варијанти мање осетљив на избор вредности константе ψ.

8.8 Утицај прикупљања профила на перформансе

Наредни експеримент, чији су резултати приказани у оквиру поглавља евалуације, односи се на утицај процеса прикупљања профила на време потребно за превођење и извршавање програма. Као што је објашњено у секцији 3.5, *GraalVM Native Image* подразумева прикупљање профила у засебном режиму превођења – формирањем инструментационог извршног кода. Једна од фаза превођења током формирања инструментационог извршног кода задужена је за убацивање инструментационих бројача у графовску међурепрезентацију компајлера, која одговара појединачној методи програма. Профили се прикупљају извршавањем инструментационог извршног фајла. Превођење програма са улазним профилима представља потпуно засебан процес који се спроводи формирањем оптимизационог извршног фајла.

У претходним експериментима, приказаним у текућем поглављу евалуирани су различити аспекти перформанси оптимизационог извршног фајла. Иако алгоритам презентован у овој тези не мења начин на који се изводи прикупљање профила у преводиоцу *GraalVM Native Image*, овде ће бити приказана времена потребна за превођење и извршавање инструментованих извршних кодова бенчмарка, који су коришћени и у преосталим експериментима.

У табели 8.2 демонстриран је утицај прикупљања профила на репрезентативним бенчмарцима из Renaissance, DaCapo и Scalabench скуповима бенчмарка. Поређено је време потребно да се преведе изворни код уз додавање инструментационих бројача и да се потом изврши инструментовани код наспрам превођења и извршавања подразумеваног, неинструментованог кода, у чије превођење нису укључене оптимизације на основу профила. Подразумевани режим превођења се користи у компарацији како би се подвукао утицај самог профилисања, а његове перформансе су приказане и у секцији 8.2 (конфигурација *default-ee*).

Примећено је да инструментација утиче на време превођења тако што га увећава до 30% у случају већине бенчмарка. Ипак, на неким бенчмарцима попут Scalac, примећено је нешто веће увећање – 37%. Инструментација може да уведе успорење извршавања генерисаног кода до 4 пута, које је забележено за бенчмарк Reactors.

Табела 8.2: Утицај прикупљања профила на времена превођења и извршавања програма

	Време превођења [с]		Време извршавања [с]	
Бенчмарк	**Подразумевано**	**Профилисање**	**Подразумевано**	**Профилисање**
h2	36.6	46.1	13.3	23.6
mnemonics	20.2	25.7	9.6	21.7
reactors	26.4	34.6	35.2	141.3
scalac	119.1	164.2	1.5	5.7
scalariform	38.4	46.3	0.5	1.1

Како се профилисање спроводи над наменским извршним кодом, и покреће се у засебном процесу, перформансе оптимизованог извршног кода нису погођене дужим временом које је потребно да се добије и изврши инструментациони код.

Поступак прикупљања профила у конкретном окружењу је могуће оптимизовати, али је изван опсега овог истраживања. Друга предност постојања одвојеног процеса прикупљања профила је та што једном када су профили смештени у датотеци, иста датотека може бити коришћена у више процеса генерисања оптимизованих извршних кодова. Другим речима, превођење оптимизованог програма не захтева превођење и извршавање у инструментационом режиму у случају да постоји барем једна датотека која садржи профиле, а која је настала на основу неког од претходних извршавања истог улазног програмског кода.

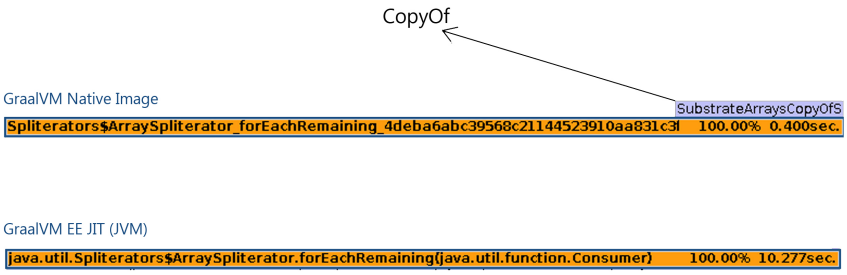
8.9 Утицај различитих реализација механизама преводилаца *Graal* и *Native Image* на перформансе

У главном експерименту из секције 8.2 на слици 8.2 приказане су перформансе бенчмарка извршаваних у неколико конфигурација. Између осталих, приказана је конфигурација која користи компајлер *Graal* у оквиру Java виртуелне машине (JVM) (HotSpot или OpenJDK са JIT превођењем) и која показује значајно боље перфомансе у поређењу са конфигурацијама које користе исти компајлер, али у комбинацији са *GraalVM Native Image* (са AOT превођењем). Упркос унапређењима перформанси инлајнера у оквиру преводиоца *GraalVM Native Image*, постоји потенцијал за постизање бољих перформанси, које би биле приближније онима које одговарају систему *GraalVM* на JVM. Осим прилика за унапређење хеуристике оптимизације инлајновања, у овој секцији се истиче и неколико разлога који такође доприносе разликама у перформансама између две виртуелне машине. У наставку ће бити приказани неки од механизама и оптимизација који су имплементирани другачије или нису подржани у оквиру преводиоца *Native Image* у наведеној верзији која је била актуелна у тренутку извођења евалуације.

8.9.1 Исечци кода

Исечци (енг. *snippets*) представљају механизам у систему *GraalVM*, који се користи да се изрази, на ограниченом подскупу програмског језика Java, имплементација на ниском нивоу (енг. *low level*) операција на високом нивоу (енг. *high level*) [131]. На пример, `instanceof` (испитивање типа током извршавања програма) у случају класе из које нема изведених класа, може бити изражено као читање из заглавља објекта чији се тип испитује и поређење прочитане вредности са константом. Слично, позив `System.arraycopy` (који копира елементе из једног низа у други) може бити изражен као петља директно у компилационој јединици која позива `arraycopy`, а затим може бити оптимизован уколико су типови елемената познати. Већина исечака је архитектурално независна захваљујући њиховом изражавању у вишем програмском језику. Ипак, могу садржати и неке блокове који су специфични за одређену платформу у форми специјалних позива метода (енг. *intrinsic method calls*), који спроводе мапирање на одређене машинске инструкције, тако да имплементације исечака могу варирати у зависности од преводиоца, хардвера и виртуелне машине.

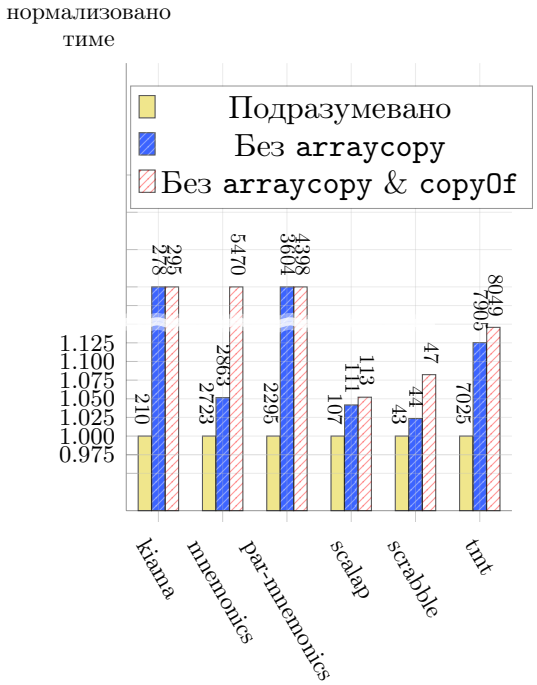
Разматра се метода `Arrays.copyOf` из JDK, која дуплира задати низ и њена замена кодом који имплементира исто понашање у оквиру JVM и *Native Image*. Слика 8.11 садржи оквире са графика визуелизованих стекова извршавања, који одговара бенчмарку Mnemonics у две конфигурације – када се компајлер *Graal* користи у комбинацији са виртуелном машином HotSpot и у оквиру система *Native Image*. Сваки оквир на стеку представља једну јединицу компилације, а оквири су на графику распоређени тако да су оквири позваних метода смештени на оквире метода позивалаца. Стек позива на слици 8.11 садржи јединицу компилације за методу `ArraySpliterator.forEachRemaining` из Java библиотеке `Stream`. На графику који приказује извршавање програма преведног пре извршавања у систему *Native Image* постоји засебан оквир за `copyOf`, која представља специјализовану *Native Image* методу за алокацију и копирање



Слика 8.11: Разлика у имплементацији исечка `copyOf` у преводиоцу *Native Image* и на JVM (део графика визуелизованих стекова извршавања)

низова – исечак једноставно позива уграђену методу. На JVM нема засебног позива – исечак директно уграђује дупликацију низа у графовску међурепрезентацију методе `forEachRemaining`. Док за веће низове обично нема значајније разлике у перформансама између ова два приступа, уколико постоји већи број позива `copyOf` над мањим низовима, режијски трошкови позива долазе до изражаја.

На слици 8.12, приказане су разлике у перформансама на примеру шест бенчмарка, на којима је примећен највећи утицај искључивања имплементације исечака за `System.arraycopy` и `Arrays.copyOf` на JVM. Искључивање имплементације не одговара у потпуности стању у преводиоцу *Native Image* зато што се имплементација `arraycopy` у програмском језику C++ за Java виртуелну машину разликује у односу на механизам имплементације у систему *Native Image*. Ипак, овај експеримент показује утицај који исечци `arraycopy` и `copyOf` имају на време извршавања бенчмарка.



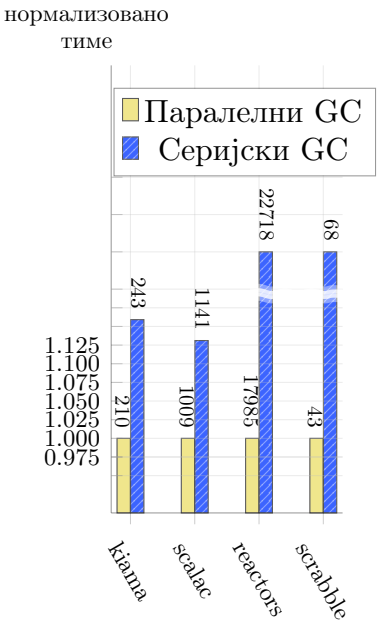
Слика 8.12: Утицај исечка за копирање низова на перформансе програма (ниже вредности су боље)

8.9.2 Ослобађање меморије

Програмски језик Java користи аутоматски механизам за ослобађање меморије (енг. *garbage collection*, скр. GC). У оквиру JVM доступно је неколико GC имплементација за сваку JDK верзију. Бенчмарци су у покретани са *GraalVM Enterprise* верзијом и JDK 8, који подразумевано користи паралелни алгоритам ослобађања меморије (енг. *parallel garbage collection* [132]). Паралелни GC „замрзава" нити апликације у току ослобађања меморије. Као што и само име каже, паралелни механизам ослобађања меморије користи више нити, чиме се постижу боље перформансе у односу на серијски GC алгоритам, који је подразумевано коришћен у ранијим JDK верзијама. Подразумевани механизам

за ослобађање меморије је у верзији преводиоца *Native Image* коришћеног у овом пројекту серијски алгоритам, који паузира нити апликација, али користи само једну нит за само ослобађање меморије [133]. Иако су имплементације два серијска алгоритма у оквиру JDK8 и *Native Image* различите, корисно је упоредити перформансе бенчмарка који се извршавају са JDK 8 уз коришћење паралелног и серијског алгоритма, пошто такво поређење даје увид у значај који би паралелна имплементација имала у окружењу *Native Image*.

На слици 8.13 приказана су четири бенчмарка покретана са *GraalVM EE JDK 8*, чије су перформансе значајно боље уколико се користи паралелни GC механизам у односу на серијски. Највећи утицај на перформансе уочљив је у случају бенчмарка `Scrabble` – око 35%. У случају већине од преосталих 12 тестираних бенчмарка, разлика у перформансама је до 5%.



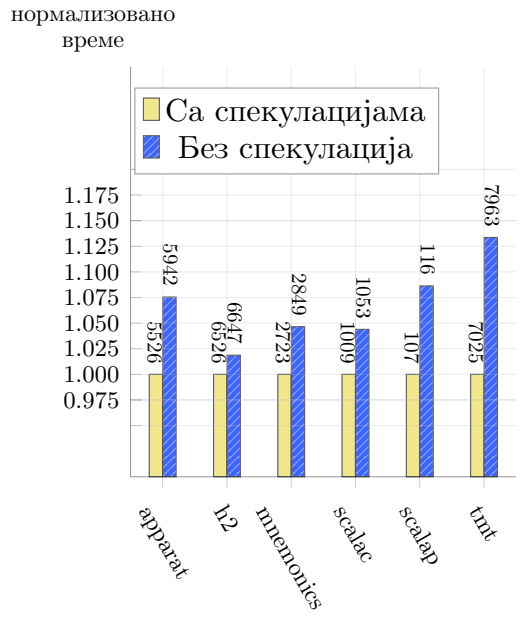
Слика 8.13: Поређење алгоритама за ослобађање меморије за GraalVM EE JIT

8.9.3 Спекулативне оптимизације

Превођење у време извршавања на JVM користи спекулативне оптимизације (енг. *speculative optimizations*), које предвиђају карактеристике вредности у програму како би боље оптимизовале код и допринеле побољшању перформанси. Свакој спекулативној оптимизацији кода претходи провера да ли је спекулација валидна, и она мора бити јефтина за израчунавање. Уколико током извршавања провера не прође, извршавање оптимизованог кода би било неисправно, па се окида механизам деоптимизација, односно извршавање се враћа у интерпретер. Након тога се код компајлира поново, али са мањим нивоом спекулација [6, 134]. *Native Image* не подржава деоптимизације, а самим тим ни спекулације. У овој секцији је на подскупу бенчмарка приказано како спекулације побољшавају перформансе програма извршаваних на JVM.

Идентификоване су три спекулативне оптимизације у компајлеру *Graal*, за које је примећен највећи утицај на перформансе у случају њиховог искључивања: спекулативно померање услова који обезбеђују исправност извршавања (енг. *speculative guard motion*) [6], које генерализује ове услове тако да омогући да буду што више инваријантни у односу на тело петље (енг. *loop-invariant operation*) (ова оптимизације је укључена и у *GraalVM Community Edition* [135]), оптимистична анализа заузећа меморије (енг. *optimistic aliasing analysis*), која спекулише да ли се два показивача односе на исти објекат, и спекулативно испитивање типова (енг. *speculative type-checking*), које спекулише да најпре може бити испробано једноставније испитивање типова. Наведене оптимизације постоје и у преводиоцу *Native Image*, али нису спекулативне – деоптимизација никада неће бити омогућена.

Како би се перформансе на JVM приближиле перформансама преводиоца *Native Image*, у следећем експерименту искључене су спекулације у оквиру наведених оптимизација (док су оптимизације и даље активне). Слика 8.14 приказује шест бенчмарка на којима је уочено значајно погоршање перформанси при искључивању спекулација: Apparat, H2, Mnemonics, Scalac, Scalap и Tmt. Бенчмарци су приказани на x-оси, док је на y-оси приказано нормализовано време извршавања. Жуте колоне представљају подразумевану верзију, која укључује спекулације, а плаве колоне верзију без спекулација. Резултати су нормализовани у односу на подразумевану верзију. На овим бенчмарцима успорење као последица онемогућавања спекулација у издвојеним оптимизацијама варира између 13% и 2%



Слика 8.14: Утицај спекулативних оптимизација на перформансе (ниже вредности су боље)

9 Поређење алгоритма *PRINC* са релевантним решењима

Циљ ове секције је анализа како алгоритам предложен у овом раду може бити прилагођен и имплементиран у оквиру других програмских преводилаца. У ову сврху одабрана су два најчешће коришћена компајлерска окружења – LLVM [45, 136, 137] и GCC [46]. Анализа и дискусија у овој секцији заснивају се на следећим аспектима решења: инфраструктури за прикупљање и употребу профила, међурепрезентацији преводиоца, детекцији често извршаваног кода, оптимизацијама заснованим на профилима и оптимизацији инлајновања. Решење није имплементирано у оквиру поменутих преводилаца, већ је циљ ове дискусије демонстрација да предложени алгоритам може бити искоришћен и у оквиру других преводилаца, чиме се овом раду даје шири контекст.

Након тога биће приказано детаљно поређење имплементираног алгоритма са најрелевантнијим решењима из литературе. Преглед литературе из секције 2.6 у наставку је проширен компарацијом главних аспеката решења.

9.1 GCC

Међурепрезентација. GCC користи три главне компајлерске међурепрезентације: GENERIC [138], GIMPLE [139] и RTL [140]. GENERIC је репрезентација заснована на стаблима која не зависи од улазног програмског језика и користи је предњи део преводиоца. Средњи део преводиоца користи репрезентацију која се назива GIMPLE. Ово је такође репрезентација застована на стаблима и директно се изводи из међурепрезентације GENERIC. GIMPLE има троадресну форму и додатна ограничења у односу на GENERIC. GIMPLE стабло се касније трансформише у SSA форму, над којом се спроводи већина оптимизационих пролаза, укључујући и оптимизацију инлајновања. Последња репрезентација RTL је репрезентација која оперише на ниском нивоу и одговара циљној архитектури преводиоца.

Детекција често извршаваних делова кода. Бројне оптимизације у преводиоцу GCC имају корист од информација о често извршаваним деловима кода [141]. Учестаност извршавања кода се у GCC-у одређује на нивоу базичних блокова. Оптимизације попут инлајновања, прераспоређивања блокова (енг. *block reordering*) и одмотавања петљи користе наменски предикат, који показује да ли се дати базични блок може сматрати често извршаваним, ретко извршаваним или се никада не извршава. Овај предикат се углавном користи како би се спречило агресивно оптимизовање кода који

се ретко извршава. Предикат се поставља на основу фреквенције извршавања базичних блокова на основу профила, када су они доступни. Вредност предиката се одређује на основу граничних вредности које фреквенција извршавања неког блока мора премашити да би се он сматрао често или ретко извршаваним. Ове граничне вредности могу бити подешаване.

Инфраструктура за прикупљање профила. У инфраструктури GCC, слично као и у окружењу *GraalVM Native Image*, профили се прикупљају на следећи начин [142]. Програм се најпре преводи са опцијом која омогућава инструментацију. Излаз покретања резултујућег извршног фајла је датотека која садржи инструментационе профиле, која се потом користи као улаз за друго превођење истог програма. Инструментациони бројачи се уграђују у међурепрезентацију програма, на основу чега се производи информација о фреквенцији извршавања делова кода попут броја позива функција, броја извршавања базичних блокова, броја извршавања грана графа тока контроле програма (а самим тим и одговарајућих базичних блокова), из чега се директно могу извести вероватноће извршавања грана контролних структура [141].

Оптимизација инлајновања. Инлајнер у оквиру окружења GCC се ослања на *bin pack* алгоритам. Он користи бројне метрике попут максималне величине јединице компајлирања, максималне величине кандидата за инлајновање, раста великих функција, итд. како би приоритизовао опције за инлајновање пре постизања граничне вредности [141]. Када су профили доступни, функције-кандидати за инлајновање се приоритизују на основу односа користи и цене унутар хеуристичке функције. Ова функција користи фреквенције извршавања делова кода из профила, као и раст функције у оквиру које се инлајновање дешава.

9.2 LLVM

Међурепрезентација. Међурепрезентација инфраструктуре LLVM [143] је строго типизирана троадресна репрезентација у SSA форми, која омогућава неструктурисани ток контроле и користи фи вредности (енг. *phi*) за спајање различитих путања. Трансформација међурепрезентације из преводиоца *Graal* у LLVM међурепрезентацију је праволинијска. Такође, све стандардне компајлерске оптимизације имплементиране за *Graal* у одговарајућој репрезентацији могу се имплементирати и у репрезентацији LLVM-а са приближном количином труда.

Детекција често извршаваних делова кода. Инлајнер се ослања на неколико параметара за одређивање граничне вредности, на основу које се доноси одлука да ли се део кода сматра често или ретко извршаваним. Када су укључени виши нивои оптимизација, оптимизације се ослањају на класификацију делова кода као често извршаваних. Оптимизација поделе кода на често и ретко извршаване делове кода (енг. *hot/cold-splitting optimization*) користи профиле грана како би класификовала базичне блокове тренутне јединице компилације као више или мање фреквентне. Ово је такође корисно за оптимизације попут инлајновања или издвајања (аутлајновања) функција. LLVM имплементира издвајања ретко извршаваних региона из често извршаваних делова кода [144].

Инфраструктура за прикупљање профила. Прикупљање профила је традиционално омогућено применом технике инструментације. Профили који се прикупљају и потом користе садрже информације о фреквенцији позивања функција, као и гранама контролних структура. Такође, профили садрже и информације о виртуелним позивима, тј. фреквенције типова објекта чије методе су заправо позване на датој локацији у коду. Овакав садржај профила је веома сличан информацијама које се прикупљају у окружењу *GraalVM Native Image*. Могуће је прикупити и контекстно неосетљиве профиле, као и профиле који садрже прецизнију локацију у коду [145].

Оптимизација инлајновања. Инлајнер користи хеуристику како би елиминисао мање фреквентне индиректне позиве функција у циљу усмеравања оптимизације према чешћим позивима [85, 146]. Конкретне методе које одговарају пријемницима позива, који су, на основу профила о виртуелним методама, најфреквентнији уграђују се као директни позиви метода уколико њихова фреквенција прелази постављену граничну вредност [147]. Информације о фреквенцији извршавања базичних блокова доступна је инлајнеру, који их користи како би дефинисао граничне вредности фреквенција позива на локацијама у коду у односу на почетак тела методе у којој се место позива налази или глобално у коду [148]. Анализа трошка и користи узима у обзир величину кандидата за инлајновање, као и величину функције у оквиру које се инлајновање одвија. Хеуристике се користе како би се израчунала уштеда у броју циклуса за сваку локацију позива, а такође и за ограничавање рекурзивног инлајновања. Значајна количина буџета за инлајновање је усмерена према агресивнијем и прецизнијем спровођењу оптимизације инлајновања често извршаваних делова кода.

9.3 Примењивост алгоритма *PRINC* у окружењима LLVM и GCC

У оквиру ове секције укратко су описани релевантни аспекти окружења за превођење LLVM и GCC. У наставку следи дискусија о примењивости алгоритма предложеног и имплементираног у овом раду на ова два система.

Предложени алгоритам за детекцију често извршаваних делова кода проналази фреквентне стазе трагова ширењем често извршаваних контекста позива из профила. У описаној имплементацији улаз алгоритма за детекцију често извршаваних делова кода је скуп профила на основу којих се одређује фреквенција базичних блокова и веровантноћа позива конкретних метода на локацијама индиректних позива. Прикупљени профили у окружењима GCC и LLVM садрже исти тип информација. Ова окружења подржавају инструментацију након фазе инлајновања, што за последицу има да су прикупљени профили делимично контекстно осетљиви [149]. Описани алгоритам детекције често извршаваних делова кода може бити примењен над профилима прикупљеним у системима GCC и LLVM како би се конструисале стазе трагове. Стазе потом могу бити коришћене да се одреде позиване функције које припадају често извршаваним контекстима позивања, а затим и за постављање предиката који говори о фреквентности одговарајућих базичних блокова.

Оба описана окружења користе детекцију често извршаваних региона кода у оквиру многобројних оптимизација. Они омогућавају превођење често извршаваног кода укључујући његову оптимизацију са модификованим буџетом. Оба преводиоца дефинишу граничне вредности на основу којих се учестаност извршавања делова кода сматра великом или не. Граничне вредности постављене у оквиру решења реализованог у овом раду могу бити прилагођене тако да одговарају преводиоцима GCC и LLVM.

Унапређивање хеуристика за инлајновање употребом информација о фреквенцији извршавања кода је тема од интереса у заједницама које учествују у развоју окружења LLVM и GCC [150]. Генерално, циљ је унапредити међупроцедуралне оптимизације (енг. *interprocedural optimizations*) коришћењем информација о фреквенцији извршавања делова кода, где структуре података стазе трагова могу бити од користи имајући у виду да повезују више потпрограма. Хеуристике које користе инлајнери у окружењима *GraalVM*, GCC и LLVM користе анализу цене и користи које узимају у обзир бројне метрике попут величине и раста компилационе јединице за израчунавање цене појединачног спровођења оптимизације. Како и GCC и LLVM наглашавају корист од употребе информације о фреквентности извршавања кода у циљу приоритизовања инлајновања значајних позива, предложена модификација постојећем инлајнеру, која подразумева повећање буџета за инлајновање таквих позива, може бити једноставно инкорпорирана. Буџет за спровођење оптимизације инлајновања се одређује на основу скупа параметара инлајнера, који су дефинисани самом хеуристиком инлајнера. Дакле, вредности параметара инлајнера морају бити наменски подешене у циљу остваривања најбољих перформанси у складу са остатком оптимизационе компајлерске инфраструктуре.

9.4 Поређење са релевантном литературом

У поглављу 2.6 дат је преглед литературе из области значајних за ово истраживање. Под овим се подразумевају стратегије прикупљања профила и класификација алата који их примењују, употреба делимично контекстно осетљивих профила, као и примена профила током превођења са нагласком на примени профила у оквиру оптимизације инлајновања. Ипак, поглавље 2 не садржи детаљне описе и конкретна поређења неких од најзначајних радова са овим радом имајући у виду да је пре њиховог упоређивања било потребно описати детаље предложеног алгоритма. У овом поглављу ће бити приказана претходно изостављена поређења.

Релевантан рад у области примене делимично контекстно осетљивих профила објавили су Serrano и Zhuang [40]. Они су описали прикупљање делимичне трасе извршавања употребом додатног наменског хардвера. Потом су такве профиле користили како би реконструисали прецизније контекстно осетљиве информације. У оквиру ове технике, делимичне трасе извршавања су спајане уколико садрже значајна преклапања. Резултати овог спајања су стабла која садрже делимично контекстно осетљиве информације (енг. *partial calling-context tree*, скр. PCCT). PCCT стабла су слична стазама трагова, описаним у овој тези (листинг 7.2).

Иако спајање мањих PCCT стабала у већа PCCT стабла представља сличан проблем оном описаном у овој тези, постоје четири кључне ставке, које разликују ова два рада:

1. У улазним профилима који се добијају и инфраструктури *Native Image PGO*, не постоји преклапање за исту позицију у активационом стаблу. Другим речима, један чвор активационог стабла увек је представљен тачно једним профилним улазом. Разлог лежи у томе што се профили у оквиру система *Native Image PGO* прикупљају применом интрументације унутар компилационих јединица, тако да свака јединица има јединствене инструментационе бројаче, а сваком чвору активационог стабла одговара тачно једна компилациона јединица. Док трасе извршавања прикупљене на случајан начин показују преклапање до неког нивоа, профилни подаци који се користе као улаз алгоритма из ове тезе не садрже никаква преклапања у активационом стаблу – уколико постоји профил који одговара контексту $\ell_1, \ldots, \ell_m, \ldots, \ell_n$, не постоји профил који се односи на контекст $\ell_m, \ldots, \ell_n, \ldots, \ell_p$.

2. Захваљујући разликама у улазним подацима, дизајн алгоритма се такође разликује. Како техника описана у раду аутора Serrano и Zhuang проналази преклапања између профила како би спојила два стабла која садрже парцијалне профиле, она не може бити примењена над улазним подацима који не садрже преклапања попут оних у систему *Native Image*. Уместо спајања међусобно преклапајућих информација, алгоритам *PRINC* екстендује парцијалне контексте одређивањем потенцијалних контекста позивања. Таква реконструкција подразумева и естимацију одговарајућих метрика, што цео проблем реконструкције чини доста захтевнијим.

3. Дужина контекста извршавања програма доступних у систему *Native Image* је типично мања у односу на дужину траса коришћених у техници коју су применили Serrano и Zhuang. Као што је показано у секцији 8.6, просечна дужина контекста позивања варира између три и четири оквира на стеку позива. Са друге стране, дужина траса у PCCT техници зависи од величине прозора прикупљања профила наменског хардвера и самим тим може да достигне значајно дуже путање. У раду се помињу трасе дуге и до 32 оквира. Као што је претходно објашњено, *Native Image* одсеца парцијалне контексте на границама јединице компилације, што често може бити услед немогућности инлајновања позива виртуелне методе у инструментационом извршном фајлу.

   Последице поседовања дужих контекста позивања су незанемарљиве – Serrano и Zhuang пријављују да за парцијалне трасе програма дужине 16 и више оквира, преко 80% реконструисаних контекста позивања има само једног позиваоца, док је за контексте дужине 32 и више тај број и већи и премашује 90%. Са друге стране, аутори су приметили да проценат јединствених позивалаца у случају краћих контекста позивања значајно мањи. Цела техника описана у овој тези заснива се на реконструкцији путања на основу кратких контекста позивања, што даље имплицира учесталу потребу за одређивањем вишеструких позивалаца и апроксимацијом одговарајућих метрика.

4. На крају, захваљујући агресивнијој примени поступка ширења контекста значајан део овог истраживања посвећен је апроксимацији фреквенција извршавања појединачних чворова реконструисане стазе трагова (другим речима парцијалног стабла

позива). Циљ је одредити потенцијалне позиваоце који у већој мери доприносе учестаности извршавања кода и помоћи инлајнеру касније са прецизнијим информацијама о фреквенцији извршавања кандидата за инлајновање. Прецизније, уводи се фактор умањења (дефинисан у једначини 6.4) сваки пут када се стаза трагова прошири навише, како би се адресирала чињеница да је фреквенција целокупног стабла трагова представља тежинску суму фреквенција позива преко сваког могућег позиваоца.

Møller и Veileborg [98] су у раду описали статичку анализу алгоритма за оптимизацију библиотеке JDK 8 Streams, чији је циљ да трансформише функционалне `Stream` изразе у еквивалентне ручно писане петље. Имплементирани систем ова два аутора најпре идентификује `Stream` операцију, а затим спроводи инлајновање њених функција. У раду се користи опсервација да је, обично, целокупна конструкција и извршавање низа `Stream` функција лоцирано у истој јединици компилације. Самим тим, инлајновање свих функција у оквиру те компилационе јединице и спровођење *escape* анализе, као и елиминација уписа и читања поједностављује `Stream` операцију и своди је на једноставну петљу. Møller и Veileborg демонстрирали су исправност резултата на једноставним `Stream` програмима. Циљ рада представљеног у овој тези је да побољша учинак оптимизације инлајновања независно од библиотечких функција коришћених у програму. Другим речима, у овом раду нису усвајане претпоставке нити је спроведена анализа специфична за одређени код. Као што је приказано у евалуацији у секцији 8.2 остварено је значајно побољшање перформанси и у бенчмарцима који су базирани на `Stream` функцијама. Ипак, претпоставка је да се интеграцијом измена описаним у раду аутора Møller и Veileborg могу постићи још значајнија побољшања.

10 Закључак

Време извршавања програма сматра се једним од основних показатеља перформанси програма и увек је од ултимативне важности да се оно што више смањи. Убрзавању програма доприносе бројне компајлерске оптимизације попут девиртуелизације полиморфних позива, одмотавања петљи, инлајновања функција и бројних других. Иако могу значајно допринети побољшању перформанси програма, оптимизације које спроводе агресивнију трансформацију кода, попут оптимизације инлајновања, могу узроковати и значајно увећање генерисаног кода. У системима у којима је меморија критичан ресурс ово може бити посебно неповољно, па се примена оптимизација мора вршити контролисано.

Значајан број компајлера користи информације о извршавању програма како би усмерио оптимизације на оне делове кода чије је извршавање значајно за перформансе целокупног програма. У оквиру оптимизација се, такође, користе ове информације како би се унапредиле одлуке које би, иначе, биле донете на основу статичких података. Компајлерима којима током превођења програма нису доступне информације о извршавању тог програма, оне се достављају у виду профила, који садрже информације о претходним извршавањима истог програма. У профилима се налазе метрички подаци, попут броја извршавања, који су повезани са локацијом у програму на коју се та метрика односи. У зависности од прецизности локације, која представља суфикс програмског стека, профили који их садрже могу бити контекстно осетљиви или неосетљиви. Како је прикупљање и чување потпуно контекстно осетљивих профила (који садрже цео стек позива метода које се извршавају у тренутку прикупљања профила) углавном неефикасно и скупо, развијено је више приступа да се смањи цена ових поступака. Један од приступа је и прикупљање делимично контекстно осетљивих профила, чиме се постиже компромис између ефикасности процеса и прецизности профила. Профили ипак морају садржати довољно програмског контекста како би могли бити успешно искоришћени у оквиру оптимизација.

Предмет истраживања је употреба профила ради детекције често извршаваних делова кода, измене распореда превођења и унапређења хеуристике оптимизације инлајновања функција. Циљ овог рада је пројектовање и имплементација унапређеног алгоритма за АОТ превођење програма, који користи делимично контекстно осетљиве профиле како би убрзао извршни код преведених програма без значајног увећања генерисаног кода и продужавања времена потребног за превођење.

У оквиру рада је развијен нови алгоритам за инлајновање и измену распореда превођења на основу делимично контекстно осетљивих профила – *PRINC*. Алгоритам је

имплементиран као саставни део АОТ преводиоца *GraalVM Native Image*. Делимично контекстно осетљиви профили добијени су у оквиру истог преводиоца извршавањем наменски преведеног програма са уграђеним инструментационим кодом, који прати извршавање програма. Техника инструментације омогућава прикупљање прецизних профила, али уз значајну цену, па је један од начина за постизање компромиса између прецизности и цене процеса профилисања управо прикупљање профила са краћим контекстом, тј. са мање информација о позицији профилисаног догађаја на програмском стеку. Уз то, поступак прикупљања парцијалних профила је веома ефикасан у конкретном окружењу захваљујући међурепрезентацији преводиоца *GraalVM Native Image*.

На основу улазних профила, алгоритам реконструише фрагменте стабла позива, који су у раду названи стазе трагова, како би издвојио секције кода које се често извршавају и омогућио агресивнију оптимизацију ових секција имајући у виду да је подједнака оптимизација целокупног кода неисплатива. Алгоритам за детекцију често извршаваних делова кода користи чињеницу да су метрике доступних профила, које садрже фреквенцију извршавања кода, иако повезане са некомплетном информацијом о позицији у програму, прецизне захваљујући техници прикупљања. Алгоритам опортунистички продужава парцијалне профиле и израчунава фреквенције извршавања таквих профила покушавајући да апроксимира вредности које би биле снимљене у случају прикупљања дужих контекста. Додатно, алгоритам омогућава груписање и филтрирање профила (оригинално прикупљених и апроксимираних) у циљу формирања већих целина кода, над којима потом оптимизације могу бити успешније извршене. Често извршавани код се потом преводи са већим буџетом за оптимизације у односу на остатак кода.

У конкретном преводиоцу распоред превођења није унапред дефинисан, већ се динамички одређује током самог превођења на основу претходно формираних јединица компилације. Како у конкретној конфигурацији преводиоца свака метода програма може бити компајлирана највише једном, уколико је нека метода алгоритмом означена као често извршавана, она треба да буде преведена са већим буџетом иако постоји неки контекст позивања из кога извршавање исте методе није значајно за време извршавања целокупног програма. Самим тим је кључно унапред одредити скуп често извршаваних јединица компилације, након чијег превођења и остатак програма може бити подразумевано обрађен. Алгоритам *PRINC* подразумева измену реда за распоређивање превођења тако да се на почетку распоређују корене методе стаза трагова.

Трећа главна компонента алгоритма подразумева агресивнију примену оптимизације инлајновања изменом буџета хеуристике оптимизације. Такође, одлуке оптимизације су модификоване тако да „фаворизују" инлајновање фрагмената често извршаваног кода на основу формираних стаза трагова. Ипак, како су стазе трагова формиране апроксимативним приступом, алгоритам инлајновања укључује сугестије компоненте за детекцију често извршаваног кода, али задржава и додатну логику за одлучивање о инлајновању позива на појединачним локацијама програма.

У истраживању је спроведена исцрпна евалуација над 16 бенчмарка из скупова DaCapo, Scalabench и Renaissance. Постигнута су убрзања од 2.5% до 40%, док су успорења до 3% уочена на само 2 бенчмарка. Величина генерисаног кода је увећана између 0.8% и 9%, а за већину бенчмарка увећање је мање од 2.5%. Иако утицај алгоритма на време

превођења није кључан пошто оно не утиче на перформансе извршавања програма, показано је да алгоритам успорава превођење програма између 2.8% и 10% за чак 12 бенчмарка, а највеће успорење износи 23%. Спроведен је и низ додатних експеримената који приказују утицај појединачних параметара алгоритма на перформансе програма, као и поступак проналажења најбољих вредности параметара. У оквиру квалитативне евалуације приказано је да алгоритам може бити имплементиран и у другим оптимизационим компајлерима уз адекватно прилагођавање конкретном окружењу. Полисе и хеуристике алгоритма су јасно издвојене управо ради олакшавања процеса прилагођавања целокупног аплгоритма другим окружењима, подешавањем параметара који их чине.

Осим развијеног алгоритма, резултат овог истраживања представљају и идеје за будуће унапређење процеса превођења. Алгоритам за побољшање превођења и оптимизације инлајновања може бити допуњен употребом профила прикупљених техником узорковања. Како профили добијени техником узорковања и техником инструментације представљају различите податке, једни описују време извршавања док други изражавају учестаност извршавања, њихова комбинација може бити корисна за унапређивање алгоритма за детекцију и оптимизацију често извршаваног кода. Профили прикупљени техником узорковања су потпуно контекстно осетљиви, па унапређени алгоритам мора узети у обзир да различити типови профила садрже и различиту количину информација о позицији у програмском коду. Друга линија унапређења пројекта је у домену омогућавања вишеструког превођења исте методе у зависности од контекста позивања, уколико су они означени као значајни за перформансе програма. Различите јединице компилације које одговарају истој методи омогућавају другачију примену оптимизационих пролаза за сваку јединицу. На тај начин би се омогућила прецизнија специјализација често извршаваних делова кода, која не би била зависна од редоследа превођења.

Литература

[1] John Aycock. A Brief History of Just-in-Time. *ACM Comput. Surv.*, 35(2):97–113, Jun 2003.

[2] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct 2017.

[3] Michael R. Jantz and Prasad A. Kulkarni. Performance Potential of Optimization Phase Selection During Dynamic JIT Compilation. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 131–142, New York, NY, USA, 2013. ACM.

[4] Urs Hölzle and David Ungar. Optimizing Dynamically-dispatched Calls with Run-time Type Feedback. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 326–336, New York, NY, USA, 1994. ACM.

[5] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Würthinger. An Optimization-driven Incremental Inline Substitution Algorithm for Just-in-time Compilers. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 164–179, Piscataway, NJ, USA, 2019. IEEE Press.

[6] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. Speculation without Regret: Reducing Deoptimization Meta-Data in the Graal Compiler. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 187–193, New York, NY, USA, 2014. Association for Computing Machinery.

[7] Josef Eisl, Matthias Grimmer, Doug Simon, Thomas Würthinger, and Hanspeter Mössenböck. Trace-Based Register Allocation in a JIT Compiler. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '16, New York, NY, USA, 2016. Association for Computing Machinery.

[8] Graham Yiu. Partial Inlining with multi-region outlining based on PGO information, 2017. `https://reviews.llvm.org/D38190` [Pristupano: 20.03.2022.].

[9] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-mei W. Hwu. Profile-guided Automatic Inline Expansion for C Programs. *Softw. Pract. Exper.*, 22(5):349–369, May 1992.

[10] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. A Comparative Study of Static and Profile-based Heuristics for Inlining. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, DYNAMO '00, pages 52–64, New York, NY, USA, 2000. ACM.

[11] J.C. Huang and T. 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, 1999.

[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] Litong Song and K.M. Kavi. A technique for variable dependence driven loop peeling. In *Fifth International Conference on Algorithms and Architectures for Parallel Processing, 2002. Proceedings.*, pages 390–395, 2002.

[14] 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. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[15] 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*, VMIL '13, pages 1–10, New York, NY, USA, 2013. Association for Computing Machinery.

[16] Google LLC. V8 Engine, 2021. `https://v8.dev/` [Pristupano: 29.06.2021.].

[17] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[18] 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*, CGO '10, pages 42–52, New York, NY, USA, 2010. Association for Computing Machinery.

[19] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online Feedback-Directed Optimization of Java. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 111–129, New York, NY, USA, 2002. Association for Computing Machinery.

[20] Aibek Sarimbekov, Philippe Moret, Walter Binder, Andreas Sewe, and Mira Mezini. Complete and Platform-Independent Calling Context Profiling for the Java Virtual Machine. *Electronic Notes in Theoretical Computer Science*, 279(1):61–74, 2011. Proceedings of the Bytecode 2011 workshop, the Sixth Workshop on Bytecode Semantics, Verification, Analysis and Transformation.

[21] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Syst. J.*, 39(1):211–238, Jan 2000.

[22] Olivier Flückiger, Andreas Wälchli, Sebastián Krynski, and Jan Vitek. Sampling Optimized Code for Type Feedback. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2020, pages 99–111, New York, NY, USA, 2020. Association for Computing Machinery.

[23] IBM Corporation. Profile-guided optimization (PGO) using GCC, 2021. `https://developer.ibm.com/articles/gcc-profile-guided-optimization-to-accelerate-aix-applications/` [Pristupano: (08.12.2021.)].

[24] Robert G. Burger and R. Kent Dybvig. An Infrastructure for Profile-Driven Dynamic Recompilation. ICCL '98, page 240, USA, 1998. IEEE Computer Society.

[25] C. Krintz. Coupling on-line and off-line profile information to improve program performance. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 69–78, 2003.

[26] Denys Shabalin and Martin Odersky. Interflow: Interprocedural Flow-Sensitive Type Inference and Method Duplication. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*, Scala 2018, pages 61–71, New York, NY, USA, 2018. Association for Computing Machinery.

[27] Evelyn Duesterwald and Vasanth Bala. Software Profiling for Hot Path Prediction: Less is More. *SIGARCH Comput. Archit. News*, 28(5):202–211, Nov 2000.

[28] Oracle Company. HotSpot Runtime Overview, 2021. `https://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html` [Pristupano: 29.12.2021.].

[29] 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*, PLDI 2017, pages 662–676, New York, NY, USA, 2017. ACM.

[30] Kevin Casey, David Gregg, M. Anton Ertl, and Andrew Nisbet. Towards Superinstructions for Java Interpreters. In Andreas Krall, editor, *Software and Compilers for Embedded Systems*, pages 329–343, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[31] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. *SIGPLAN Not.*, 32(10):318–326, October 1997.

[32] Martin Hirzel and Trishul Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, pages 117–126. Citeseer, 2001.

[33] John Whaley. A Portable Sampling-Based Profiler for Java Virtual Machines. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 78–87, New York, NY, USA, 2000. Association for Computing Machinery.

[34] J. Eugene Ball. Predicting the Effects of Optimization on a Procedure Body. In *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '79, pages 214–220, New York, NY, USA, 1979. ACM.

[35] Matthew Arnold and Barbara G. Ryder. A Framework for Reducing the Cost of Instrumented Code. pages 168–179, 2001.

[36] Michael D. Bond and Kathryn S. McKinley. Probabilistic Calling Context. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '07, pages 97–112, New York, NY, USA, 2007. Association for Computing Machinery.

[37] Glenn Ammons, Jong-Deok Choi, Manish Gupta, and Nikhil Swamy. Finding and removing performance bottlenecks in large systems. In *European Conference on Object-Oriented Programming*, pages 172–196. Springer, 2004.

[38] Steven P Reiss and Manos Renieris. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 221–230. IEEE, 2001.

[39] Giorgio Ausiello, Camil Demetrescu, Irene Finocchi, and Donatella Firmani. K-Calling Context Profiling. *SIGPLAN Not.*, 47(10):867–878, Oct 2012.

[40] Mauricio Serrano and Xiaotong Zhuang. Building Approximate Calling Context from Partial Call Traces. In *2009 International Symposium on Code Generation and Optimization*, pages 221–230, 2009.

[41] Maja Vukasovic and Aleksandar Prokopec. Exploiting Partially Context-Sensitive Profiles to Improve Performance of Hot Code. *ACM Transactions on Programming Languages and Systems*, Sep 2023. `https://doi.org/10.1145/3612937`.

[42] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. *SIGPLAN Not.*, 41(10):169–190, October 2006.

[43] 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 *OOPSLA*, pages 657–676, 2011.

[44] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tuma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: benchmarking suite for parallel applications on the JVM. In Kathryn S. McKinley and Kathleen Fisher, editors, *PLDI*, pages 31–47. ACM, 2019.

[45] LLVM Project. Llvm, 2012. `https://llvm.org/` [Pristupano: 12.12.2022.].

[46] Free Software Foundation. Using the GNU Compiler Collection, 2018. `https://gcc.gnu.org/onlinedocs/gcc-13.2.0/gcc.pdf` [Pristupano: 15.05.2023.].

[47] Scott Milton and Heinrich (Heinz) Schmidt. Dynamic Dispatch in Object-Oriented Languages. 03 1994. `https://openresearch-repository.anu.edu.au/handle/1885/40783`.

[48] Edward Fredkin. Trie Memory. *Commun. ACM*, 3(9):490–499, Sep 1960.

[49] B. G. Ryder. Constructing the call graph of a program. *IEEE Trans. Softw. Eng.*, 5(3):216–226, May 1979.

[50] Grove, David and Chambers, Craig. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, Nov 2001.

[51] Donald E. Knuth. An Empirical Study of FORTRAN Programs. *Softw. Pract. Exp.*, 1(2):105–133, 1971.

[52] Alan D Samples. Profile-driven compilation. Technical report, CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, 1991.

[53] Peter Hofer, David Gnedt, and Hanspeter Mössenböck. Lightweight Java profiling with partial safepoints and incremental stack tracing. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 75–86, 2015.

[54] Dries Buytaert, Andy Georges, Michael Hind, Matthew Arnold, Lieven Eeckhout, and Koen De Bosschere. Using Hpm-Sampling to Drive Dynamic Compilation. *SIGPLAN Not.*, 42(10):553–568, Oct 2007.

[55] Peter Hofer and Hanspeter Mössenböck. Fast Java Profiling with Scheduling-Aware Stack Fragment Sampling and Asynchronous Analysis. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 145–156, New York, NY, USA, 2014. Association for Computing Machinery.

[56] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Evaluating the accuracy of Java profilers. *ACM Sigplan Notices*, 45(6):187–197, 2010.

[57] Oracle Company. XPROF: Internal Hotspot Profiler, 2022. `https://docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html` [Pristupano: 20.03.2022.].

[58] Oracle Company. HPROF: A Heap/CPU Profiling Tool, 2022. `https://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html` [Pristupano: 20.03.2022.].

[59] EJ Technologies. Jprofiler, 2023. `https://www.ej-technologies.com/resources/jprofiler/help/doc/JProfiler.pdf` [Pristupano: 21.09.2023.].

[60] YourKit GmbH. YourKit, 2022. `https://www.yourkit.com/` [Pristupano: 20.03.2022.].

[61] Susan L Graham, Peter B Kessler, and Marshall K McKusick. Gprof: A call graph execution profiler. *ACM Sigplan Notices*, 17(6):120–126, 1982.

[62] M. Dmitriev. Selective profiling of Java applications using dynamic bytecode instrumentation. In *IEEE International Symposium on - ISPASS Performance Analysis of Systems and Software, 2004*, pages 141–150, 2004.

[63] Oracle Company. Netbeans: Open source Java profiler. v6.7., 2022. `https://web.archive.org/web/20210108060217/http://profiler.netbeans.org/` [Pristupano: 22.03.2022.].

[64] Eclipse Foundation. Eclipse Test and Performance Tool Platform, 2022. `http://archive.eclipse.org/archived_projects/tptp.tgz` [Pristupano: 20.03.2022.].

[65] Omri Traub, Stuart Schechter, and Michael D Smith. Ephemeral instrumentation for lightweight program profiling. *Unpublished technical report, Department of Electrical Engineering and Computer Science, Harvard University, Cambridge, Massachusetts*, 2000.

[66] 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*, LCTES 2017, pages 1–10, New York, NY, USA, 2017. Association for Computing Machinery.

[67] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.*, 42(6):89–100, Jun 2007.

[68] Oracle Company. Java Mission Control, 2023. `https://www.oracle.com/java/technologies/jdk-mission-control.html` [Pristupano: 12.01.2023.].

[69] James Gosling. Java Intermediate Bytecodes: ACM SIGPLAN Workshop on Intermediate Representations (IR'95). In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, pages 111–118, New York, NY, USA, 1995. Association for Computing Machinery.

[70] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2007.

[71] 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*, SCALA 2017, pages 29–40, New York, NY, USA, 2017. ACM.

[72] 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*, CGO 2018, pages 126–137, New York, NY, USA, 2018. ACM.

[73] Jack W. Davidson and Sanjay Jinturkar. Memory access coalescing: A technique for eliminating redundant memory accesses. *SIGPLAN Not.*, 29(6):186–195, Jun 1994.

[74] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant expressions. *SIGPLAN Not.*, 33(5):1–14, May 1998.

[75] 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*, SCALA '13, pages 9:1–9:8, New York, NY, USA, 2013. ACM.

[76] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. POPL '02, New York, NY, USA, 2002. Association for Computing Machinery.

[77] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive Inlining. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 134–145, New York, NY, USA, 1997. ACM.

[78] Robert W. Scheifler. An analysis of inline substitution for a structured programming language. *Commun. ACM*, 20(9):647–654, Sep 1977.

[79] David Detlefs and Ole Agesen. *Inlining of Virtual Methods*, pages 258–277. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[80] P. P. Chang and W.-W. Hwu. Inline Function Expansion for Compiling C Programs. *SIGPLAN Not.*, 24(7):246–257, Jun 1989.

[81] Manuel Serrano. *Inline expansion: When and how?*, pages 143–157. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[82] Keith D. Cooper, Mary W. Hall, and Linda Torczon. Unexpected Side Effects of Inline Substitution: A Case Study. *ACM Lett. Program. Lang. Syst.*, 1(1):22–32, Mar 1992.

[83] Peng Zhao and José Nelson Amaral. *To Inline or Not to Inline? Enhanced Inlining Decisions*, pages 405–419. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[84] Dhruva R. Chakrabarti and Shin-Ming Liu. Inline Analysis: Beyond Selection Heuristics. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 221–232, Washington, DC, USA, 2006. IEEE Computer Society.

[85] LLVM Project. LLVM Cost-Benefit Estimation Implementation at GitHub, 2023. `https://github.com/llvm-mirror/llvm/blob/88ab6705571782fa664ecfa71b2f959a0daf2d78/lib/Analysis/InlineCost.cpp` [Pristupano: 12.03.2023.].

[86] Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler Inliner. *J. Funct. Program.*, 12(5):393–434, Jul 2002.

[87] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeño JVM. *SIGPLAN Not.*, 35(10):47–65, Oct 2000.

[88] Matteo Basso, Aleksandar Prokopec, Andrea Rosà, and Walter Binder. Optimization-Aware Compiler-Level Event Profiling. *ACM Trans. Program. Lang. Syst.*, 45(2), Jun 2023.

[89] Thomas Ball and James R. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, USA, 1996. IEEE Computer Society.

[90] Diego Novillo. SamplePGO: The Power of Profile Guided Optimizations without the Usability Burden. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, LLVM-HPC '14, pages 22–28. IEEE Press, 2014.

[91] Matthew Arnold and Peter Sweeney. Approximating the calling context tree via sampling. Technical report, IBM, 2000.

[92] Roy Levin, Ilan Newman, and Gadi Haber. Complementing Missing and Inaccurate Profiling Using a Minimum Cost Circulation Algorithm. In *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'08, pages 291–304, Berlin, Heidelberg, 2008. Springer-Verlag.

[93] Cliff Click. Global Code Motion/Global Value Numbering. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 246–257, New York, NY, USA, 1995. ACM.

[94] Thomas Kistler and Michael Franz. Continuous Program Optimization: A Case Study. *ACM Trans. Program. Lang. Syst.*, 25(4):500–548, Jul 2003.

[95] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. An Empirical Study of Method Inlining for a Java Just-in-Time Compiler. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, pages 91–104, USA, 2002. USENIX Association.

[96] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-Guided Receiver Class Prediction. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '95, pages 108–123, New York, NY, USA, 1995. Association for Computing Machinery.

[97] Keith D. Cooper, Timothy J. Harvey, and Todd Waterman. An Adaptive Strategy for Inline Substitution. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, pages 69–84, Berlin, Heidelberg, 2008. Springer-Verlag.

[98] Anders Møller and Oskar Haarklou Veileborg. Eliminating Abstraction Overhead of Java Stream Pipelines Using Ahead-of-Time Program Optimization. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[99] Oracle Company. GraalVM, 2023. `https://www.graalvm.org/` [Pristupano: 24.05.2023.].

[100] Oracle Company. Graal compiler, 2023. `https://github.com/oracle/graal/tree/master/compiler` [Pristupano: 24.05.2023.].

[101] Oracle Company. Native Image, 2023. `https://www.graalvm.org/22.0/reference-manual/native-image/` [Pristupano: 24.05.2023.].

[102] Oracle Company. Truffle framework, 2023. `https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/` [Pristupano: 24.05.2023.].

[103] Yoshihiko Futamura. Partial computation of programs. In Eiichi Goto, Koichi Furukawa, Reiji Nakajima, Ikuo Nakata, and Akinori Yonezawa, editors, *RIMS Symposia on Software Science and Engineering*, pages 1–35, Berlin, Heidelberg, 1983. Springer Berlin Heidelberg.

[104] Oracle Company. Java Virtual Machine Specification (Java SE 8 Edition): Chapter 4, the Class File Format. `https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html` [Pristupano: 06.12.2021.].

[105] Codruţ Stancu, Christian Wimmer, Stefan Brunthaler, Per Larsen, and Michael Franz. Comparing Points-to Static Analysis with Runtime Recorded Profiling Data. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 157–168, New York, NY, USA, 2014. Association for Computing Machinery.

[106] 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*, VMIL '13, pages 1–10, New York, NY, USA, 2013. ACM.

[107] Cliff Click and Michael Paleczny. A Simple Graph-based Intermediate Representation. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, pages 35–49, New York, NY, USA, 1995. ACM.

[108] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. Association for Computing Machinery.

[109] Oracle Company. OpenJDK 8 Optional Class, 2021. `https://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/Optional.java#l120` [Pristupano:29.06.2021.].

[110] Rodric M. Rabbah, Hariharan Sandanagobalane, Mongkol Ekpanyapong, and Weng-Fai Wong. Compiler Orchestrated Prefetching via Speculation and Predication. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 189–198, New York, NY, USA, 2004. Association for Computing Machinery.

[111] LLVM Project. LLVM profile-guided optimizations, 2021. `https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization` [Pristupano: 10.12.2021.].

[112] Thomas Kotzmann and Hanspeter Mossenbock. Run-Time Support for Optimizations Based on Escape Analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 49–60, Washington, DC, USA, 2007. IEEE Computer Society.

[113] 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*, CGO '14, pages 165:165–165:174, New York, NY, USA, 2014. ACM.

[114] Rajkishore Barik and Vivek Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 41–52, USA, 2009. IEEE Computer Society.

[115] Anatole Le, Ondřej Lhoták, and Laurie Hendren. Using inter-procedural side-effect information in jit optimizations. In Rastislav Bodik, editor, *Compiler Construction*, pages 287–304, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[116] Mark N. Wegman and F. Kenneth Zadeck. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, April 1991.

[117] Jeffrey Dean and Craig Chambers. Towards Better Inlining Decisions Using Inlining Trials. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, LFP '94, pages 273–282, New York, NY, USA, 1994. ACM.

[118] Suresh Jagannathan and Andrew Wright. Flow-directed Inlining. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 193–205, New York, NY, USA, 1996. ACM.

[119] Andreas Sewe, Jannik Jochem, and Mira Mezini. Next in Line, Please!: Exploiting the Indirect Benefits of Inlining by Accurately Predicting Further Inlining. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 317–328, New York, NY, USA, 2011. ACM.

[120] Edwin Steiner, Andreas Krall, and Christian Thalinger. Adaptive Inlining and On-stack Replacement in the CACAO Virtual Machine. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, PPPJ '07, pages 221–226, New York, NY, USA, 2007. ACM.

[121] Denys Shabalin. *Just-in-time performance without warm-up.* PhD thesis, EPFL, Lausanne, 2020.

[122] Yosi Ben Asher, Omer Boehm, Daniel Citron, Gadi Haber, Moshe Klausner, Roy Levin, and Yousef Shajrawi. *Aggressive Function Inlining: Preventing Loop Blockings in the Instruction Cache*, pages 384–397. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[123] 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*, VMIL 2022, pages 13–24, New York, NY, USA, 2022. Association for Computing Machinery.

[124] Oracle Company. Control-flow-graph analysis in the Graal codebase, 2021. https://github.com/oracle/graal/blob/5708f348ad6a49511f0e3caf5314d72ca8c017e7/compiler/src/org.graalvm.compiler.nodes/src/org/graalvm/compiler/nodes/cfg/ControlFlowGraph.java [Pristupano: 07.12.2021.].

[125] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Visualization of program dependence graphs. In Laurie Hendren, editor, *Compiler Construction*, pages 193–196, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-78791-4_13.

[126] John Rose. JVM implementation challenges: Why the future is hard but worth it, 2015. https://www.jfokus.se/jfokus15/preso/JVMChallenges.pdf [Pristupano: 04.02.2022.].

[127] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. Association for Computing Machinery.

[128] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: A Benchmark for Software Transactional Memory. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 315–324, New York, NY, USA, 2007. ACM.

[129] George B. Dantzig. *Origins of the Simplex Method*, pages 141–151. Association for Computing Machinery, New York, NY, USA, 1990. https://doi.org/10.1145/87252.88081.

[130] George B Dantzig, Alex Orden, and Philip Wolfe. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1955.

[131] Doug Simon, Christian Wimmer, Bernhard Urban, Gilles Duboscq, Lukas Stadler, and Thomas Würthinger. Snippets: Taking the High Road to a Low Level. *ACM Trans. Archit. Code Optim.*, 12(2), June 2015.

[132] Oracle Company. Parallel Garbage Collector, 2021. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html [Pristupano:20.04.2021.].

[133] Oracle Company. Serial Native Image Garbage Collector, 2021. https://www.graalvm.org/reference-manual/native-image/MemoryManagement/#serial-garbage-collector [Pristupano: 20.04.2021.].

[134] Stephen J. Fink and Feng Qian. Design, Implementation and Evaluation of Adaptive Recompilation with on-Stack Replacement. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '03, pages 241–252, USA, 2003. IEEE Computer Society.

[135] Oracle Company. Speculative guard motion in GraalVM, 2022. `https://github.com/oracle/graal/commit/6dcc8e4a57d23e7aaf85eeb8dae7ef501b59c18b#diff-1e4c4d8dd65775bb5c116be6e862315ddebf9b0d84aec949a79af159ef899df4` [Pristupano: 27.05.2022.].

[136] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.

[137] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[138] Free Software Foundation. GCC GENERIC, 2018. `https://gcc.gnu.org/onlinedocs/gccint/GENERIC.html` [Pristupano: 09.05.2023.].

[139] Free Software Foundation. GCC GIMPLE, 2018. `https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html` [Pristupano: 09.05.2023.].

[140] Free Software Foundation. GCC RTL, 2018. `https://gcc.gnu.org/onlinedocs/gccint/RTL.html` [Pristupano: 09.05.2023.].

[141] Jan Hubicka. Profile driven optimisations in GCC. In *GCC Summit Proceedings*, pages 107–124. Citeseer, 2005.

[142] Martin Jambor, Jan Hubicka, Richard Biener, Martin Liska, Michael Matz, and Brent Hollingsworth. PGO in GCC 11, 2022. `https://documentation.suse.com/sbp/server-linux/single-html/SBP-GCC-11/index.html#sec-gcc11-pgo` [Pristupano: 20.05.2022.].

[143] LLVM Project. LLVM Language Reference Manual, 2018. `https://llvm.org/docs/LangRef.html` [Pristupano: 20.01.2023.].

[144] Aditya Kumar. Hot Cold Splitting Optimization Pass In LLVM, 2019. `https://llvm.org/devmtg/2019-10/talk-abstracts.html#tech8` [Pristupano: 10.05.2023.].

[145] Pavel Kosov and Sergey Yakushkin. LLVM PGO Instrumentation: Example of CallSite-Aware Profiling, 2020. `https://llvm.org/devmtg/2020-09/slides/PGO_Instrumentation.pdf` [Pristupano: 20.05.2023.].

[146] LLVM Project. LLVM Inliner Implementation at GitHub, 2018. https://github.com/llvm-mirror/llvm/blob/6f1d64eb934e12ca5e8dcd378f88d1e6b80e8c55/lib/Transforms/IPO/Inliner.cpp [Pristupano: 26.05.2023.].

[147] Ivan Baev. Profile-based Indirect Call Promotion, 2015. https://llvm.org/devmtg/2015-10/#talk3 [Pristupano: 10.05.2023.].

[148] LLVM Project. LLVM Inlining Parameters, 2018. https://llvm.org/doxygen/structllvm_1_1InlineParams.html [Pristupano: 20.05.2023.].

[149] LLVM Project. LLVM PGO Context Sensitivity, 2023. https://reviews.llvm.org/D54175 [Pristupano: 12.06.2023.].

[150] Free Software Foundation. GCC 8 Changes, 2018. https://gcc.gnu.org/gcc-8/changes.html [Pristupano: 10.05.2023.].

Биографија аутора

Маја Вукасовић је рођена 25.05.1993. године у Београду. Основну школу завршила је у Београду, као ђак генерације и носилац дипломе „Вук Караџић". Након тога завршава Девету гимназију „Михаило Петровић Алас", такође у Београду као носилац дипломе „Вук Караџић". Током школовања, освајала је награде на такмичењима из математике, физике, српског и енглеског језика.

На Електротехнички факултет у Београду уписала се 2012. године на Одсеку за софтверско инжењерство. Дипломирала је 2016. године са просечном оценом 9.98. Дипломски рад под називом „Напредни генератор програмског кода за машине стања на језику UML" одбранила је са оценом 10. Након основних студија уписала је мастер академске студије на Електротехнички факултет у Београду 2016. године. Мастер рад под називом „Симулатор векторског процесора са предикатским извршавањем у више трака" одбранила је 2017. године са оценом 10. Просечна оцена након завршених мастер академских студија је 10.00.

Докторске студије уписала је 2017. године на Електротехничком факултету у Београду на модулу Софтверско инжењерство. Положила је све испите са оценом 10 и остварила 120 ЕСПБ. У истраживачком раду усмерила се ка програмским преводиоцима, са посебним нагласком на оптимизације током превођења програма. Објавила је један рад у часопису са SCI листе, 10 радова на страним и домаћим конференцијама као и један рад у домаћем часопису. Учествовала је на међународном пројекту ISSES, четири пројекта Министарства просвете, науке и технолошког развоја, као и пројекту Иновационог фонда „CoCos.ai".

Током студија радила је на праксама у фирмама „SOL" и „Kudos" и била укључена на неколико пројеката компаније „Nordeus". Школске 2013/2014. године прикључила се демонстраторском тиму на Катедри за рачунарску технику и информатику, а нешто касније и на Катедри за примењену математику. На Катедри за рачунарску технику и информатику је запослена од 2016. године, најпре као сарадник у настави, а потом као асистент. Тренутно је ангажована на шест предмета који се изводе на студијским програмима Софтверско инжењерство и Рачунарска техника и информатика. Као асистент на Електротехничком факултету била је ангажована на 11 предмета. Рецензирала је радове на конференцијама ТЕЛФОР и ЕТРАН. Од 2018. године ангажована је као истраживач у фирми „Oracle Labs". Коаутор је за три наставна материјала из области заштите података.

Прилог 1.

Изјава о ауторству

Потписани-а Маја Вукасовић

број индекса 2017/5010

Изјављујем

да је докторска дисертација под насловом

Побољшање перформанси програма употребом
делимично контекстно осетљивих профила

- 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, 22.09.2023.

М. Вукасовић

Прилог 2.

Изјава о истоветности штампане и електронске верзије докторског рада

Име и презиме аутора Маја Вукасовић

Број индекса 2017/5010

Студијски програм Софтверско инжењерство

Наслов рада Побољшање перформанси програма употребом делимично контекстно осетљивих профила

Ментор Др Драган Бојић, редовни професор

Потписани/а Маја Вукасовић

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, 22.09.2023

М. Вукасовић

Прилог 3.

Изјава о коришћењу

Овлашћујем Универзитетску библиотеку „Светозар Марковић" да у Дигитални репозиторијум Универзитета у Београду унесе моју докторску дисертацију под насловом:

Побољшање перформанси програма употребом делимично контекстно осетљивих профила

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, 22.09.2023.

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.