УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Милоје Ђ. Јоксимовић

КОМПАРАТИВНА АНАЛИЗА И ЕВАЛУАЦИЈА АЛГОРИТАМА РАСПОРЕЂИВАЊА МЕТОДА ПРОГРАМА У ЦИЉУ ОПТИМИЗАЦИЈЕ ПЕРФОРМАНСИ

мастер рад

Београд, 2025.

Ментор:

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

Чланови комисије:

др Мирко Спасић, доцент
Универзитет у Београду, Математички факултет

Марко Спасић, асистент
Универзитет у Београду, Математички факултет

Датум одбране: октобар, 2025.

Majụu, ouụy

Наслов мастер рада: Компаративна анализа и евалуација алгоритама распоређивања метода програма у циљу оптимизације перформанси

Резиме: Рад испитује утицај распореда метода у извршивој датотеци на перформансе апликација. У анализи се пореде референтни алгоритми (који насумично распоређују методе) са хеуристичким алгоритмима и алгоритмима заснованим на профилу програма. За генерисање различитих распореда метода у апликацијама користи се компајлер *GraalVM Native Image*. У оквиру рада развијена је и инфраструктура за анализу меморијског отиска кодних секција, тј. количине RAM меморије коју процес заузима током извршавања. Резултати анализе показују да алгоритми насумичног распоређивања метода доводе до лоших перформанси услед повећаног броја страничних промашаја. Са друге стране, једноставне хеуристике постижу значајна побољшања, до 25 процената у смањењу меморијског отиска кодне секције. Приступи засновани на профилима извршавања програма дају најбоље резултате, посебно у смањењу меморијског отиска кодне секције, где су побољшања и до 55 процената, при чему се алгоритам који се заснива на времену првог позива методе посебно истиче у смањењу меморијског отиска и времена покретања. Ови налази указују на значај алгоритама вођених профилом програма. Они доприносе унапређењу ефикасности извршивих датотека апликација, посебно у погледу величине меморијског отиска и времена првог одзива. Поред тога, анализирани алгоритми су побољшани тако да постижу боље перформансе или им је поједностављена имплементација.

Кључне речи: распоређивање метода у извршивој датотеци, оптимизација перформанси, профајлирање, *GraalVM*, време првог одзива, величина резидентног скупа

Садржај

Глава 1

Увод

Распоред метода у извршивој датотеци утиче на перформансе извршавања програма, што је посебно важно за апликације са строгим временским и просторним ограничењима. Ако се методе распоређују без разматрања њихове повезаности, то може довести до повећања броја промашаја страница (енгл. *page faults*) и повећаног коришћења меморије. Да би се избегли ови проблеми, компајлери и пост-линк оптимизатори уређују редослед метода на начин који побољшава њихову просторну локалност (енгл. *space locality*).

Савремени компајлери користе разне хеуристике и оптимизације вођене профилом програма (енгл. *profile guided optimizations*, скраћено *PGO*) како би одредили распоред метода у извршивој датотеци. Профил представља скуп података који описује понашање програма током његовог извршавања.

У овом раду анализирају се и пореде различити алгоритми за распоређивање метода у извршивој датотеци. Можемо их поделити на: алгоритме засноване на информацијама из профила извршавања, алгоритме вођене хеуристикама и алгоритме који врше насумични избор распореда. Циљ рада је да се стекне увид у утицај ових алгоритама на употребу меморије и перформансе програма.

Истраживање се спроводи у оквиру система *GraalVM* који, између осталог, омогућава аутоматизовано мерење метрика перформанси програма. У сврху поређења алгоритама распоређивања метода користи се референтни скуп програма који се иначе користи и за мерење перформанси у оквиру система *GraalVM*.

У оквиру тезе се измерене вредности метрика перформанси референтних програма за сваки алгоритам распоређивања метода који рад разматра додатно обрађују, визуелизују и анализирају. Допринос рада се огледа у анализи и

поређењу постојећих алгоритама, поједностављењем њихове имплементације и проширењу инфраструктуре за мерење прецизнијим мерењем заузећа меморије појединачних секција програма, специфично секције кода. Мерење меморијског отиска кодних секција представља допринос у области евалуације перформанси компајлерских оптимизација и отвара нове могућности за будућа истраживања у домену оптимизације распореда метода.

Рад је организован на следећи начин. Поглавље 2 описује систем *GraalVM* и његове фазе компајлирања с посебним освртом на делове који се тичу алгоритама распоређивања метода. Поглавље 3 описује приступе за распоређивање метода и алгоритме који су имплементирани у оквиру система *GraalVM*. Поглавље 4 најпре објашњава посматране метрике и њихов значај, сумира резултате и објашњава у којим окружењима су мерене претходно описане метрике. У наставку поглавља представљени су и анализирани резултати свих експеримената, укључујући поређење алгоритама по различитим метрикама. Такође, представљене су и имплементације побољшања алагоритама. На крају, поглавље 5 представља закључак рада.

Глава 2

Систем *GraalVM*

GraalVM [12] представља компајлерску инфраструктуру и извршно окружење високих перформанси. Настала је као проширење *Java* развојног комплета (енгл. *Java Develepoment Kit, JDK*). Основна идеја система је била да омогући заједничко окружење за извршавање програма написаних у различитим програмским језицима, попут *Java*, *JavaScript*, *Python*, *Ruby* и *Scala*, при чему сви језици користе исте механизме оптимизације.

Систем нуди више напредних могућности: компајлер *Graal JIT* (енгл. *Just-In-Time, JIT*) који омогућава JIT компајлирање (компајлирањем у фази извршавања ради постизања бољих перформанси), компајлер *GraalVM Native Image* који омогућава превођење апликација у самосталне извршиве датотеке, као и радни оквир *Truffle* који омогућава изградњу интерпретера за различите језике и подршку за полиглотско програмирање. Велики део система је отвореног кода (енгл. *open-source*) и доступан под називом *GraalVM Community Edition*.

2.1 Компајлер *GraalVM Native Image*

У оквиру система *GraalVM*, кључну компоненту представља сам компајлер *Graal*, који је у потпуности написан у програмском језику Јава и представља савремену алтернативу класичним *JIT* компајлерима као што је *C2*[1] унутар *HotSpot JVM*-а [11]. У оквиру компајлера *Graal*, програм се представља помоћу графовске међурепрезентације (енгл. *Graal Intermediate Representation – IR*). Ова међурепрезентација програма омогућава фино гранулисану анализу и оптимизацију, укључујући напредне технике као што су уметање метода (енгл. *in-*

[1]Компајлер *C2* је имплементиран у програмском језику *C++*.

lining), елиминација виртуелних позива (енгл. *devirtualization*), и оптимизације засноване на профилу програма.

Компајлирање пре времена извршавања (енгл. *Ahead Of Time, AOT*) подразумева процес у коме се пре самог извршавања бајткод програма преводи у машински кôд, чиме се добија извршива датотека која се назива и основна извршива датотека (енгл. *native executable*) [17]. Извршива датотека не захтева присуство окружења за извршавање Јава апликација (енгл. *Java Runtime Environment*) на машини на којој се покреће и прилагођена је циљној архитектури. На пример, на оперативном систему *Linux* користи се формат *ELF (Executable and Linkable Format)*, док се на оперативном систему *Windows* користи формат *EXE (Executable)*. Коришћење извршивих датотека је веома важно у контексту окружења у облаку (енгл. *cloud environments*), где је битна брзина покретања и мала потрошња меморије.

Да би извршавање без присуства Јава виртулне машине било могуће, у извршиву датотеку се уграђује мини виртуелна машина под називом *SubstrateVM*. Ова мини виртуелна машина садржи све неопходне делове за основно функционисање Java апликације: управљање нитима, сакупљање отпадака, обраду изузетака, руковање меморијом, као и имплементацију кључних делова Java стандардне библиотеке. На тај начин, основна извршива датотека представља потпуно самосталну јединицу за извршавање.

Компајлер *GraalVM Native Image* преводи бајткод у графовске репрезентације различитог нивоа апстракције. Оне укључују: висок ниво (енгл. *high-tier graph IR*), средњи ниво (енгл. *mid-tier graph IR*) и ниски ниво (енгл. *low-tier graph IR*). Ове репрезентације омогућавају фино гранулисану анализу и оптимизације програма. Након тога, репрезентација ниског нивоа се трансформише у линеаризовану листу инструкција ниског нивоа (енгл. *low-level intermediate representation, LIR)*, погодну за оптимизације ближе машинском нивоу. Коначно, из линеаризоване листе инструкција се генерише асемблерски кôд који се затим преводи у извршиви машински кôд.

Главна разлика између класичне Јава виртуелне машине и система *GraalVM* у *AOT* режиму је у начину на који се бајткод претвара у машински кôд. У традиционалном *JVM* моделу, извршавање почиње интерпретацијом бајткода, а најучесталије коришћени методи се током рада компајлирају у машински кôд коришћењем компајлера *JIT*. Овакав приступ омогућава прилагодљиву оптимизацију засновану на профилу програма који се прикупља у току извршавања,

али подразумева и извесне трошкове током рада, као што су загревање апликације (енгл. *warm-up time*), повећани меморијски отисак и потреба за присуством окружења *JVM*.

Приступ компајлера *GraalVM Native Image* је да компајлира комплетну апликацију унапред, чиме се умањује време покретања и смањује меморијски отисак. Међутим, овај приступ подразумева и неке ограничења, на пример, рефлексија и динамичко учитавање класа морају бити изричито обележени и конфигурисани пре компајлирања, што захтева додатне анализе и алате [10].

У погледу перформанси, *AOT* режим може бити нарочито ефикасан за апликације којима је кључна брзина покретања и предвидљив образац коришћења меморије током времена. Примери таквих апликација су командолинијски алати (енгл. *command line tools*), микросервиси (енгл. *microservices*) и контејнеризоване апликације (енгл. *containerized applications*). Са друге стране, класична *JVM* и даље пружа боље перформансе у апликацијама са дугим временом рада, захваљујући својој способности да се у току извршавања прилагоди стварном понашању програма кроз напредно профајлирање.

Извршива датотека и распоређивање метода

Да би се AOT компајлирањем генерисала оптимизована извршива датотека, *GraalVM Native Image* примењује модел затвореног света (енгл. *closed-world assumption*). Овај модел подразумева да су све класе, методе и поља који могу бити употребљени током извршавања апликације познати у фази компајлирања. Као последица тога, учитавање нових класа у време извршавања није дозвољено, што омогућава оптимизације које нису могуће приликом претпоставке отвореног света. Неке од додатно омогућених оптимизација су елиминација непотребних зависности (нпр. међу класама, библиотекама и модулима) и статичка иницијализација која омогућава да се сви статички блокови и променљиве иницијализују унапред.

Један од кључних корака у процесу AOT компајлирања је анализа достижности (енгл. *reachability analysis*), чији је циљ да одреди који делови програма ће заиста бити потребни током извршавања. Ова анализа се састоји из два основна дела:

- анализа бајткода сваке методе, којом се идентификују друге методе, класе и поља који се из те методе могу достићи,

- анализа статички иницијализованих објеката, којом се утврђују класе које се могу достићи преко иницијалних вредности поља и статичких конструктора.

Анализа достижности (енгл. *reachability analysis*) полази од свих могућих улазних тачака у програм од којих је једна и метода `main`. Улазне тачке могу бити и друге методе које су експлицитно дефинисане или се могу налазити у оквиру неке инфраструктуре као што је оквир за веб развој или систем за учитавање сервиса. Затим се итеративно проширује скуп достижних елемената тако што се за сваки већ идентификован метод или класу додају све њихове директне зависности (позвани методи, коришћене класе, иницијализована поља и ресурси). Процес траје све док се не утврди затворени скуп достижних елемената. Само достижни елементи се уграђују у коначну, извршиву датотеку.

Компајлер *GraalVM Native Image*, након анализе достижности, компајлира достижне методе и одређује њихов распоред у извршивој датотеци, чиме дефинише редослед и физичку позицију сваког метода. Распоред метода се одређује на основу изабраног алторитма, који може бити заснован, на пример, на анализи учесталости позива (енгл. *hotness*) или структурној повезаности (нпр. граф позива). Сви алгоритми су описани у оквиру следећег поглавља. Циљ распоређивања је да се методе које се често позивају заједно налазе физички близу једна другој у меморији, чиме се се смањује број промашаја страница и убрзава покретање и рад програма.

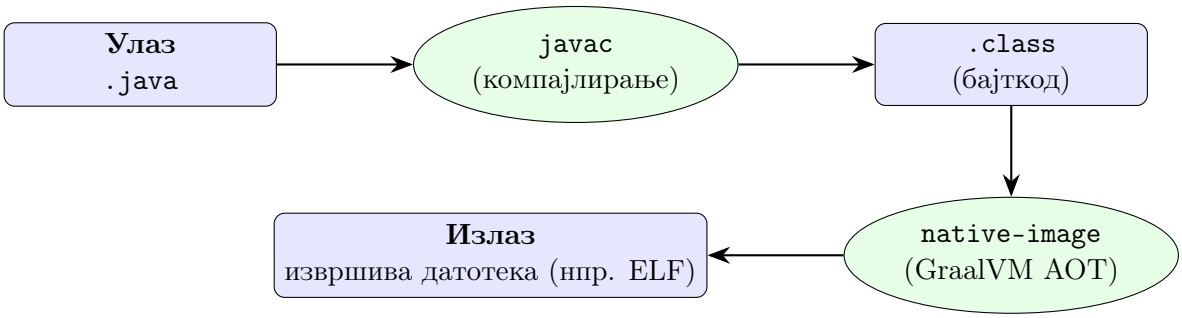
На крају процеса креирања извршиве датотеке, компајлер *GraalVM Native Image* смешта машински кôд метода у физичку секцију кôда на диску, према редоследу добијеним алгоритмом распоређивања метода.

Фазе компајлирања

АОТ компајлирање, илустровано дијаграмом на слици 2.1, подразумева превођење целокупног Јава изворног кода у извршиву датотетку.

Да би се направила самостална извршива датотека од Јава апликације, изворни кôд Јава апликације се прво преведе компајлером *javac* [14] у Јава бајткод. Затим, бајткод служи као улаз у компајлер *GraalVM Native Image* који га трансформише у машински кôд.

У листингу 2.1 приказан је стандардни излаз приликом креирања извршиве



Слика 2.1: Процес компајлирања од Јава изворног кода до извршиве датотеке

датотеке референтног програма који се назива *petclinic*[2] [16]. Овај излаз илуструје све фазе компилације, које су у наставку описане.

```
[1/8] Initializing...                                (8.8s @ 0.34GB)
 Java version: 26+3, vendor version: Oracle GraalVM 26-dev+3.1
 Graal compiler: optimization level: 3, target machine: x86-64-v3,
    PGO: user-provided
 C compiler: gcc (linux, x86_64, 11.4.0)
 Garbage collector: Serial GC (max heap size: 80% of RAM)
 3 user-specific feature(s):
 - com.oracle.svm.thirdparty.gson.GsonFeature
 - org.eclipse.angus.activation.nativeimage.AngusActivationFeature
 - org.springframework.aot.nativex.feature.PreComputeFieldFeature
------------------------------------------------------------------------
 3 experimental option(s) unlocked:
 - '-H:CodeSectionLayoutOptimization' (origin(s): command line)
 - '-H:+PGOIgnoreVersionCheck' (origin(s): command line)
 - '-H:+ProfileMethodTimestamps' (origin(s): command line)
------------------------------------------------------------------------
Build resources:
 - 16.73GB of memory (50.6% of system memory, using available
    memory)
 - 23 thread(s) (104.5% of 22 available processor(s), determined at
     start)
[2/8] Performing analysis...   [******]         (50.7s @ 6.18GB)
    37,989 types, 58,531 fields, and 197,595 methods reachable
    14,099 types, 6,311 fields, and  29,219 methods for reflection
    75 types, 75 fields, and 60 methods registered for JNI access
    0 downcalls and 0 upcalls registered for foreign access
    4 native libraries: dl, pthread, rt, z
[3/8] Building universe...                           (16.0s @ 4.64GB)
```

[2]Овај програм се у оквиру система *GraalVM* обично користи за тестирање перформанси.

```
[4/8] Parsing methods...      [***]             (5.6s @ 5.36GB)
[5/8] Inlining methods...     [****]            (2.5s @ 5.48GB)
[6/8] Compiling methods...    [********]        (63.4s @ 6.61GB)
[7/8] Laying out methods...   [****]            (17.0s @ 7.73GB)
/tmp/bundleRoot-7833962631/output/SortByMethodName/reports/code-
   layout-SortByFirstCall_20250714-154820.json
[8/8] Creating image...       [******] (38.7s @ 6.16GB)
  56.73MB (19.94%) for code area:   132,601 compilation units
  72.88MB (25.62%) for image heap:  765,659 objects and 1,277
     resources
 126.44MB (44.45%) for debug info generated in 8.9s
 154.84MB (54.44%) for other data
 284.45MB in total image size, 137.76MB in total file size
------------------------------------------------------------------------
Top 5 origins of code area:    Top 5 object types in image heap:
   9.28MB java.base             18.26MB byte[] for code metadata
   7.73MB hibernate.Final.jar   15.55MB byte[] for embedded
   5.58MB svm.jar(Native Image) 11.14MB byte[] for java.lang.String
   4.00MB h2-2.1.214.jar         5.42MB java.lang.String
   2.86MB tomcat-embed-core.jar 5.36MB java.lang.Class
   1.80MB java.xml               3.95MB core.hub.DynamicHubCompanion
Use '--emit build-report' to create a report with more details.
------------------------------------------------------------------------
Security report:
 - Binary includes Java deserialization.
 - CycloneDX SBOM with 99 component(s) is embedded in binary (8.88
    kB). 785 type(s) could not be associated to a component.
------------------------------------------------------------------------
Recommendations:
 G1GC: Use the G1 GC ('--gc=G1') for improved latency and
    throughput.
 FUTR: Use '--future-SortByMethodNames=all' to prepare for future
    releases.
 AWT:  Use the tracing agent to collect metadata for AWT.
 HEAP: Set max heap for improved and more predictable memory usage.
 CPU:  Enable more CPU features with '-march=native' for improved
    performance.
------------------------------------------------------------------------
 20.2s (9.5% of total time) in 1082 GCs | Peak RSS: 10.23GB | CPU
    load: 12.12
------------------------------------------------------------------------
Build artifacts:
```

```
 /tmp/bundleRoot-7833962631546735237/output/SortByMethodName/
    petclinic-wrk-3-0-1-mixed-tiny-pgo-ee (executable)
 /tmp/bundleRoot-7833962631546735237/output/SortByMethodName/
    petclinic-wrk-3-0-1-mixed-tiny-pgo-ee.debug (debug_info)
========================================================================
Finished generating 'petclinic-wrk-3-0-1-mixed-tiny-pgo-ee' in 3m
   32s.
```

Листинг 2.1: Фазе компајлирања референтног програма *Petclinic*

Иницијализација (*Initializing* у листингу 2.1) У овој фази компајлер *GraalVM Native Image* припрема интерне параметре окружења, учитава конфигурацију и иницијализује потребне алате. Парсирају се аргументи командне линије на основу којих се иницијализују подсистеми који ће бити коришћени током компилације.

Анализа (*Performing analysis* у листингу 2.1). Ова фаза примењује статичку анализу Јава кода апликације и његових зависности коришћењем анализе достижности (енгл. *reachability analysis*). У извршиву датотеку биће укључени само елементи за које анализа закључи да су достижни. У овој фази се такође региструју класе и методе за рефлексију и стране библиотеке. Анализа је кључна за минимизацију величине извршиве датотеке и елиминацију некоришћеног кода.

Изградња универзума (*Building universe* у листингу 2.1). Универзум (енгл. *universe*) представља *GraalVM*-ову интерну репрезентацију свих типова, метода и класа које је анализа закључила да су достижне у претходној фази. Интерна репрезентација укључује конструкцију симболичког приказа програма, при чему се за сваки тип одређује име типа заједно са пакетом, хијерархијом наслеђивања, интерфејсима и зависностима. Ова структура је неопходна за следеће фазе оптимизације и генерисања кода.

Парсирање метода (*Parsing methods* у листингу 2.1). У овој фази, *GraalVM Native Image* парсира Јава бајткод метода који су били означени као достижни. Парсер трансформише Јава бајткод у *Graal IR*, графовску међурепрезентацију. Ово је нужан корак јер *Graal IR* представља унификовану графовску репрезентацију програма која омогућава фино гранулисану анализу

података и контролу тока, што је предуслов за примену напредних оптимизација.

Уметање метода (*Inlining methods* у листингу 2.1). Уметање метода подразумева замену позива метода њиховим телом, чиме се уклања трошак позива. Уметање метода често омогућава да компајлер види целокупан контекст израза, што повећава могућност примене оптимизација као што су:

- елиминација мртвог кода (енгл. *dead code elimination*) која елиминише кôд који сигурно неће бити извршен,
- преизрачунавање константи (енгл. *constant folding*) која мења изразе директно резултатом и
- размотавање петљи (енгл. *loop unrolling*) која понавља петље више пута ради смањења броја итерација и гранања.

Компајлер *GraalVM Native Image* одлучује који позиви ће бити уметнути на основу хеуристика, величине метода и профила извршавања, ако су доступни. Ова фаза значајно утиче на перформансе коначне извршиве датотеке.

Компајлирање метода (*Compiling methods* у листингу 2.1). Овде почиње компајлирање међурепрезентације *Graal IR* у машински кôд за циљну архитектуру (нпр. *x86-64*). Током ове фазе, компајлер може применити различите оптимизације које се тичу генерисаног кода. Креиране функције се смештају у меморију и мапирају за даље распоређивање.

Распоређивање метода (*Laying out methods* у листингу 2.1). Ова фаза одређује физички распоред метода у извршивој датотеци. Уколико се примењују оптимизације вођене профилима, компајлер користи алгоритме распоређивања метода који се ослањају на податке из профила како би одредио редослед метода. У случају да профили нису доступни, примењују се алгоритми засновани на хеуристикама. Ово је централна фаза за овај рад и сви детаљи и конкретни алгоритми везани за ову фазу биће представљени у поглављу 3.

Креирање извршиве датотеке (*Creating image* у листингу 2.1). У завршној фази, све секције програма, укључујући машински кôд, метаподатке, и информације за дебаговање, се смештају у коначну извршиву датотеку. Овде се

завршава цео процес компајлирања, при чему је добијен програм самосталан, без потребе за *JVM*-ом и спреман за директно покретање на циљном оперативном систему и циљној архитектури рачунара.

Приликом изградње извршиве датотеке коју спроводи *GraalVM Native Image*, у свакој фази емитују се статистички подаци који помажу разумевању тока компајлирања, укључујући време трајања фазе, заузеће меморије и путање значајних креираних датотека. Такође се приказују информације важне за разумевање резултата појединих фаза. На пример, за фазу анализе достижности то су број типова, поља и метода који су означени као достижни.

2.2 Опције превођења и извршавања

Опције се у компајлеру *GraalVM Native Image* деле у две категорије: опције изградње, које се називају *hosted* опције и које се примењују у време изградње (енгл. *build-time*), и опције извршавања, које се називају *runtime* опције, и које се примењују у време извршавања генерисане извршиве датотеке.

Опције изградње могу се задати приликом изградње на два начина:

1. `-H:±opcija` — у случају опција које могу бити само укључене или искључене. На пример, опција `-H:+PrintCodeLayout` обезбеђује испис редоследа метода након што се распореде, са додатним подацима као што су позиција у меморији, величина методе, број позива, време позива и слично.

2. `-H:opcija=vrednost` — у случају када је потребно поставити одређену вредност за дату опцију. На пример, у контексту распоређивања метода, `-H:CodeSectionLayoutOptimization=SortByHotness` поставља алгоритам за распоређивање метода који ће се користити на вредност *SortByHotness*.

Опције извршавања најчешће се задају приликом покретања извршиве датотеке и разлика у односу на опције изградње приликом задавања је што користе префикс `-R` уместо `-H`. На пример, `-R:+PrintGCSummary` укључује исписивање статистике сакупљача отпадака.

2.3 Алат *mx*

Алат *mx* је основни алат за изградњу, управљање и извршавање пројеката у оквиру система *GraalVM*. За разлику од класичних алата за изградњу као што су *Maven* [2] или *Gradle* [3], алат *mx* је специјализован за потребе развоја *Graal*-а и његових компоненти. Он обједињује изградњу, тестирање и управљање зависностима, као и покретање интеграционих тестова и референтних програма, чиме елиминише потребу за комбинацијом више различитих алата.

Улога алата *mx* није ограничена на изградњу пројекта, већ обухвата и оркестрацију експеримената: покретање програма у више конфигурација, мерење различитих метрика, као и извоз резултата у стандардизованом *JSON* формату. Овај приступ елиминише потребу за ручним прикупљањем података и олакшава и омогућава доследно и поуздано анализирање резултата различитих експеримента.

У оквиру алата *mx*, постоји инфраструктура за мерење ресурса и перформанси апликација компајлираних помоћу *GraalVM Native Image*, која обухвата више нивоа анализе и прикупљања података. Основна функција за праћење меморијске употребе је `ps_poller`, која периодично мери меморијску употребу циљног процеса и свих процеса у његовој сесији. Ова функција покреће задати процес у новој сесији и користи системску команду `ps` за прикупљање података о меморијском отиску. Псеудокод функције `ps_poller` се налази у листингу 2.2.

Алгоритам 2.2: Функција `ps_caller`

Улаз: Циљна команда и њени аргументи `target_cmd`[3], интервал мерења `poll_interval`

Излаз: Излазна датотека са периодичним узорцима RSS меморије

1: **function** PS_POLLER(*target_cmd*, *poll_interval*)
2: `target_proc` ← покрени нову сесију са *target_cmd*
3: **while** `target_proc` је активан **do**
4: сачекај *poll_interval* секунди
5: *data* ← изврши `ps` да се прикупи меморијски отисак свих процеса у сесији `target_proc`
6: упиши *data* у излазну датотеку
7: **return** статус извршења `target_proc`

[3]На пример, за `ls -l /home/user`, `ls` је циљна команда, а `-l /home/user` су аргументи

Инфраструктура такође укључује аутоматизовано покретање и заустављање процеса, обраду грешака и изузетака током мерења, валидацију прикупљених података и генерисање извештаја са визуализацијом резултата.

За мерење перформанси клијент-сервер (енгл. *client-server*) апликација, инфраструктура садржи специјализоване алате који симулирају оптерећење система кроз велики број истовремених захтева (енгл. *load balancing*). Ови алати симулирају више клијентских конекција истовремено, прикупљајући статистике и резулате мерења апликације. Мерења се врше у контролисаним условима са фиксним бројем конекција и дужином теста, обезбеђујући могућност репродуковања резултата.

2.4 Профил извршавања програма

У савременим компајлерима важну улогу играју технике оптимизације које користе информације добијене током извршавања програма, познате као *профил* извршавања програма [13]. У пракси се најчешће користе два приступа добијања информација из извршавања програма, тј. два приступа профајлирању. Први приступ је динамичко профајлирање у контролисаним условима, где се програм извршава са унапред дефинисаним тестовима и улазима. Други приступ је профајлирање током реалне употребе програма у продукционом окружењу. Без обзира на приступ, добијени профил се затим користи ради оптимизације током наредног компајлирања или линковања.

Профил је скуп података који може да обухвати различите врсте информација, као што су: учесталост позива функција и метода, број извршавања грана у условним изразима, редослед у којем се функције позивају, као и временске ознаке одређених догађаја током извршавања. Ови подаци се могу прикупити током извршавања инструментованог програма.

Инструментован кôд је кôд апликације допуњен тако да садржи и инструкције које сакупљају профил извршавања програма. Ове инструкције се додају током компајлирања или линковања, и укључују инструкције инкрементације бројача позива метода, инструкције мерења и записивање времена првог извршавања и слично. Инструментован програм се затим покреће над реалним или посебно припремљеним улазима, што омогућава прикупљање репрезентативних података о његовом понашању у типичном сценарију коришћења.

У контексту компајлера *GraalVM Native Image*, профили се чувају у датотекама са екстензијом `.iprof`. Ове датотеке су у JSON формату и садрже структурисане податке, о извршавању инструментованог програма, који се затим користе током AOT компајлирања у сврху оптимизације програма.

Главна поља која се налазе у `.iprof` датотеци обухватају: верзију формата, описе типова који се појављују у програму, листу метода са њиховим именима и потписима, као и опционе секције које садрже профилне податке. Међу профилним секцијама налазе се:

- број позива метода (у `.iprof` датотеци: *callCountProfiles*),
- време првог позива методе (у `.iprof` датотеци: *firstCallTimestampProfile*),
- статистике гранања (у `.iprof` датотеци: *conditionalProfiles*),
- динамички типови објеката у виртуелним позивима (у `.iprof` датотеци: *virtualInvoke-Profiles*),
- праћење синхронизације (у `.iprof` датотеци: *monitorProfiles*) и
- узорковање стекова позива (у `.iprof` датотеци: *samplingProfiles*).

На основу поља из профила могуће је категорисати методе на различите начине. Један од њих је на учестало позиване методе, које се називају топлим (енгл. *hot*) и ретко или никада позиване методе, које се називају хладним (енгл. *cold*). Подела на топле и хладне методе се прави на основу броја позива метода. У поглављу 3 биће описано како се користи за конструкцију алгоритама за распоређивање метода.

Глава 3

Распореди метода

Распоред метода у извршивој датотеци може утицати на перформансе извршавања. У оквиру компајлера *GraalVM Native Image* развијени су различити алгоритми распоређивања метода који ће у наставку бити описани. Они се деле у три категорије:

- Прва категорија су алгоритми насумичног распоређивања метода. Ови алгоритми се користе као основа за поређење са осталим алгоритмима. Очекивање је да свака оптимизација треба да дâ боље резултате од насумичног распоређивања метода.
- Друга категорија обухвата хеуристички приступ који у основи користи распоређивање метода засновано на именима метода.
- Трећи категорија обухвата алгоритме који се заснивају на коришћењу профила извршавања програма.

3.1 Алгоритми насумичног распоређивања метода

Алгоритми из ове секције представљају основу за поређење, илуструјући утицај случајно изабраног распореда метода на перформансе. Очекивање је да сви алгоритми засновани на другим приступима дају боље резултате од ових алгоритама. Ови алгоритми су имплементирани у оквиру рада на тези и названи су `ChooseRandom` и `ChooseRandomOnePerPage`.

Алгоритам `ChooseRandom`

Алгоритам `ChooseRandom`, приказан у листингу 3.1, додељује насумичан редослед методама, користећи фиксну иницијалну вредност (енг. *seed*) 42 у експериментима, чиме се обезбеђује детерминизам и омогућава поновљивост експеримената.

Алгоритам 3.1: Алгоритам *ChooseRandom* за насумично одређивање распореда метода

Улаз: Листа метода M

Излаз: Листа метода L у насумичном редоследу

1: **function** ChooseRandomLayout(M)
2: Постави семе генератора случајних бројева на $s = 42$
3: $L \leftarrow$ Измешана листа M насумично према семену s
4: **return** L

Алгоритам `ChooseRandomOnePerPage`

Алгоритам `ChooseRandomOnePerPage` је варијанта стратегије `ChooseRandom` дизајнирана са намером да максимизује број промашаја приликом приступа страницама (енг. *page faults*). Као и `ChooseRandom`, алгоритам `ChooseRandomOnePerPage` додељује методама случајан редослед, али додатно осигурава да сваки метод буде смештен на засебну меморијску страницу.

Алгоритам *ChooseRandom* служи као референтни алгоритам који представља реалистично полазиште: методе се распоређују насумично, али и даље је након учитавања странице која садржи један метод могуће, и врло вероватно, да ће бити учитан и неки други потребни метод или више њих. У зависности од њиховог распореда, број промашаја страница може да буде различит. Насупрот томе, алгоритам *ChooseRandomOnePerPage* представља екстремни сценарио, јер приморава да се свака метода налази на засебној страници и тиме осигурава најгори могући случај просторног локалитета, односно за сваки позвани метод сигурно ће доћи до промашаја странице. На тај начин можемо да проценимо најгоре могуће време извршавања програма. Међутим, овај алгоритам нема смисла користити за поређење меморијског отиска.

Псеудокод *ChooseRandomOnePerPage* алгоритма је дат у листингу 3.2.

Алгоритам 3.2: Алгоритам *ChooseRandomOnePerPage* за насумично одређивање распореда метода где једна метода припада једној страници

Улаз: Листа метода M, величина странице ps

Излаз: Листа метода L са по једним методом по страници

1: **function** ChooseRandomOnePerPageLayout(M, ps)
2: Постави семе генератора случајних бројева на $s = 42$
3: $L \leftarrow$ Измешана листа M насумично према семену s
4: **for all** $m \in L$ **do**
5: **if** $|m| < ps$ **then**
6: Бинарном коду m додај онолико бајтова колико је потребно да се попуни страница
7: **else**
8: Распореди m преко $\lceil |m|/ps \rceil$ страница
9: Ако последња страница није у потпуности попуњена, додај онолико бајтова колико је потребно да се допуни
10: **return** L

3.2 Алгоритми засновани на хеуристикама

Алгоритми распоређивања метода засновани на хеуристикама користе само статичке податке који су присутни након фазе анализе у процесу компајлирања. Такви подаци су, на пример: имена метода, имена класа, типови параметара и типови повратне вредности.

Алгоритам *SortByMethodName*

sta Сортирање по називу методе подразумева да се методе сортирају по кључу. Кључ за сортирање формира се према обрасцу:

```
H.n(P):R
```

при чему је

- `H` ознака за пуно име класе или интерфејса у коме је метода дефинисана (заједно са свим пакетима којима та класа припада),

- `n` ознака за назив методе,
- `P` ознака за типове параметара методе,
- `R` ознака за тип повратне вредности методе.

На пример, за методу `add` унутар класе `LinkedList` која припада пакету `util` који даље припада пакету `java`, пуно име је `java.util.LinkedList`. Ако метода `add` прима један параметар типа `int` и враћа вредност типа `boolean`, добија се кључ:

```
java.util.LinkedList.add(int):boolean
```

Алгоритам сортирања ће на основу овако формираног кључа груписати методе из истог пакета, па затим из истих класа заједно. Претпоставка овог алгоритма је да методе које су у истој класи најчешће позивају друге методе из исте класе и истог пакета у којем се и саме налазе. Таквим груписањем побољшава се просторни локалитет приликом извршавања програма. Псеудокод алгоритма је приказан у листингу 3.3.

Алгоритам 3.3: Алгоритам *SortByMethodName* за распоређивање метода по њиховом имену

Улаз: Листа метода M

Излаз: Сортирана листа метода L

1: **function** SortByMethodName(M)

2: Дефиниши функцију кључа за методу m из листе:

$$\text{key}(m) \leftarrow \text{format}(m, \{\%H.\%n(\%P) : \%R\})$$

3: $L \leftarrow$ Сортиране методе по формираном кључу

4: **return** L

Алгоритам `SortByMethodNameSemantics`

Алгоритам `SortByMethodNameSemantics` је унапређена верзија апгоритма `SortByMethodName`, није отвореног кода и доступан је само у оквиру напредне, комерцијалне верзије система *GraalVM* (*GraalVM Enterprise Edition*). Овај алгоритам врши детаљнију анализу, тј. не анализира само блискост по класи

и типовима параметара и повратних вредности метода. Семантичка анализа врши се издвајањем кључних речи из назива метода и такође разматрањем графа контроле тока и позива метода у коду. Методе из исте класе се партиционишу у пет концептуалних слојева који се затим спајају.

3.3 Алгоритми засновани на профилу извршавања програма

Профил програма представља кључне податке на основу којих алгоритми распоређивања врше распоређивање метода. Анализом различитих информација из профила, попут броја позива метода или временских ознака првог позива (енгл. *first call timestamp*) могуће је усмерити распоред метода ка побољшању одређених метрика приликом извршавања програма попут меморијског отиска тако што ће се методе груписати у складу с временом првог позива или учесталости позива.

Битно је напоменути да немају све методе профиле и за већину реалних програма неке методе неће имати профиле, јер нису биле извршене. На пример, методе које обрађују грешку која се никад није догодила током профајлирања неће имати профил.

Основна идеја свих алгоритама засновних на профилу извршавања је да се методе партиционишу на топле и хладне, тако да се на почетку кодне секције налазе топле методе, а након њих хладне. Очекивање је да хладне методе буду ређе коришћене или се можда уопште неће користити током извршавања, што омогућава смањење употребе меморије. Сви алгоритми засновани на профилу извршавања програма су затвореног кода и доступни само у оквиру напредне, комерцијалне верзије система *GraalVM*.

Алгоритам `SortByHotness`

Алгоритам `SortByHotness` се заснова на подацима о броју позива метода који се добија из профила програма. Методе са профилима сортирају се опадајуће према броју позива, док се методе без профила додају након њих.

Алгоритам `ClosestIsBest`

Овај алгоритам представља проширење класичног приступа распоређивању метода вођеног профајлирањем. Класичан алгоритам су представили Петис и Хансен [15]. У оквиру овог алгоритма, све методе се најпре сортирају по учесталости позива. Након тога, гради се неусмерен граф где чворови представљају методе, док ивице носе тежину која одговара броју директних позива између метода које та ивица спаја. Број позива метода добија се из профила програма, а парови метода се добијају из графа контроле тока. Граф се гради тако што се крене од најучесталијег метода и додају се наредни методи по редоследу учесталости.

Генерисани граф се обрађује итеративним спајањем кластера. Кластери су групе чворова које су у почетку једночлане (свака метода чини један кластер) и спајају се по следећим правилима. *Најјача веза* између два кластера представља ивицу највеће тежине између крајњих чворова тих кластера. При сваком спајању два кластера, редослед метода у резултујућем кластеру одређује се на основу најјаче везе. На пример, претпоставимо да имамо:

- Леви кластер: [`a`, `b`, `c`]
- Десни кластер: [`d`, `e`, `f`, `g`]

Тада резултујући редослед зависи од тога која је веза најјача:

- ако је `a-d` најјача: [`g`, `f`, `e`, `d`, `a`, `b`, `c`]
- ако је `a-g` најјача: [`d`, `e`, `f`, `g`, `a`, `b`, `c`]
- ако је `c-d` најјача: [`a`, `b`, `c`, `d`, `e`, `f`, `g`]
- ако је `c-g` најјача: [`a`, `b`, `c`, `g`, `f`, `e`, `d`]

Алгоритам итеративно спаја најјаче повезане кластере. Мотивација за овај алгоритам је да се поред учесталости искористи и структура позива тако да се међусобно позиване методе приближе под претпоставком да ће углавном бити потребно да оне буду заједно учитаване, јер се извршавају једна за другом. Псеудокод класичног алгоритма је приказан у листингу 3.4. Побољшања овог алгоритма која су имплементирана у оквиру система *GraalVM* обухватају ограничавање величине графа над којим се ради спајање као и коришћење величине странице као ограничења приликом прављења кластера.

Алгоритам 3.4: Алгоритам *ClosestIsBest* за распоређивање метода на основу графовске кластеризације

Улаз: Листа метода M са фреквенцијама позива $w(m_i, m_j)$ између метода

Излаз: Уређена листа метода L

1: Иницијализуј граф $G = (V, E)$ са $V = \{\{m\} \mid m \in M\}$ као појединачне кластере
2: Иницијализуј тежине ивица $w(C_i, C_j)$ из фреквенција позива између кластера $C_i, C_j \in V$
3: **while** $E \neq \emptyset$ **do**
4: $(C_a, C_b) \leftarrow$ ивица са максималном тежином у E
5: Уклони (C_a, C_b) из E
6: $C_{new} \leftarrow$ спој кластере C_a и C_b
7: Уклони C_a, C_b из V
8: Додај C_{new} у V
9: **for all** $C \in V \setminus \{C_{new}\}$ **do**
10: $w(C_{new}, C) \leftarrow w(C_a, C) + w(C_b, C)$ ▷ Спој тежине ивица
11: Уклони све ивице које имају C_a или C_b као један од крајева.
12: Додај ивицу (C_{new}, C) са тежином $w(C_{new}, C)$ у E
13: $L_1 \leftarrow$ линеарни редослед метода добијен спајањем метода у кластерима
14: $L_2 \leftarrow$ методе без профила
15: **return** $L_1 \circ$ [1] L_2

Алгоритам `ClusterByEdges`

Алгоритам `ClusterByEdges` представља поједностављену варијанту алгоритма `ClosestIsBest`. Циљ овог приступа је да групише методе које се често позивају у уређене секвенце, засноване на профилима. Разматрају се само појединачне методе и директан број позива између њих и на основу тога се спајају заједно у секвенце.

Мотивација овог алгоритма је да сачува главну идеју из алгоритма `ClosestIsBest`, али уз поједностављену реализацију која не захтева конструкцију графа. То чини да време превођења програма буде краће.

[1] Операција $\circ$ означава конкатенацију листа, односно редоследно спајање елемената L_1 и L_2.

Алгоритам *SortByFirstCall*

Слично као алгоритми засновани на учесталости позива метода, из профила се може посматрати и информација о редоследу првог позива методе и на основу ње формирати алгоритам распоређивања метода. Алгоритам `SortByFirstCall` сортира методе по редоследу њиховог првог позива растуће, док методе које немају информације у профилу сортира по називу. Идеја овог приступа је да методе које имају временску локалност приликом извршавања имају приближно сличну просторну локалност у распореду у меморији. Претпоставка оваквог распореда је да ће минимизовати учесталост учитавања различитих страница меморије, пошто се позиви извршавају у временском редоследу који одговара и њиховом физичком распореду у меморији.

Редослед првог позива метода може се одредити током профајлирања инкрементирањем глобалног атомичног бројача и доделом његове тренутне вредности подацима профила методе при њеном првом позиву. Варијанта овог алгоритма се може наћи у литератури [9], а псеудокод је приказан у листингу 3.5.

Алгоритам 3.5: Алгоритам *SortByFirstCall* за распоређивање метода на основу редоследа првог позива методе

Улаз: Листа компајлираних метода M, профили P

Излаз: Листа метода сортирана по редоследу првог позива

1: **function** SortByFirstCall(M, P)
2: $L \leftarrow [(m, P[m].callorder) \mid m \in M, !P[m].callorder = null]$
3: Сортирај L у растућем редоследу користећи $P[m].callorder$ као кључ
4: $L_1 \leftarrow [m|(m, P[m].callorder) \in L]$
5: $L_2 \leftarrow$ Методе без профила
6: return $L_1 \circ L_2$

Глава 4

Евалуација

За процену утицаја разматраних алгоритама на перформансе програма користе се постојећи референтни програми (енгл. *benchmarks*). Компаративна анализа заснива се на просечним резултатима више извршавања сваког програма узимајући у обзир и стандардну девијацију резултата мерења.

4.1 Референтни програми

Референтни скуп програма садржи програме и тестове који по својој структури одговарају реалним апликацијама. Основна улога ових програма је мерење перформанси и ефикасности софтверских система.

У овом раду користимо више референтних програма. Они покривају различите парадигме развоја у екосистему Јава, од класичних једнонитних и монолитних апликација до савремених система заснованих на архитектури микросервиса.

Савремене апликације засноване на архитектури микросервиса у оквиру референтних програма су развијене коришћењем радних оквира (енгл. *framework*) као што су *Micronaut* [4], *Quarkus* [5] и *Spring Boot* [6]. Ови оквири пружају подршку за креирање продукцијских Java и Kotlin сервиса уз акценат на брзо покретање и малу потрошњу меморије. У наставку су дати њихови кратки описи.

Micronaut је радни оквир заснован на принципу реактивног програмирања (енгл. *reactive programming*) са акцентом на мало кашњење и минималан додатни трошак самог оквира у погледу меморије и процесорског времена

током извршавања. *Micronaut* је прилагођен за AOT компајлирање са компајлером *GraalVM Native Image*.

Quarkus је модерни Јава радни оквир дизајниран посебно за окружења у облаку и контексте где је битно брзо покретање и мала потрошња ресурса. Омогућава AOT компајлирање апликација помоћу компајлера *GraalVM Native Image* и напредне технике оптимизације које резултују малом извршивом датотеком и њеним брзим покретањем.

Spring Boot је најпопуларнији Јава радни оквир за развој микросервиса, који поједностављује конфигурацију и покретање апликација кроз бројне унапред дефинисане шаблоне и компоненте. Уз пројекат *Spring Native* и званичну подршку за компајлер *GraalVM Native Image* од верзије *Spring Boot 3*, омогућава се AOT компајлирање *Spring* апликација у самосталне извршиве датотеке које се брзо покрећу и ефикасно користе ресурсе, што је посебно корисно у микросервисним архитектурама и окружењима у облаку.

Ови радни оквири омогућавају изградњу апликација које користе све предности *GraalVM*-а, али они нису предмет директне анализе, већ служе као основа за референтне апликације које се користе у експериментима. Избор је мотивисан широком употребом ових радних оквира у индустрији, што резултате чини репрезентативним и блиским реалним сценаријима. Неки од најзначајнијих референтних програма развијених коришћењем споменутих оквира су *Shopcart* [16], *Tika* [7] и *Petclinic* [16]. Поред њих, посебно се издвајају и референтни скупови програма *Dacapo* [8] и *Barista* [1].

Shopcart је клијент-сервер апликација за онлајн куповину развијена коришћењем радног оквира *Micronaut*. Служи као референтни програм за процену перформанси и скалабилности микросервиса.

Tika је клијент-сервер апликација изграђена са радним оквиром *Quarkus*. Користи се за парсирање и извлачење метаподатака и текстуалног садржаја из различитих формата докумената као што су *PDF (Printable Document Format)* и *ODT (Open Document Text)*. Овај референтни програм мери перформансе задатака који захтевају обраду великих количина података. Такође, процењује перформансе апликације при обради различитих формата и величина докумената.

Petclinic је клијент-сервер апликација која представља систем ветеринарске клинике, који управља услугама, прегледима и терминима за заказивање у својству неге кућних љубимаца. Апликација *Petclinic* је развијена коришћењем радног оквира *Spring* и служи као стандардни референтни програм за демонстрацију могућности *Spring*-а у изградњи робусних, пословних апликација.

DaCapo је скуп Јава апликација отвореног кода дизајнираних за евалуирање перформанси Јава виртуелних машина. Скуп укључује разноврсне апликације из стварног света, пружајући свеобухватну процену перформанси Јава виртуелне машине под различитим оптерећењима.

Barista је скуп референтних микросервиса отвореног кода који се извшравају на платформи *JVM* и подржавају *AOT* компајлирање. Састоји се од апликација написаних у разним популарним оквирима. Скуп *Barista* је дизајниран да евалуира перформансе микросервиса у традиционалном *JVM* режиму извршавања као и у *native-image* режиму извршавања, користећи могућности које пружа *GraalVM*.

Овако разноврстан скуп референтних програма омогућава свеобухватну евалуацију компајлера *GraalVM Native Image*, како у погледу основних перформанси, тако и у специфичним сценаријима као што су покретање микросервиса, рад под оптерећењем, или процесирање велике количине података.

4.2 Метрике

У контексту овог рада, алат *mx* је коришћен за покретање референтних програма и систематско прикупљање података о њиховим перформансама. Када референтни програми заврше извршавање, генерише се датотека *bench_results.json* која садржи различите информације о том извршавању. Метрике садржане у датотеци *bench_results.json* које су најзначајније за алгоритме распоређивања метода су описане у наставку.

Пропусни опсег (енгл. *throughput*) означава број захтева које систем може обрадити у јединици времена. Пратимо *максимални пропусни опсег*, који показује максимални одрживи капацитет обраде система.

Кашњење (енгл. *latency*) означава колико брзо систем реагује на захтев. Посебно се прати *максимално кашњење* да би идентификовали најгоре случајеве кашњења. У раду се посматра 99. перцентил ових мерења. Узимање 99. перцентила је уобичајена пракса у евалуацији система јер обезбеђује стабилније и репрезентативније резултате, зато што елиминише утицај појединачних екстремних случајева и омогућава поређење које одражава реално корисничко искуство.

Време првог одзива (енгл. *time to first response, TTFR* представља време између пријема захтева и слања првог бајта одзива. Прате се само хладна покретања, што значи да ништа није присутно у кешу у тренутку мерења *TTFR*.

Величина резидентног скупа (енгл. *resident set size, RSS*) је максимална количина физичке меморије коју заузима процес. Урачуната је целокупна потрошња меморије, укључујући дељене ресурсе између процеса.

Инфраструктура за мерења и проширење

Поред основне функције `ps_poller` која постоји у оквиру алата *mx*, у склопу овог рада развијено је проширење `ps_caller_memory` за платформу *Linux* које додатно анализира меморијске податке специфично за кодне секције процеса. Ово проширење чита датотеку `/proc/<pid>/smaps` и сабира меморијске вредности које припадају страницама са извршним дозволама (`r-xp`). У `/proc/<pid>/smaps`, странице са ознаком `r-xp` представљају извршиве (*read, execute*) меморијске странице, односно део меморије који садржи машински кôд који се може извршавати. Друге странице, као што су `rw-p` (*read,write*) - подаци или `r-p` (*read-only*) - константе, не представљају кодне секције. На тај начин се прати искључиво меморијски отисак извршивог кода.

Фокусирање анализе искључиво на странице са ознаком `r-xp` пружа детаљан увид у меморијски отисак извршивог кода, који је кључан за анализу утицаја алгоритама оптимизације распореда метода као и других делова целог компајлера *GraalVM Native Image*. Псеудокод функције `ps_caller_memory` је приказан у листингу 4.1.

Алгоритам 4.1: Проширење `ps_caller_memory` са посебним кораком за мерење меморије кодне секције

Улаз: Циљни командни низ `target_cmd`, интервал мерења `poll_interval`
Излаз: Излазна датотека са периодичним узорцима RSS меморије за цео процес и кодне секције

1: **function** PS_CALLER_MEMORY(*target_cmd*, *poll_interval*)
2: `target_proc` ← покрени нову сесију са *target_cmd*
3: `target_pid` ← PID процеса `target_proc`
4: **while** `target_proc` је активан **do**
5: сачекај *poll_interval* секунди
6: *rss_data* ← изврши `ps` да се прикупи RSS свих процеса у сесији `target_proc`
7: (*rss_code*) ← анализирај `/proc/<pid>/smaps` за странице са дозволом `r-xp`
8: упиши (*rss_data*, *rss_code*) у излазну датотеку
9: **return** статус извршења `target_proc`

4.3 Резултати евалуације

За све споменуте референтне програме добијени су засебни резултати мерења коришћењем алата *mx* који су филтрирани по значајним метрикама и детаљно представљени у апендиксу А. Ради прегледности направљене су агрегиране табеле за сваку метрику. Агрегиране табеле приказују резултате свих алгоритама за распоређивање метода у релативном односу према референтном алгоритму *ChooseRandom*.

За сваку од метрика, мерења су вршена по 10 пута за сваки референтни програм. У свакој агрегираној табели је приказан просек свих мерења алгоритма у односу на просек вредности референтног алгоритма (у табели је то колона %), просек свих релативних стандардних девијација[1] у групи мерења (у табели је то колона Просек РСД). Минимум и максимум представљају процентуално најмање и највеће вредности мерења алгоритма у односу на просек вредности мерења референтног алторитма (у табели су то колоне Мин % и Макс %). На основу ових резултата може се уочити општи тренд и издвојити који алгоритам у просеку даје боље резултате, као и за које метрике и алгоритме се јављају већа одступања.

[1]Релативна стандардна девијација представља стандардну девијацију изражену процентуално у односу на просек.

Величина резидентног скупа

Величина резидентног скупа програма мерена је на свим описаним референтним програмима. Обрађени резултати свих мерења су приказани у табели 4.1.

Табела 4.1: Агрегирани резултати по алгоритмима – Метрика: RSS (у односу на *ChooseRandom* = 100%).

Алгоритам	**%**	**Просек РСД**	**Мин %**	**Макс %**
SortByMethodName	95.2	±1.2	89.8	101.5
SortByMethodNameSemantics	94.7	±1.7	88.9	102.2
SortByHotness	88.2	±1.6	82.7	99.9
ClosestIsBest	86.8	±2.8	81.9	95.9
ClusterByEdges	87.3	±2.4	82.3	96.7
SortByFirstCall	87.1	±1.9	81.7	97.6

Алгоритам *ChooseRandomOnePerPage* представља вештачки сценарио који симулира најгори могући случај по перформансе, где сваки метод први пут изазива странични промашај у меморији. Због тога, за мерења величине резидентног скупа овај алгоритам није релевантан, јер у реалним апликацијама методе се не постављају појединачно на посебне странице.

Када се посматрају алгоритми засновани на хеуристикама, алгоритми *SortByMethodName* и *SortByMethodNameSemantics*, примећује се побољшање од 4.8, односно 5.3 процента. То сведочи да се укупна меморија побољшава чак и алгоритмима заснованим на хеуристикама, док напреднија хеуристика која укључује и семантику метода даје боље резултате од алгоритма *SortByMethodName*.

С друге стране алгоритми засновани на профилима дају значајно боље резултате од референтног алгоритма, али и од алгоритама заснованих на хеуристикама. Алгоритми *SortByHotness*, *ClosestIsBest*, *ClusterByEdges*, *SortByFirstCall* дају побољшања од 11 до 13.2 процената. То показује јасно побољшање по питању меморије када се користе профили. Додатно, алгоритам *ClosestIsBest* има најбоље резултате. Међутим, како је просек релативне стандардне девијације 1.9 процената примећујемо да алгоритам *ClosestIsBest* даје приближне резултате осталим алгоритмима који се заснивају на профилима. Сви просеци релативне стандардне девијације су између 1 и 2.8 процената што говори о стабилности мерења.

Минималне и максималне вредности (Мин % и Макс %) у табели 4.1 говоре о стабилности мерења и илуструју најбоље и најгоре резултате у односу на референтни алгоритам. За алгоритме засноване на хеуристикама, *SortByMethodName* и *SortByMethodNameSemantics*, распон минималне и максималне вредности од 11.3, односно 13.3 показују да побољшање укупне меморије остаје релативно стабилно за све референтне програме, са мањим варијацијама око просека.

Алгоритми засновани на профилима (*SortByHotness*, *ClosestIsBest*, *ClusterByEdges*, *SortByFirstCall*) имају нешто шири распон минимума и максимума. На пример, *ClosestIsBest* има распон 14%, а за *SortByFirstCall* 15.9%. Из ових вредности можемо закључити да алгоритми засновани на профилима варирају у резултатима у складу са карактеристикама програма.

Величина резидентног скупа на секцији кода

Величина резидентног скупа на секцији кода програма мерена је на свим описаним референтним програмима. Резултати за величину резидентног скупа на секцији кода приказани су у Табели 4.2.

Засебно мерење величине резидентног скупа само на секцији кода вођено је чињеницом да се приликом распоређивања метода мења искључиво део секције кода и да се тиме само на њу и утиче. Очекивање је да програми с већом секцијом кода могу имати значајнија побољшања приликом употребе оптимизација алгоритама за распоређивања метода у коду. Разлог је што већи обим извршивог кода заузима више меморијских страница и тиме пружа више могућности да се побољша просторни локалитет и смањи потреба за учитавањем страница из секундарне меморије.

Табела 4.2: Агрегирани резултати по алгоритмима – Метрика: RSS на секцији кода (у односу на *ChooseRandom* = 100%)[2].

Алгоритам	%	**Просек РСД**	**Мин %**	**Макс %**
SortByMethodName	75.6	±1.4	64.0	84.5
SortByMethodNameSemantics	73.3	±0.2	60.0	84.5
SortByHotness	43.9	±2.2	36.0	51.4
ClosestIsBest	43.7	±2.7	36.0[2]	52.1
ClusterByEdges	44.4	±2.5	36.0[2]	53.5
SortByFirstCall	42.5	±1.1	36.0[2]	49.3

Разлике у резултатима су приметније на величини резидентног скупа секције кода него на величини резидентног скупа целог програма, зато што су саме величине секција кода у свим референтним програмима пет до десет пута мања него величина резидентног скупа целог програма. Поређење табеле 4.1 и табеле 4.2 потврђује то.

Из истих разлога као и у претходној секцији, о резидентном скупу целог програма, алгоритам *ChooseRandomOnePerPage* неће бити разматран ни у овој секцији. На осталим алгоритмима се може приметити значајно побољшање у односу на референтни алгоритам него код меморије целог програма. То је очекивано понашање са обзиром да побољшање распореда метода утиче на мање неопходних учитавања метода што је процентуално значајније за кодну секцију.

Алгоритми засновани на хеуристикама, *SortByMethodName* и *SortByMethodNameSemantics*, дају побољшања од 24.4 и 26.7 процената, редом. Алгоритми засновани на профилима поново дају боље резултате. Алгоритми *SortByHotness*, *ClosestIsBest*, *ClusterByEdges*, *SortByFirstCall* дају побољшања од 53.6 до 57.5 процената и може се приметити да *SortByFirstCall* даје најбоље резултате.

Вредности просека стандардних девијација мањих од 3 процената говоре о стабилности мерења, али и о томе да по овој метрици *SortByFirstCall* даје боље резултате у односу на остале алгоритме и да је податак из профила о првом времену извршавања методе релевантнији за резидентни скуп секције кода у односу на број позива метода.

Вредности најмањег и највећег процентуалног одступања алгоритама у односу на референтни алгоритмам приказују да се алгоритми *SortByMethodName* и *SortByMethodNameSemantics* имају нешто шири распон од 20.5% и 24.5%, редом, што указује да је побољшање у односу на референтни програм стабилно.

Алгоритми засновани на профилима (*SortByHotness*, *ClosestIsBest*, *ClusterByEdges*, *SortByFirstCall*) показују шири распон минимума и максимума, на пример *ClosestIsBest* има распон од 16.1% док *SortByFirstCall* има распон од 13.3%. Ово значи да иако просечне вредности указују на значајно побољшање, конкретни ефекти зависе од величине и структуре секције кода у сваком програму. Ипак, чак и минималне вредности показују да алгоритми засновани на профилима увек пружају боље резултате од алгоритама засно-

[2]Вредности 84.5 и 36.0 се понављају због заокруживања резултата при обради *bench_results.json*. Код резидентног скупа секције кода, чије су вредности између 10 и 25 MB, овај ефекат је израженији.

ваних на хеуристикама, што потврђује релевантност коришћења података из профила при оптимизацији распореда метода.

Време првог одзива

Време првог одзива програма мерено је на свим описаним клијент-сервер референтним програмима (за остале програме ова метрика није релевантна). Резултати за ову метрику приказани су у Табели 4.3. За разлику од метрика заузећа меморије, где се промена распореда метода манифестује индиректно кроз утицај на употребу страница, овде је ефекат директнији: боље распоређивање метода може убрзати учитавање иницијалних функција и тиме смањити време првог одзива.

Табела 4.3: Агрегирани резултати по алгоритмима – Метрика: Time-to-First (у односу на *ChooseRandom* = 100%).

Алгоритам	%	**Просек РСД**	**Мин** %	**Макс** %
ChooseRandomOnePerPage	356.9	±1.3	287.1	402.1
SortByMethodName	86.7	±1.5	79.6	92.2
SortByMethodNameSemantics	85.4	±1.8	77.0	91.8
SortByHotness	73.9	±3.6	68.4	81.7
ClosestIsBest	74.7	±3.7	68.1	83.5
ClusterByEdges	74.3	±2.8	68.0	83.2
SortByFirstCall	68.3	±3.3	63.6	75.7

Највећа разлика је између референтног алгоритма и алгоритма *ChooseRandomOnePerPage*, где је просечно време првог одзива преко три пута веће од референтног. Постављање свега једне методе по страници доводи до тога да се сваки пут дешава промашај и да је неопходно учитавање целе странице што доводи до успорења времена првог одзива.

Код алгоритама заснованих на хеуристикама, *SortByMethodName* и *SortByMethodNameSemantics*, уочавају се побољшања у односу на основни случај од 13.3 и 14.6 процената. Ови резултати указују да постоји корелација између припадности методе пакету и класи и вероватноћи да ће она звати неку другу методу из истог пакета или класе.

С друге стране, алгоритми који користе профиле, *SortByHotness*, *ClosestIsBest*, *ClusterByEdges*, *SortByFirstCall*, показују значајна побољшања, прва три око 26 процената, а алгоритам *SortByFirstCall* чак 32 процента. С

обзиром да је основна информација коју алгоритам *SortByFirstCall* користи из профила баш време позива, и да су оне прве позиване распоређене на самом почетку, самим тим се смањује број страница које морају бити учитане пре испоруке првог одговора.

Релативна стандардна девијација алгоритма *ChooseRandomOnePerPage* у односу на референтни алгоритам има малу вредност (±1.3%), док су минимална и максимална одступања од просека велика (287.1–402.1%), што показује да лоше распоређивање метода систематски изазива лоше перформансе.

Алгоритми засновани на хеуристикама, *SortByMethodName* и *SortByMethodNameSemantics* имају шире распоне минимума и максимума (12.6% и 14.8%), али релативне стандардне девијације остају мање од ±2%, што указује на стабилност побољшања и минималан утицај структуре програма на ефекат алгоритма.

Алгоритми засновани на профилима показују нешто шире распоне минимума и максимума: *SortByHotness* има распон од 13.3%, *ClosestIsBest* 15.4%, а *SortByFirstCall* има распон од 12.1%. Њихове вредности просека релативних стандардних девијација су мање од ±4% што показује да су резултати конзистентни и предвидљиви. За *SortByFirstCall* подаци из профила о првом позиву методе директно смањују број страница које морају бити учитане пре испоруке првог одговора, што потврђује његову ефикасност у смањењу времена првог одзива.

Кашњење

Кашњење програма мерено је на свим описаним клијент-сервер референтним програмима (за остале програме ова метрика није релевантна). Начин мерења метрике кашњења у алату *mx* је такав да је пре мерења већ све иницијализовано, односно сви потребни ресурси, класе, методе, конфигурације су учитани у меморију. Самим тим, у таквим условима, распоређивање метода не доприноси значајним одсуптањима од референтног алгоритма и сви алгоритми дају сличне резултате. Резултати су приказани у табели 4.4.

Сва одступања која постоје у табели 4.4 припадају распону који је описан релативном стандардном девијацијом и самим тим не представљају репрезентативно значајна одступања од референтног алгоритма.

Табела 4.4: Агрегирани резултати по алгоритмима – Метрика: кашњење

Алгоритам	%	**Просек РСД**	**Мин %**	**Макс %**
ChooseRandomOnePerPage	101.1	±3.7	96.8	105.5
SortByMethodName	99.8	±3.4	98.4	101.6
SortByMethodNameSemantics	99.6	±3.8	97.6	101.0
SortByHotness	99.3	±3.4	96.4	102.8
ClosestIsBest	100.6	±3.2	99.3	102.2
ClusterByEdges	99.0	±4.1	97.0	102.0
SortByFirstCall	99.3	±3.7	97.1	101.4

Пропусност

Пропусност програма мерена је на свим описаним клијент-сервер референтним програмима (за остале програме ова метрика није релевантна). У табели 4.5 су приказани агрегирани резултати по алгоритмима за метрику пропусности. За метрику пропусности веће вредности указују на боље перформансе програма, што представља промену у односу на претходне метрике.

Табела 4.5: Агрегирани резултати по алгоритмима – Метрика: пропусност.

Алгоритам	% (%)	**Просек РСД**	**Мин %**	**Макс %**
ChooseRandomOnePerPage	92.7	±2.9	75.2	98.9
SortByMethodName	101.6	±1.1	98.9	104.7
SortByMethodNameSemantics	101.9	±1.1	99.8	105.0
SortByHotness	102.4	±3.3	100.2	105.8
ClosestIsBest	97.6	±5.3	88.2	101.3
ClusterByEdges	101.9	±4.1	99.4	107.0
SortByFirstCall	101.0	±5.0	96.5	105.0

Из табеле 4.5 се може приметити да су резултати у веома малом опсегу у односу на референтни алгоритам и да не постоје значајне разлике. Једини алгоритам који значајније одсупа по том питању је *ChooseRandomOnePerPage*, који има посебно лоше распоређивање метода према ком програм има неефикасне обрасце приступа.

Чак и у случају где су пре мерења иницијализовани сви објекти, класе и методе што смањује промашаје страница, акумулација неефикасности распореда метода код алгоритма *ChooseRandomOnePerPage* води до пропусности која је у просеку мања за 7.3 процената. Међу осталим алгоритмима, највећа разлика у

резултатима је за алгоритме *ClosestIsBest*, −2.4 процената и *SortByHotness* +2.4 процената. Међутим, сви резултати су у распону стандардне девијације што их не чини репрезентативно бољима или горима од референтног алторитма.

Анализа резултата по свим метрикама

Из представљених резултата примећује се да алгоритми распоређивања метода имају утицаја на величину резидентног скупа целог програма, величину резидентног скупа секције кода и време првог одзива. Кашњење и пропусност нису под утицајем алгоритама за распоређивање метода кода.

Табела 4.6 представља рангирање алгоритама за распоређивање метода по метрикама резидентног скупа целог програма, секције кода и времена првог одзива на основу табела из претходних секција (мањи ранг значи боље резултате). Када су алгоритми означени истим бројем, то значи да је њихова разлика у склопу релативне стандардне девијације и да није значајна.

Табела 4.6: Рангирање алгоритмима по резултатима метрика

Алгоритам	**RSS**	**RSS кодне секције**	**TTFR**
ChooseRandom	3	5	4
ChooseRandomOnePerPage	4	6	5
SortByMethodName	2	4	3
SortByMethodNameSemantics	2	3	3
SortByHotness	1	2	2
ClosestIsBest	1	2	2
ClusterByEdges	1	2	2
SortByFirstCall	1	1	1

Може се приметити да алгоритам *SortByFirstCall* даје најбоље резултате за метрике времена првог одзива и величине резидентног скупа на секцији кода, док за величину резидентног скупа целог програма нема значајног одступања између свих алгоритама за распоређивање метода заснованих на профилима. То отвара простор за додатна надограђивања и развој нових алгоритама, с обзиром да је *SortByFirstCall* једини алгоритам од свих анализираних који користи време првог позива методе.

4.4 Унапређење алгоритама распоређивања метода

Унапређен је алгоритам *SortByMethodNameSemantics*, као и алгоритми засновани на профилима: *SortByHotness*, *ClusterByEdges*, *ClosestIsBest* и *SortByFirstCall*. Алгоритам *SortByMethodNameSemantics* је поједностављен без нарушавања перформанси, док су алгоритми засновани на профилима унапређени тако да методе без профила буду боље распоређене. У случајевима када су методе без профила чешће позиване током извршавања бољи распоред побољшава употребу меморије и времена првог одзива.

Алгоритам `SortByMethodNameSemantics`

Међу пет концептуалних слојева у које су методе партиционисане у алгоритму `SortByMethodNameSemantics`, један слој је посвећен методама које се баве обрадом грешака. Интутивно је претпостављено да постојање овог слоја не доприноси значајно побољшању перформанси и употреби меморије. Исправност ове претпоставке је потврђена и резултати се могу видети у табелама 4.7 и 4.8.

Табела 4.7: Величина резидентног скупа кôд секције с релативним стандардним девијацијама

Референтни програм	Оригинал	Модификација
spring-petclinic	41.7 ± 0.5 MB	41.6 ± 0.4 MB
micronaut-helloworld	13.0 ± 0.0 MB	13.1 ± 0.1 MB
micronaut-shopcart	16.0 ± 0.0 MB	15.8 ± 0.1 MB
quarkus-helloworld	12.0 ± 0.0 MB	12.0 ± 0.1 MB
quarkus-tika	15.0 ± 0.0 MB	15.1 ± 0.1 MB
dacapo	27.0 ± 0.0MB	26.9 ± 0.2 MB

Табела 4.7 представља резултате 10 мерења величине резидентног скупа кодне секције на раније описаним референтним програмима. Једино на референтном програму *micronaut-shopcart* модификован алгоритам даје бољу употребу меморије кодне секције за 0.1 MB. Међутим, како је за све остале референтне програме разлика у склопу стандардне девијације ово одступање није значајно.

Табела 4.8 представља резултат 10 мерења времена првог одзива на раније описаним референтним програмима. Примећује се да су све разлике између

Табела 4.8: Време првог одзива с релативним стандардним девијацијама

Референтни програм	Оригинал	Модификација
spring-petclinic	279.4 ± 2.8 ms	278.3 ± 2.9 ms
micronaut-helloworld	50.9 ± 0.8 ms	54.0 ± 5.7 ms
micronaut-shopcart	56.9 ± 0.8 ms	56.4 ± 1.8 ms
quarkus-helloworld	41.8 ± 1.1 ms	42.1 ± 1.4 ms
quarkus-tika	65.6 ± 1.0 ms	64.9 ± 1.8 ms

оригиналног и модификованог алгоритма у распону њихових стандарддних девијација. То потврђује да измена алгоритма нема мерљив утицај на време првог одзива.

Тестирање није рађено на метрици резидентног скупа целог програма, јер је показано да је су разлике значајније на кодној секцији. Пропусност и кашњење су се испоставили стабилни у претходним секцијама и зато нису мерени.

Алгоритми засновани на профилима

Под претпоставком да је профајлирање извршено над репрезентативним улазним подацима, сматра се да методе које имају профиле представљају чешће и значајније путеве извршавања, док се методе без профила третирају као ређе коришћене. Уместо да се методе без профила оставе у насумичном редоследу, измењени алгоритми примењују сортирање по семантици имена (логика из алгоритма *SortByMethodNameSemantics*) над тим методама, како би се обезбедио стабилнији распоред у случају да профили нису потпуни или да улазни подаци током профајлирања нису били у потпуности репрезентативни.

Ова промена не утиче мерљиво на време потребно за компајлирање програма. Утицај промена не може да се измери текућим референтним програмима јер се у оквиру постојећих мерења те методе не извршавају и самим тим не утичу ни на вредности метрика.

Међутим, из евалуације алгоритама распоређивања метода може се разумно закључити да неки алгоритми производе ефикасније програме од других. Конкретно, поређење алгоритама *ChooseRandom* и *SortByMethodNameSemantics* показује да алгоритам *SortByMethodNameSemantics* генерално доводи до бољих перформанси. Значајност промене редоследа метода без профила може се илустровати следећим примером: нека је R' потпрограм програма R који се састоји

искључиво из метода без профила, а M' његова улазна тачка, од које се даље извршавају само методе без профила. Тада се очекује да ће програм R бити бржи ако се за R' користи алгоритам *SortByMethodNameSemantics*.

Глава 5

Закључак

У оквиру овог рада анализирани су и поређени различити алгоритми распоређивања метода у компајлеру *GraalVM Native Image*, са фокусом на њихов утицај на перформансе апликација. Истраживање је обухватило три категорије алгоритама: алгоритме са насумичним распоређивањем метода који служе као основа за поређење, хеуристичке алгоритме засноване на статичким карактеристикама метода, и алгоритме вођене профилом програма. Резултати експерименталне евалуације јасно показују да распоред метода у извршивој датотеци има значајан утицај на перформансе, највише на меморијску потрошњу и време покретања.

Референтни алгоритам за поређење је алгоритам *ChooseRandom*, коме су методе распоређене у насумичном редоследу. Алгоритам *ChooseRandomOnePerPage*, дизајниран да максимизује промашаје страница, демонстрирао је драстично лошије резултате по свим метрикама, што потврђује важност просторне локалности за ефикасност програма.

Алгоритми засновани на хеуристикама, *SortByMethodName* и *SortByMethodNameSemantics* показали су приметна побољшања у односу на референтни алгоритам са насумичним распоређивањем метода, што указује на то да чак и једноставне стратегије засноване на статичким карактеристикама метода могу допринети бољим перформансама програма. Напреднија хеуристика која укључује семантичку анализа метода дала је нешто боље резултате, што потврђује вредност детаљније анализе структуре програма.

Најзначајнија побољшања постигнута су применом алгоритама вођених профилом програма. Сви профилом вођени алгоритми показали су значајна побољшања у меморијским метрикама, са редукцијом потрошње 11-13% за укупну

меморију и 42-58% за меморију кодне секције.

Посебно се издваја алгоритам *SortByFirstCall*, који је у свим метрикама дао најбоље резултате, демонстрирајући да је време првог позива методе кључни фактор оптимизација распореда метода. Резултати за метрике кашњења и пропусности показали су да распоред метода има ограничен утицај на перформансе у стабилном стању, када је систем већ загрејан и све компоненте су учитане у меморију. Ово указује на то да су оптимизације распореда метода најзначајније за фазу покретања и ране фазе извршавања апликације.

Инфраструктура развијена у оквиру овог рада омогућава детаљнију анализу меморијског отиска кодних секција. То је значајно за будућа истраживања у области оптимизације распореда метода.

Практични значај овог рада лежи у томе што пружа квантитативну евалуацију различитих стратегија распореда метода у савременом компајлеру *GraalVM Native Image*. Резултати сугеришу да би алгоритам *SortByFirstCall* могао постати подразумевани избор за распоред метода у овом систему, обезбеђујући оптималне перформансе за већину апликација. Додатно, представљене су и имплементације за побољшања алгоритама *SortByMethodNameSemantics*, *SortByHotness*, *ClusterByEdges*, *ClosestIsBest* и *SortByFirstCall*.

Такође, отвара се простор за будућа истраживања која би могла комбиновати предности различитих приступа, на пример, спајајући информације о времену првог позива са подацима о учесталости позива метода. У контексту све већег значаја AOT компајлирања за окружења која се извршавају у облаку и микросервисне архитектуре, ово истраживање доприноси развоју ефикаснијих система који могу обезбедити брже покретање и мању потрошњу ресурса.

Закључно, овај рад демонстрира да је распоред метода важан фактор за перформансе програма и да оптимизације вођене профилом програма обезбеђују побољшање просторне локалности и смањење меморијског отиска. Постоје неистражене могућности за комбиновање различитих стратегија, тако да тренутни резултати пружају јасне смернице за практичну примену и будућа истраживања у овој области.

Библиографија

[1] Barista Benchmark Suite: An Open Source Microservices Benchmark on the JVM Platform. `https://github.com/Payara/Barista`, 2023. Evaluates JVM-based microservices performance in JIT and AOT (native-image) execution modes using GraalVM capabilities.

[2] Apache maven. `https://maven.apache.org/`, n.d. The Apache Software Foundation.

[3] Gradle build tool. `https://gradle.org/`, n.d. Gradle Inc.

[4] Micronaut framework. `https://micronaut.io/`, n.d. Object Computing, Inc.

[5] Quarkus: Supersonic subatomic java. `https://quarkus.io/`, n.d. Red Hat.

[6] Spring framework. `https://spring.io/projects/spring-framework`, n.d. Pivotal Software, Inc.

[7] Apache. Apache Tika Toolkit. on-line at: `https://tika.apache.org/`.

[8] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 169–190, 2006.

[9] Ellis Hoag. RFC: Temporal Profiling Extension for IRPGO. on-line at: `https://discourse.llvm.org/t/rfc-temporal-profiling-extension-for-irpgo/68068`.

[10] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. Challenges for static analysis of java reflection - literature review and empirical study. In *2017*

IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 507–518, May 2017.

[11] Sun Microsystems. The Java HotSpot Performance Engine Architecture. on-line at: `https://www.oracle.com/java/technologies/whitepaper.html`.

[12] Oracle. GraalVM. on-line at: `https://www.graalvm.org/`.

[13] Oracle. Profile Guided Optimization. on-line at: `https://www.graalvm.org/jdk24/reference-manual/native-image/optimizations-and-performance/PGO/`.

[14] Oracle. The javac Command. on-line at: `https://docs.oracle.com/en/java/javase/23/docs/specs/man/javac.html`.

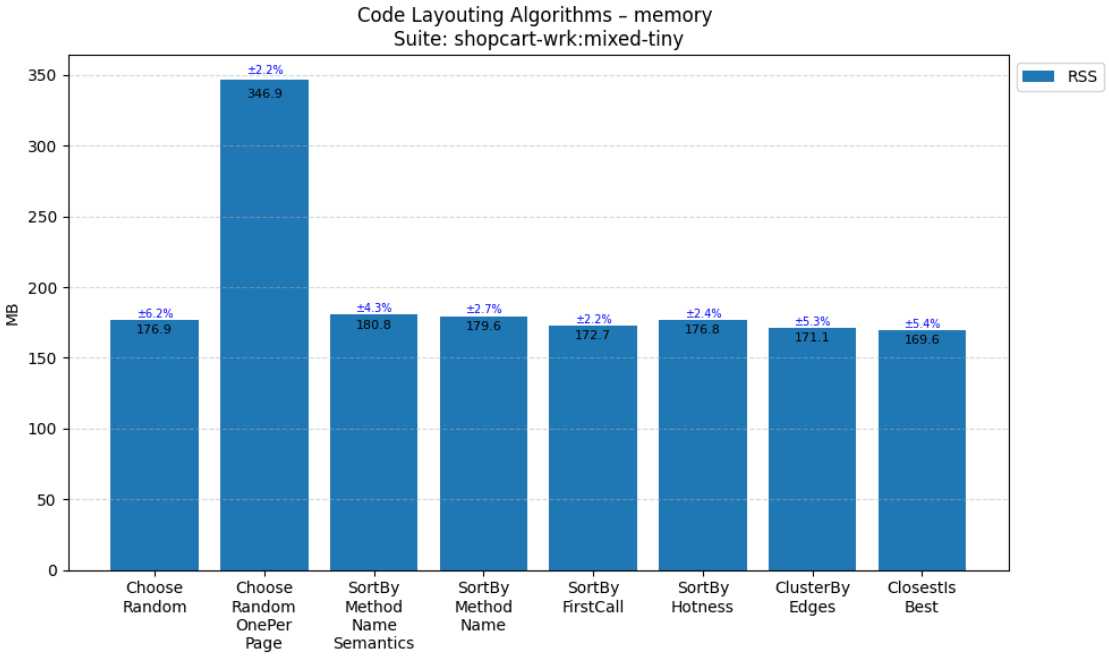
[15] Karl Pettis and Robert C Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 16–27, 1990.

[16] Piotr Plecinski, Nataliia Bokla, Tamara Klymkovych, Mykhailo Melnyk, and Wojciech Zabierowski. Comparison of Representative Microservices Technologies in Terms of Performance for Use for Projects Based on Sensor Networks. *Sensors*, 22(20):7759, 2022.

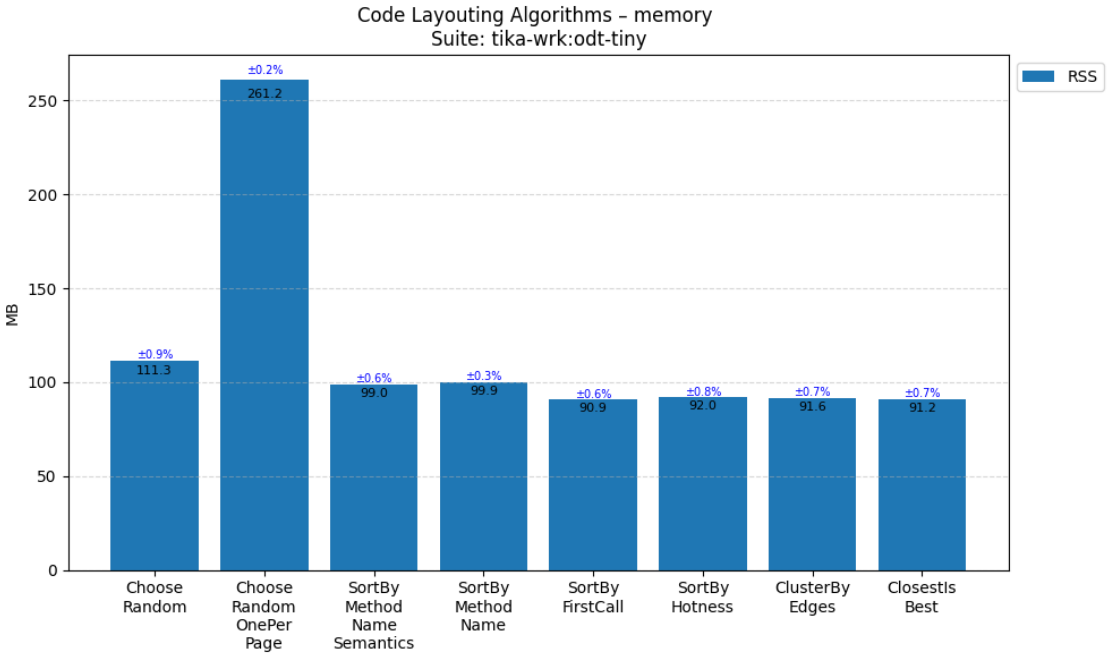
[17] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. Initialize once, start fast: application initialization at build time. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

Додатак А

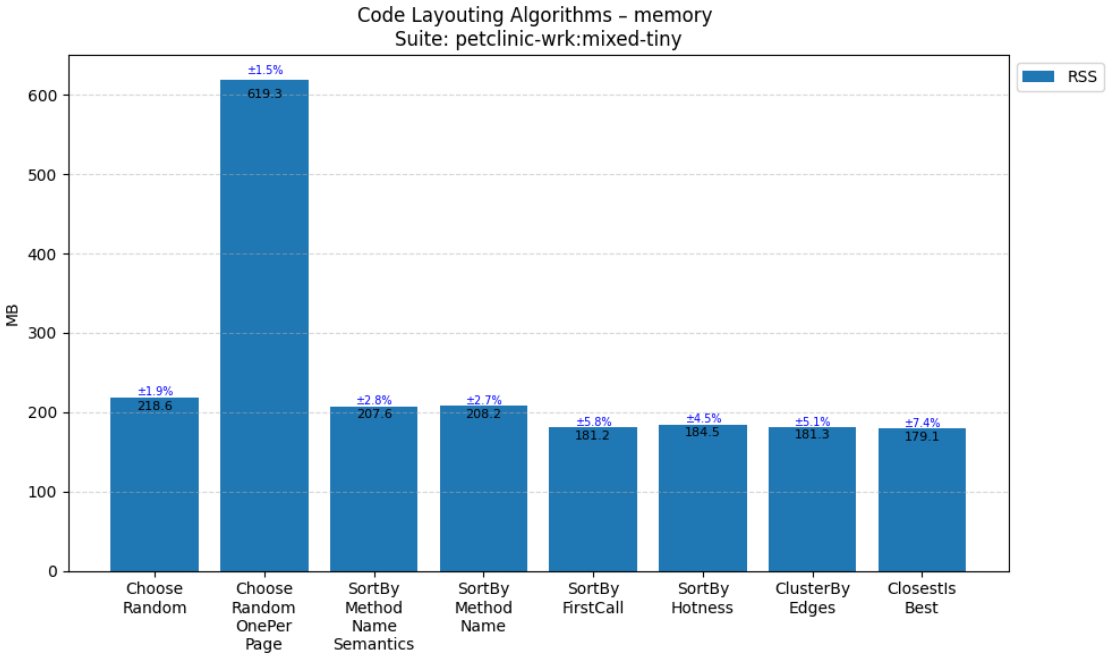
Додатне слике и резултати



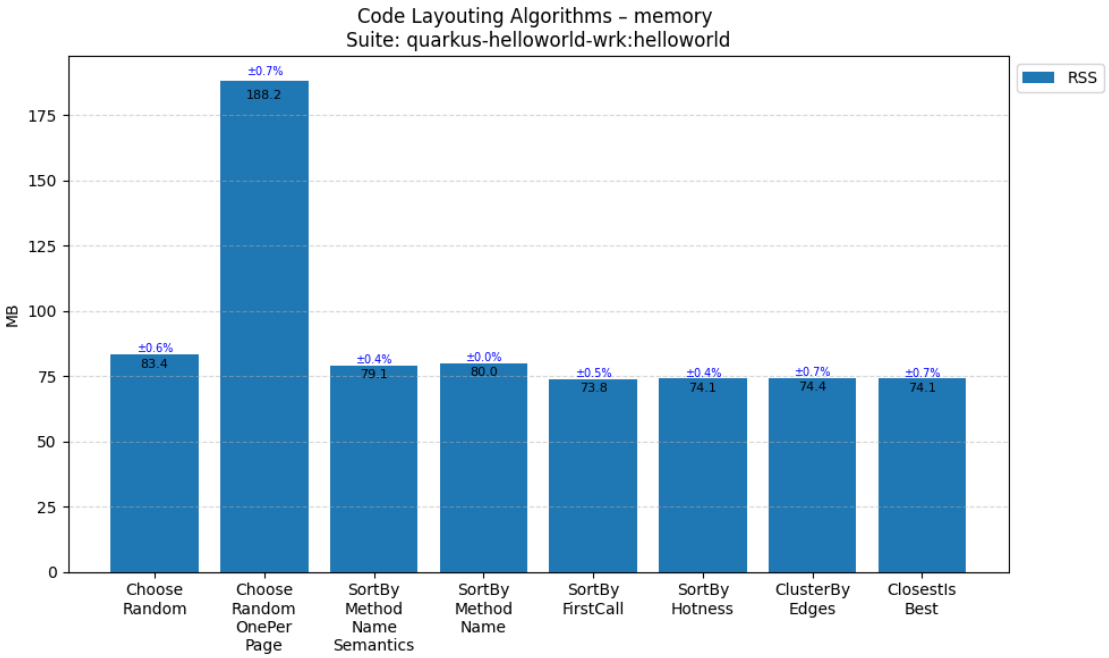
Слика А.1: Резултат мерења резидентног скупа за апликацију *Shopcart*



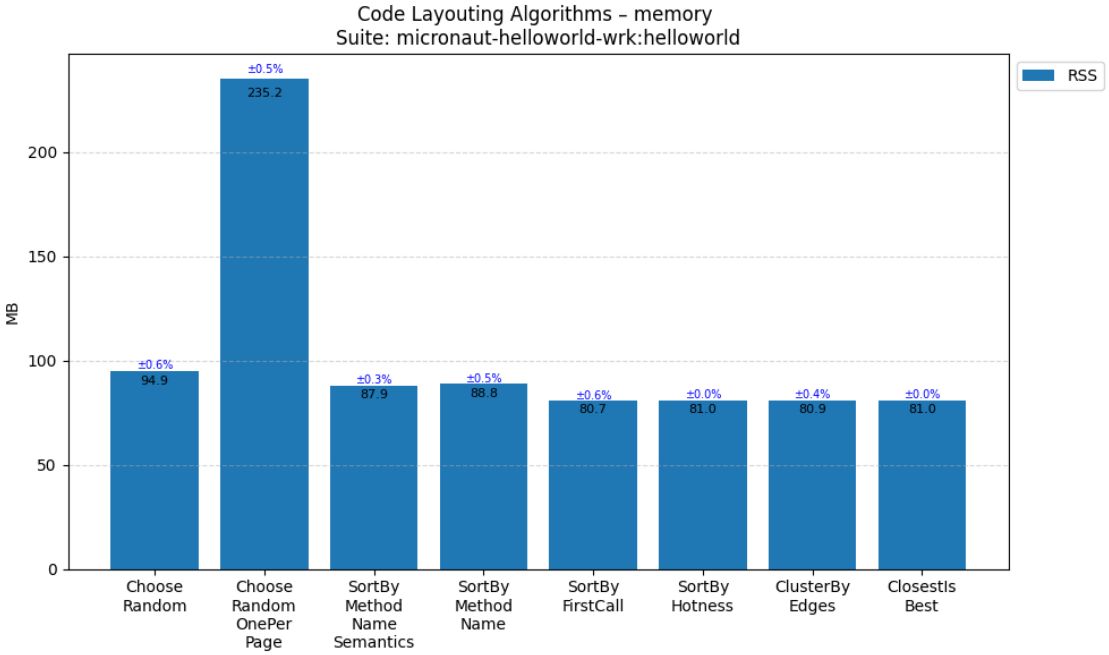
Слика А.2: Резултат мерења резидентног скупа за апликацију *Tika*



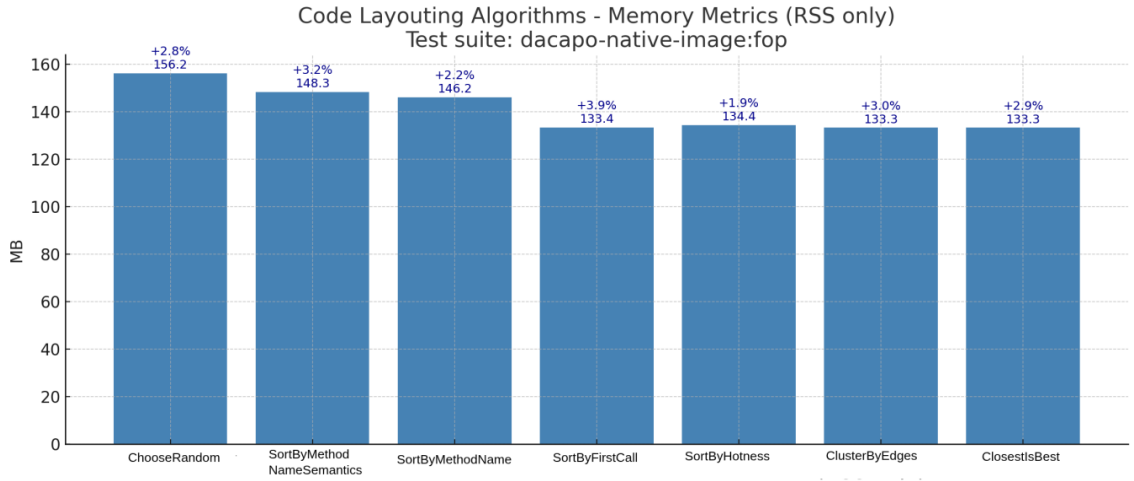
Слика А.3: Резултат мерења резидентног скупа за апликацију *Petclinic*



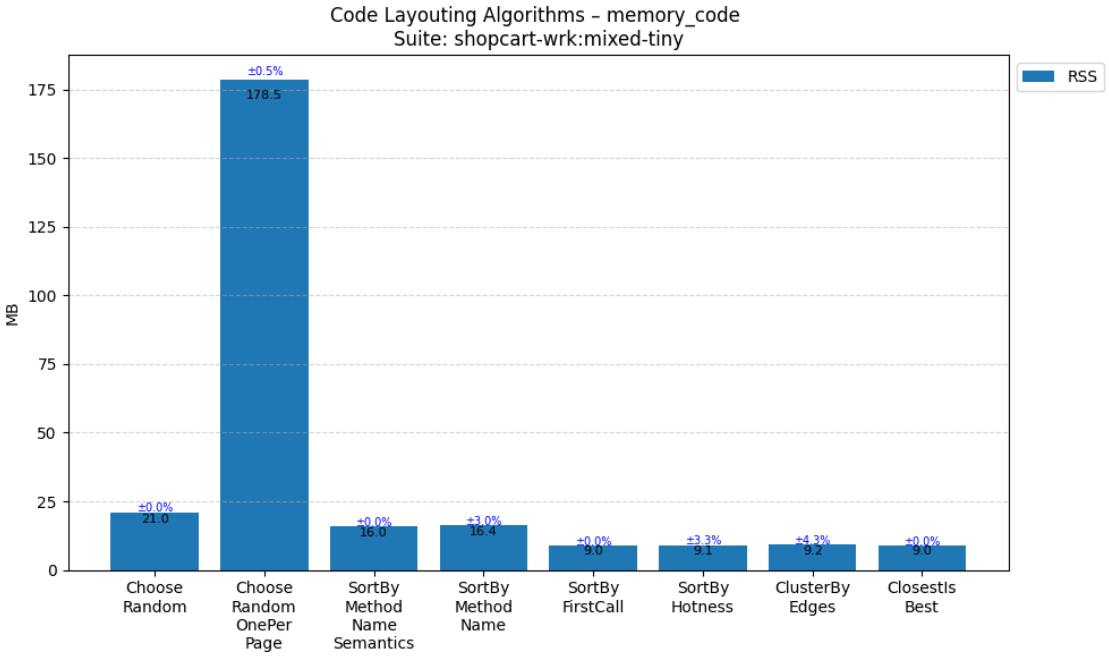
Слика А.4: Резултат мерења резидентног скупа за апликацију *Helloworld* коришћењем радног оквира *Quarkus*



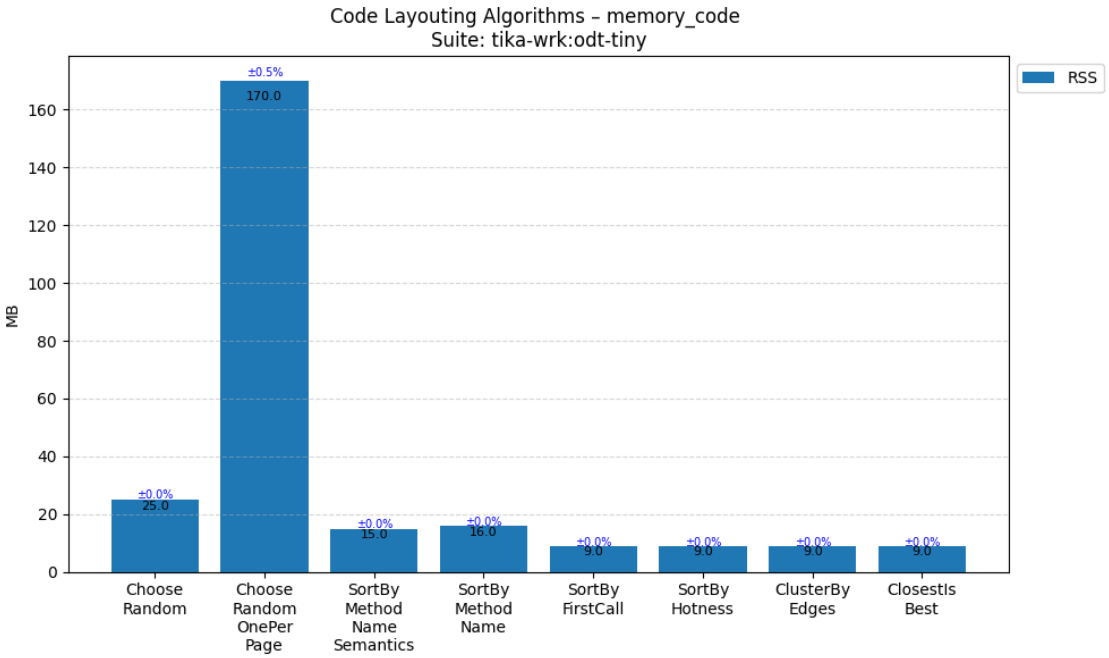
Слика А.5: Резултат мерења резидентног скупа за апликацију *Helloworld* коришћењем радног оквира *Micronaut*



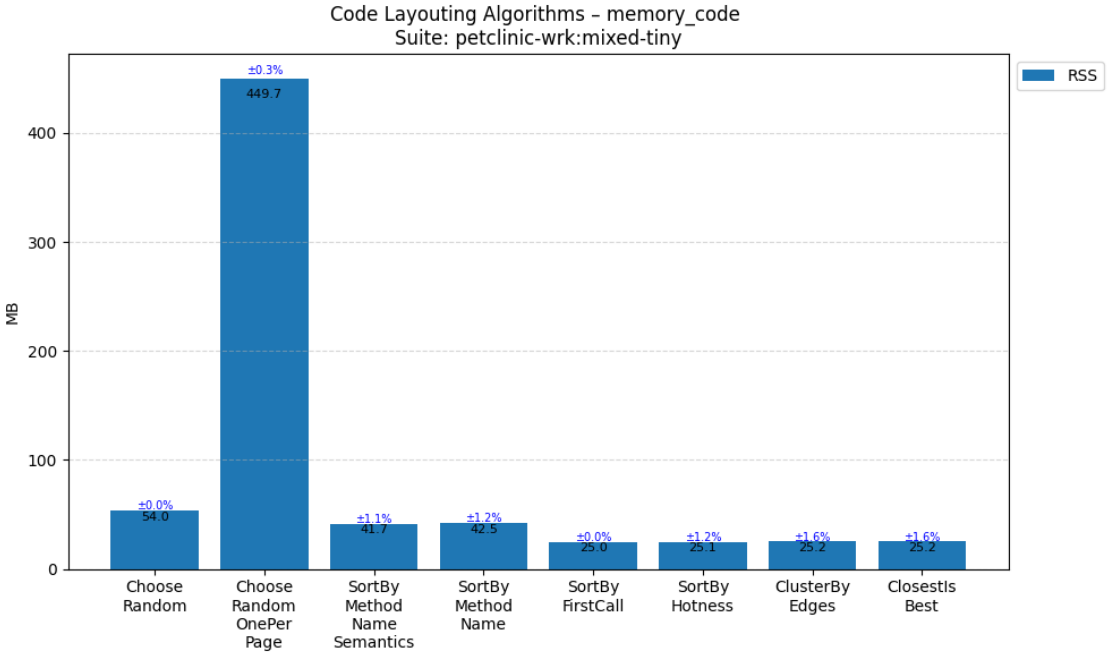
Слика А.6: Резултат мерења резидентног скупа за апликацију *Dacapo*



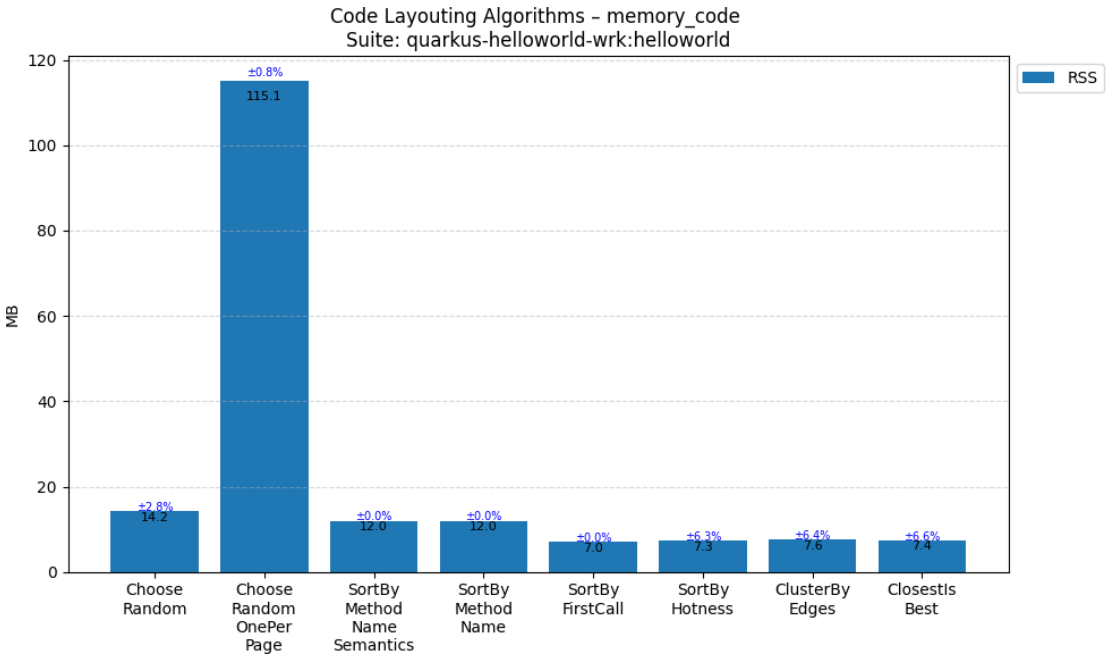
Слика А.7: Резултат мерења резидентног скупа секције кода за *Shopcart* апликацију



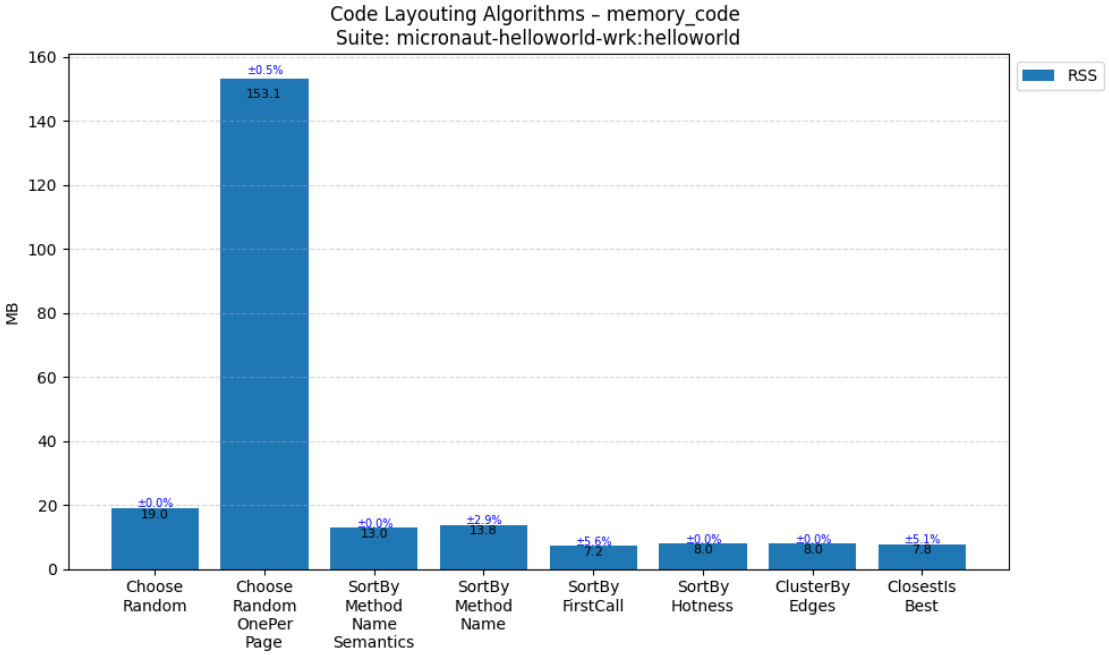
Слика А.8: Резултат мерења резидентног скупа секције кода за апликацију *Tika*



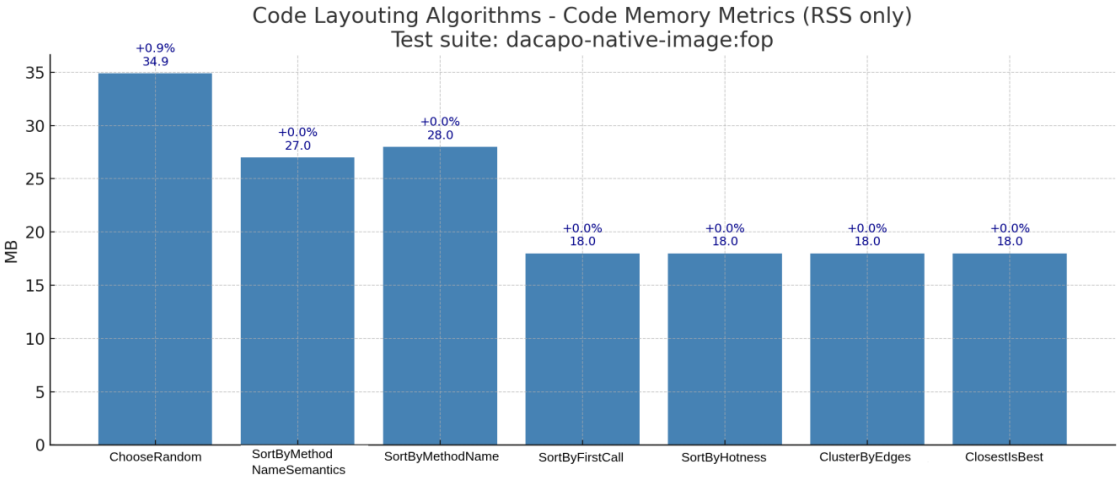
Слика А.9: Резултат мерења резидентног скупа секције кода за апликацију *Petclinic*



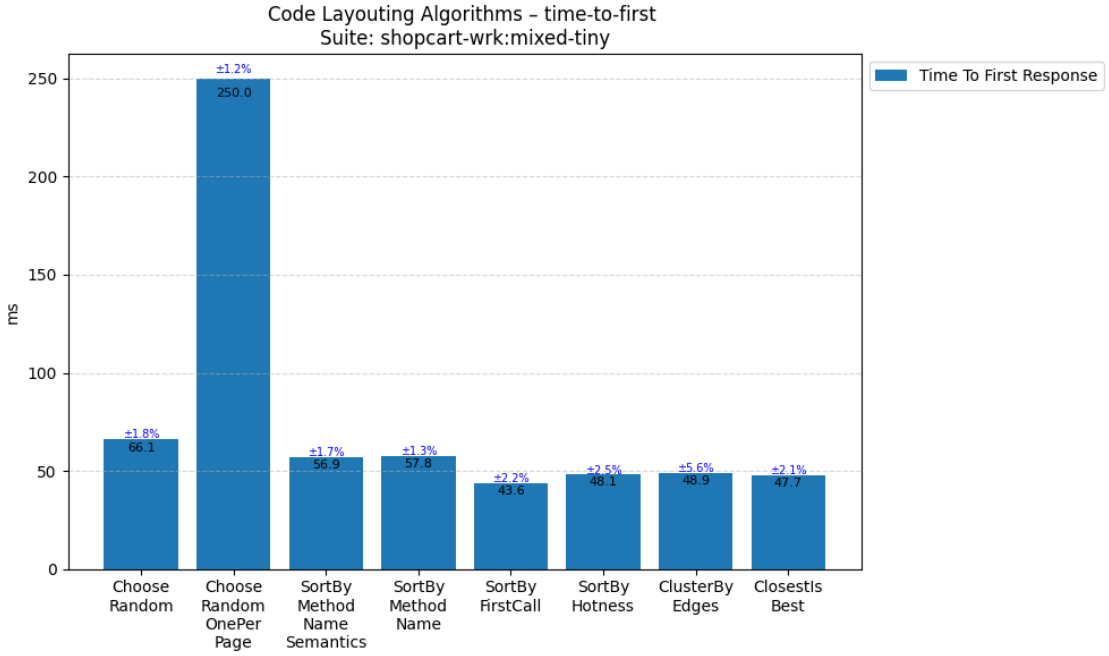
Слика А.10: Резултат мерења резидентног скупа секција кода за апликацију *Helloworld* коришћењем радног оквира *Quarkus*



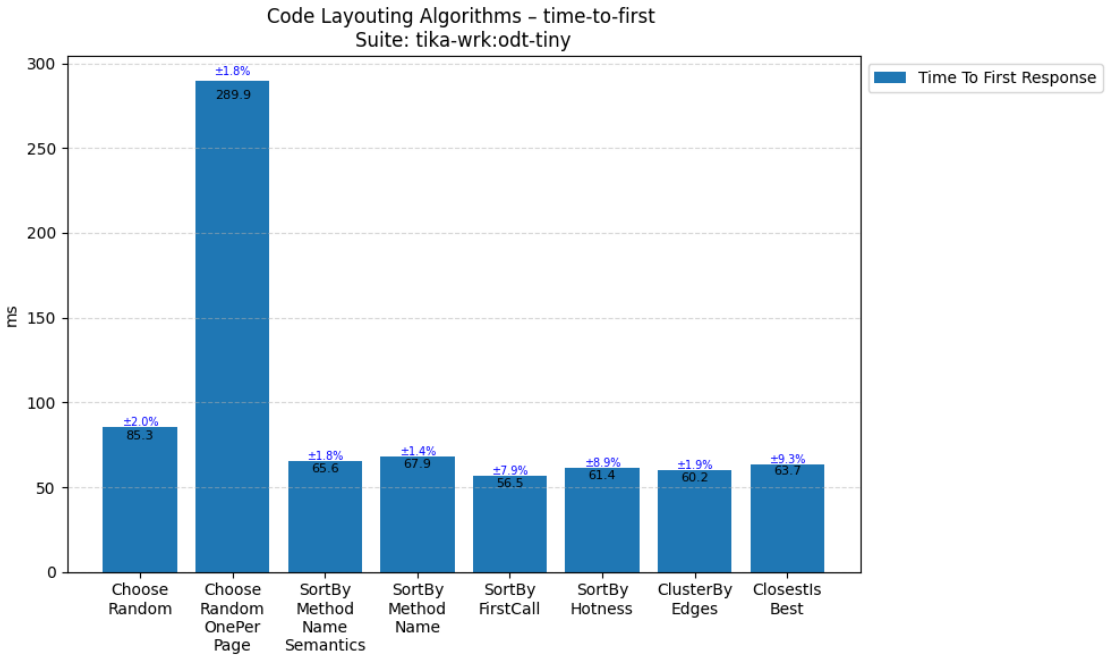
Слика А.11: Резултат мерења резидентног скупа секције кода за апликацију *Helloworld* коришћењем радног оквира *Micronaut*



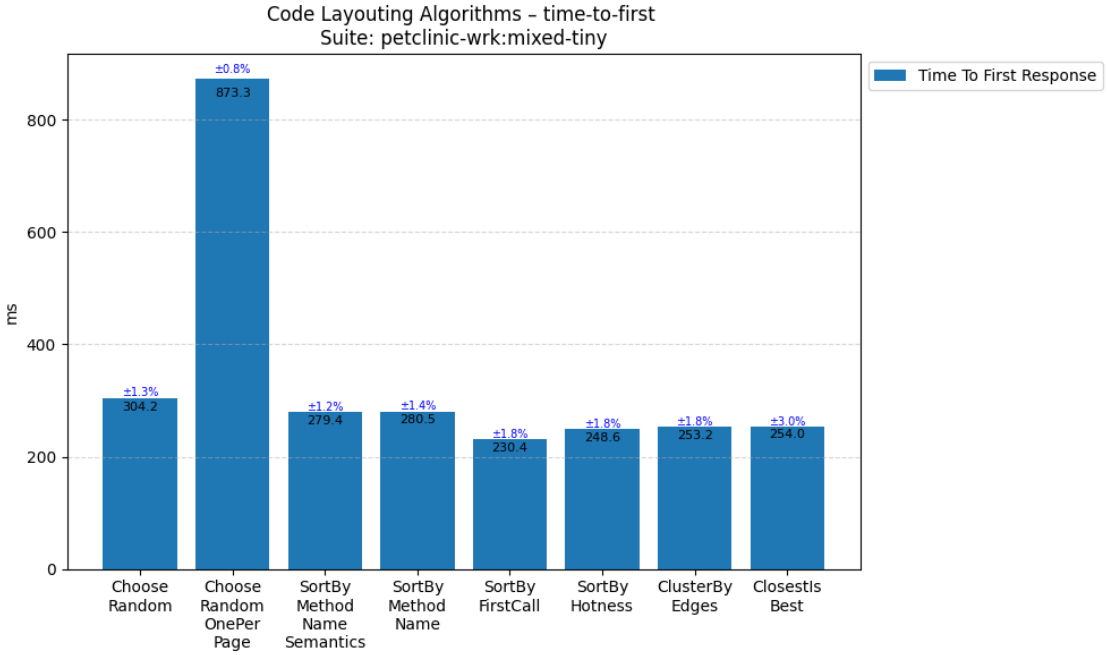
Слика А.12: Резултат мерења резидентног скупа секције кода за апликацију *Dacapo*



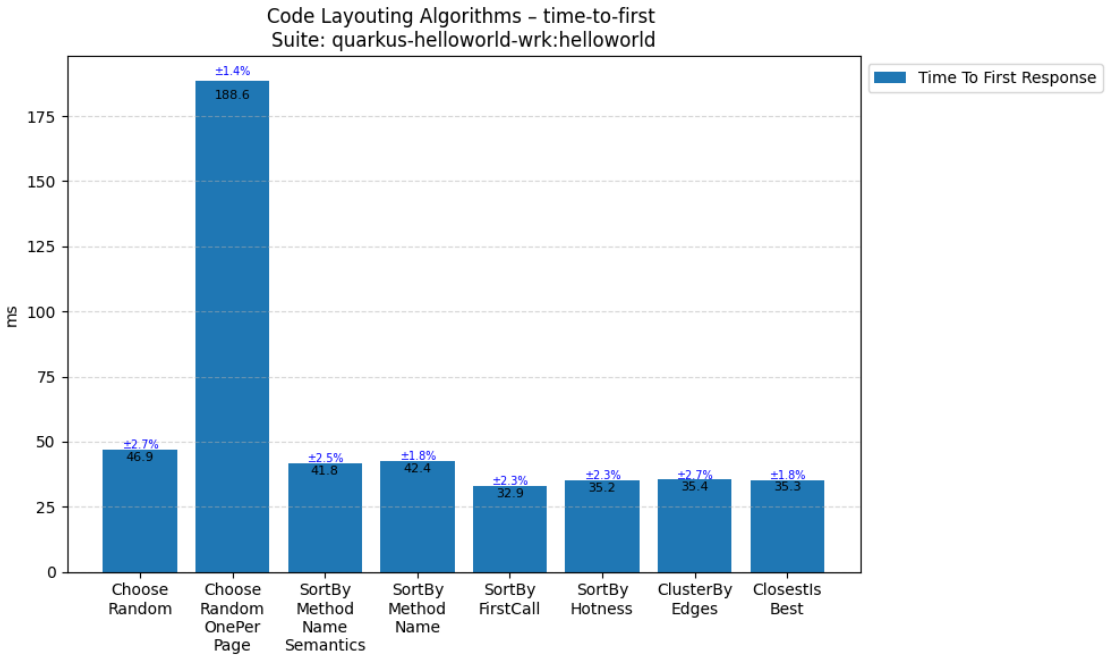
Слика А.13: Резултат мерења времена првог одзива за апликацију *Shopcart*



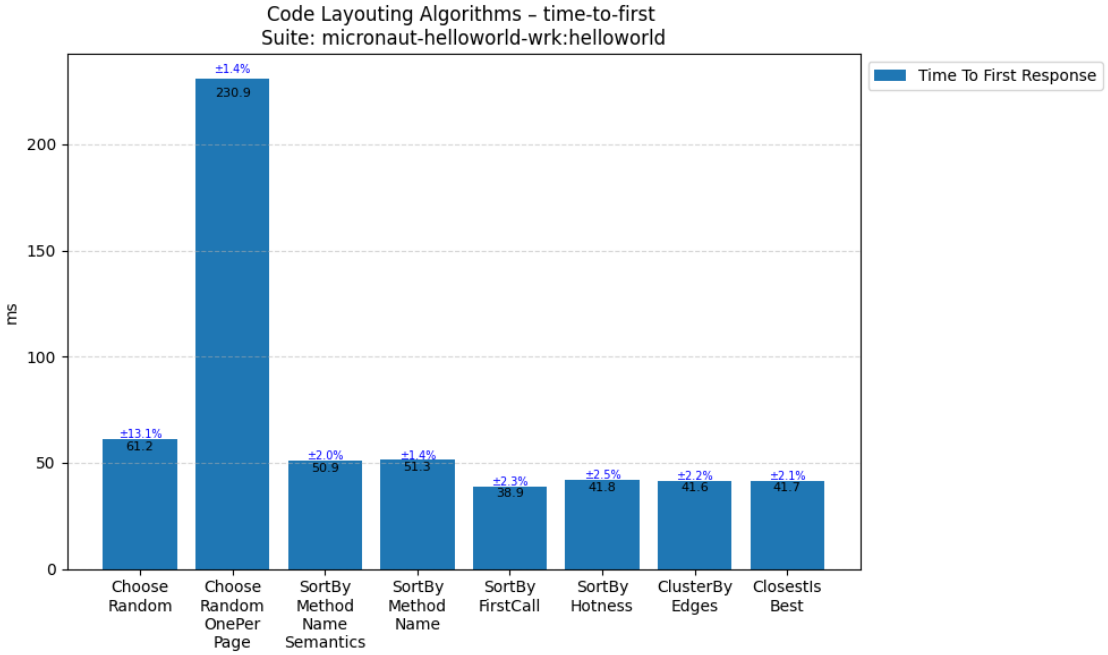
Слика А.14: Резултат мерења времена првог одзива за апликацију *Tika*



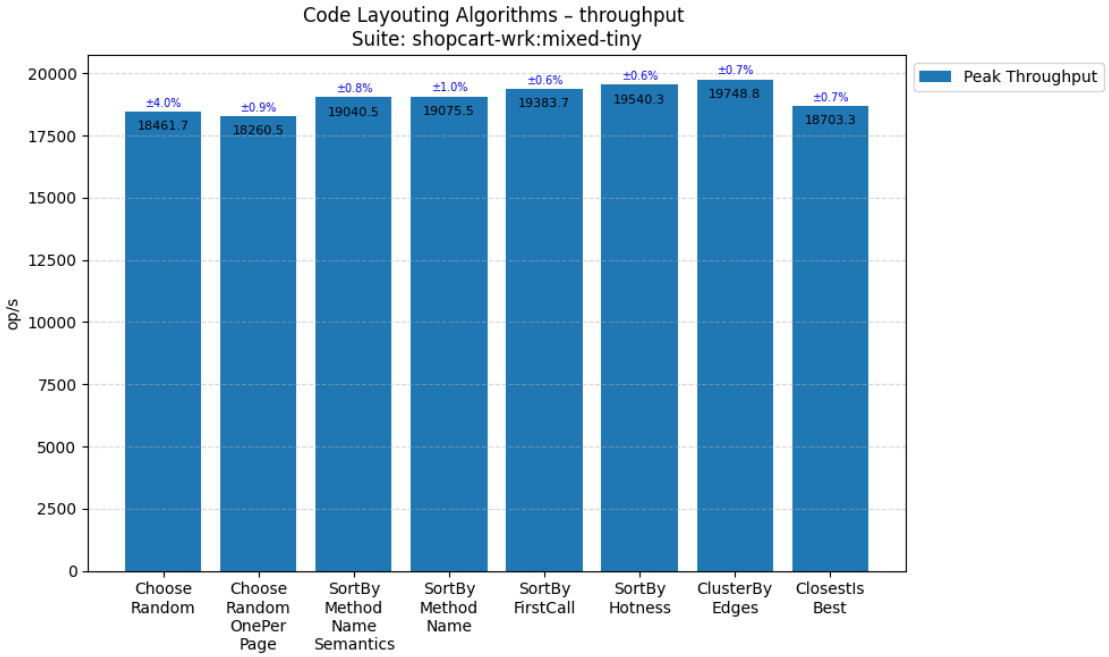
Слика А.15: Резултат мерења времена првог одзива за апликацију *Petclinic*



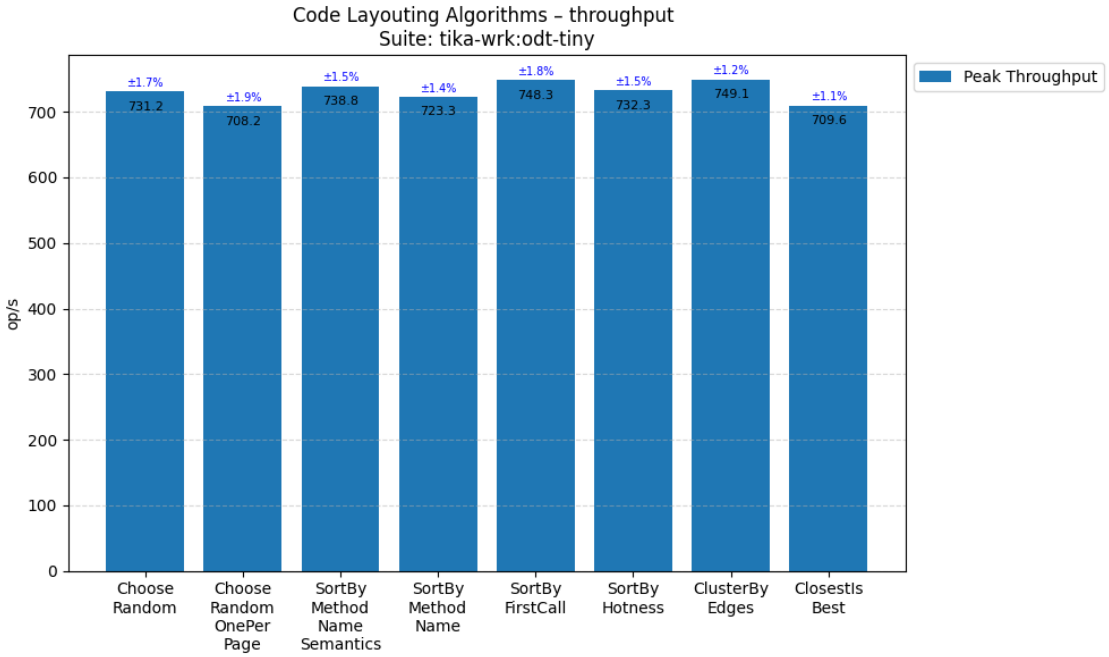
Слика А.16: Резултат мерења времена првог одзива за апликацију *Helloworld* коришћењем радног оквира *Quarkus*



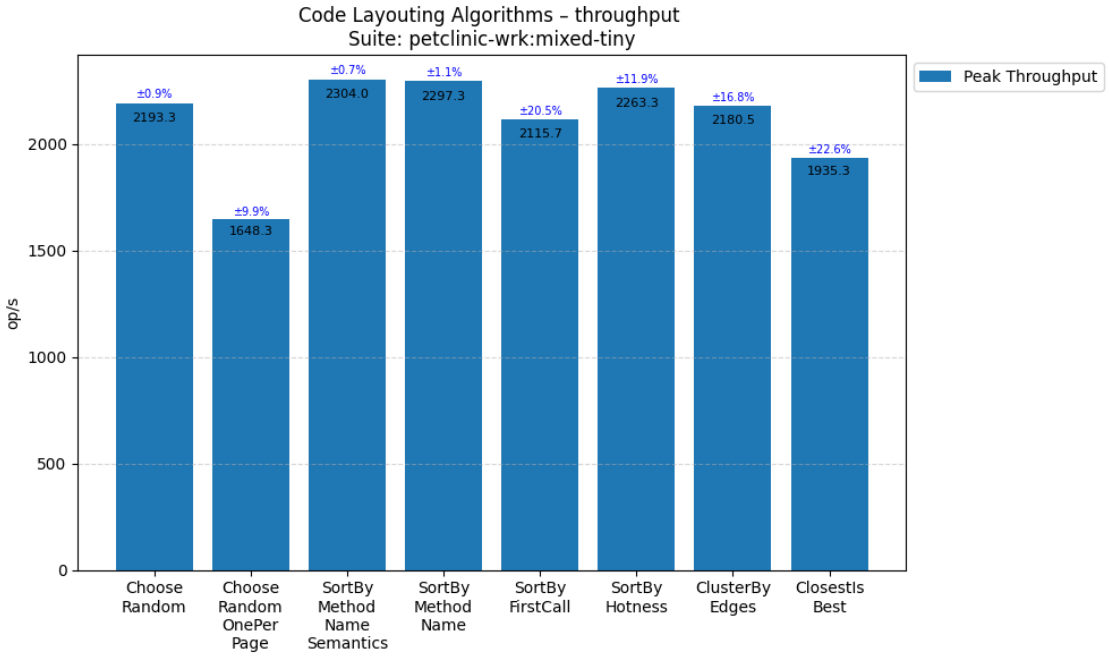
Слика А.17: Резултат мерења времена првог одзива за апликацију *Helloworld* коришћењем радног оквира *Micronaut*



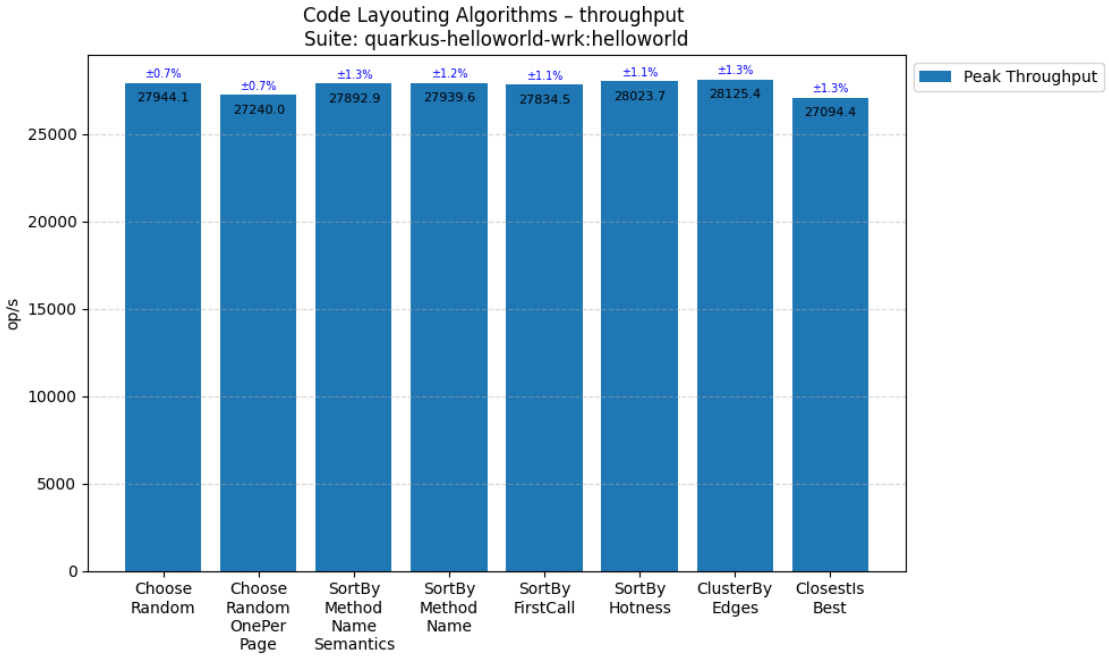
Слика А.18: Резултат мерења пропусности за апликацију *Shopcart*



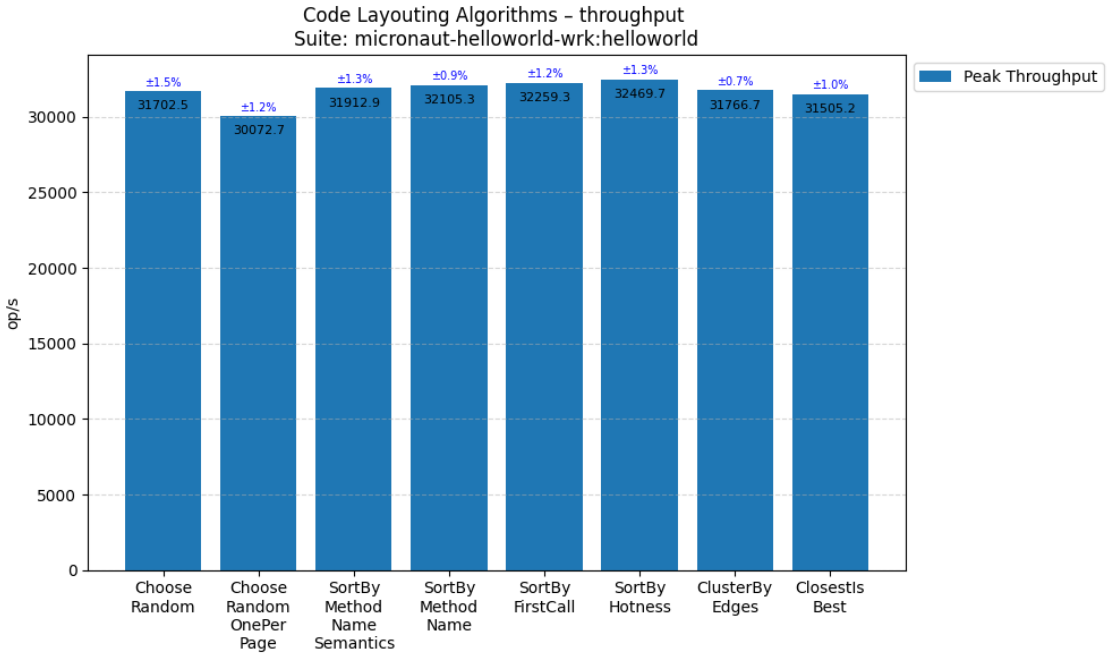
Слика А.19: Резултат мерења пропусности за апликацију *Tika*



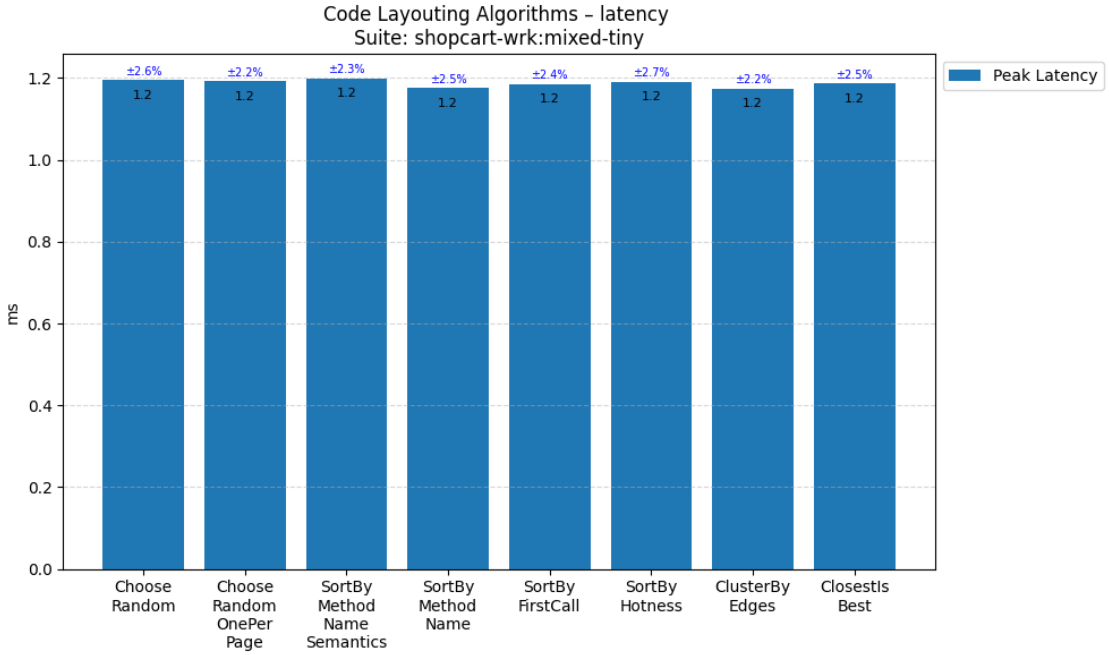
Слика А.20: Резултат мерења пропусности за апликацију *Petclinic*



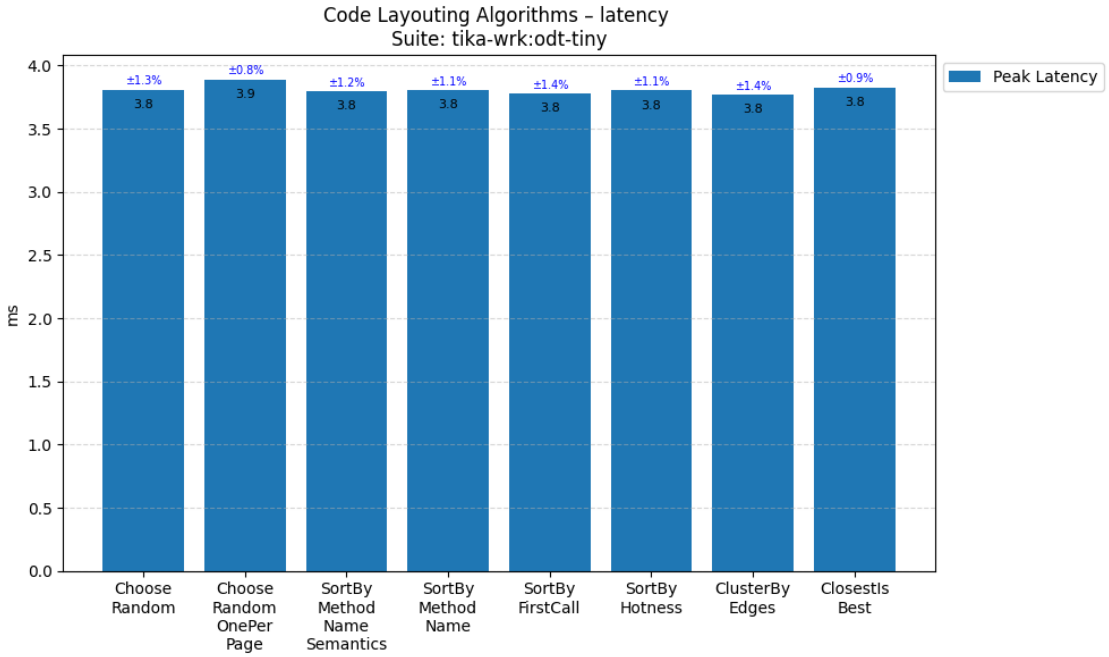
Слика А.21: Резултат мерења времена пропусности за апликацију *Helloworld* коришћењем радног оквира *Quarkus*



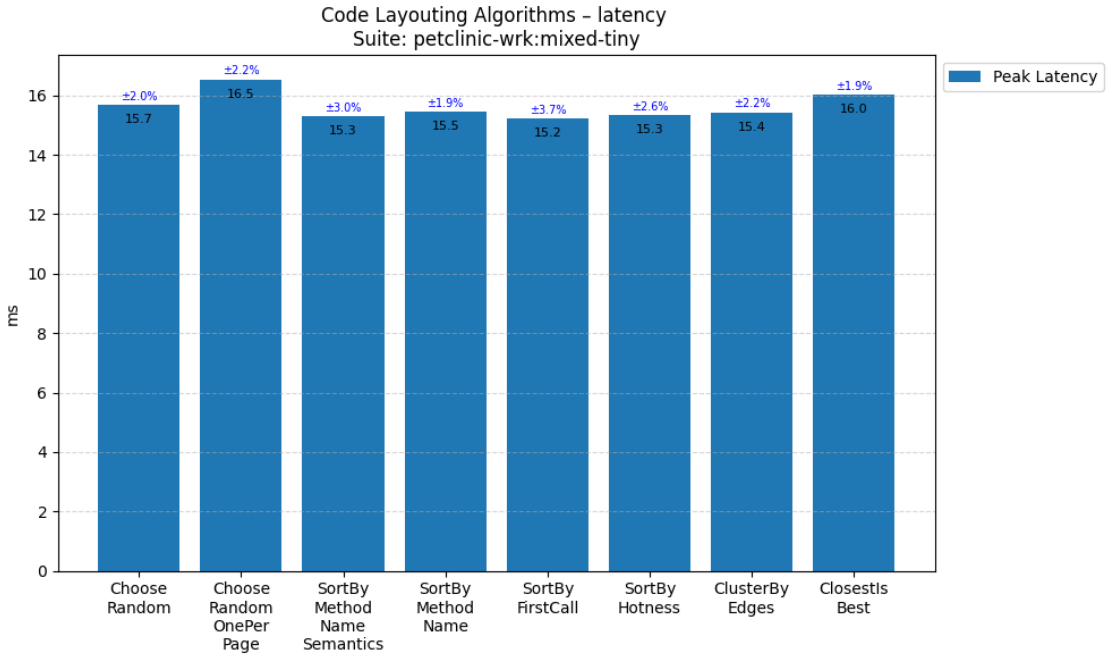
Слика А.22: Резултат мерења пропусности апликацију *Helloworld* коришћењем радног оквира *Micronaut*



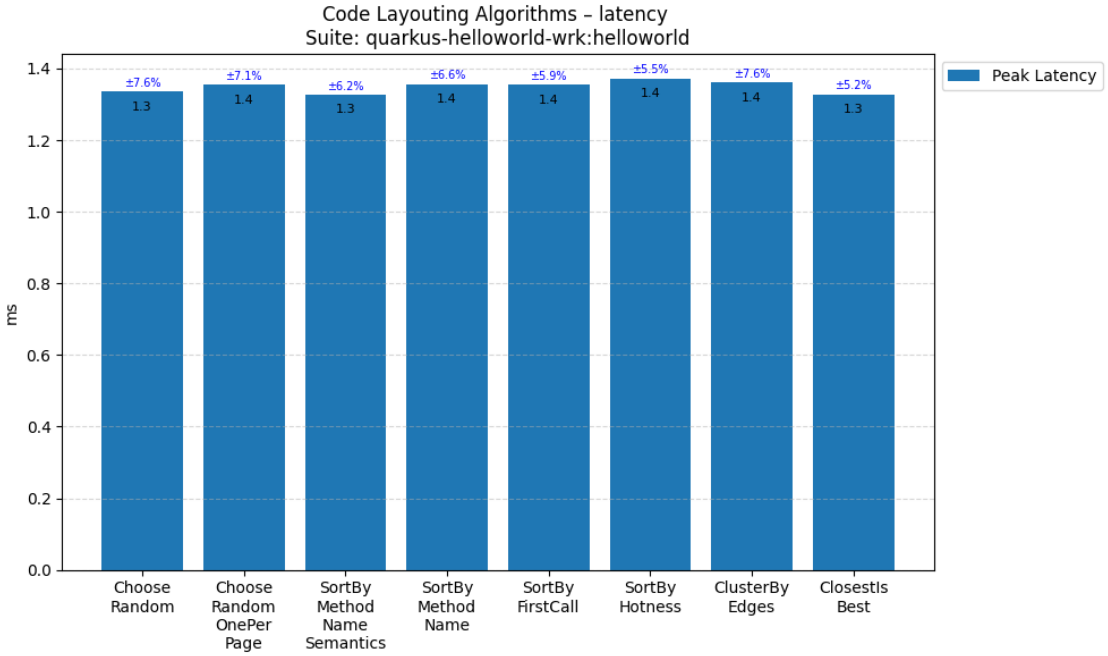
Слика А.23: Резултат мерења кашњења за апликацију *Shopcart*



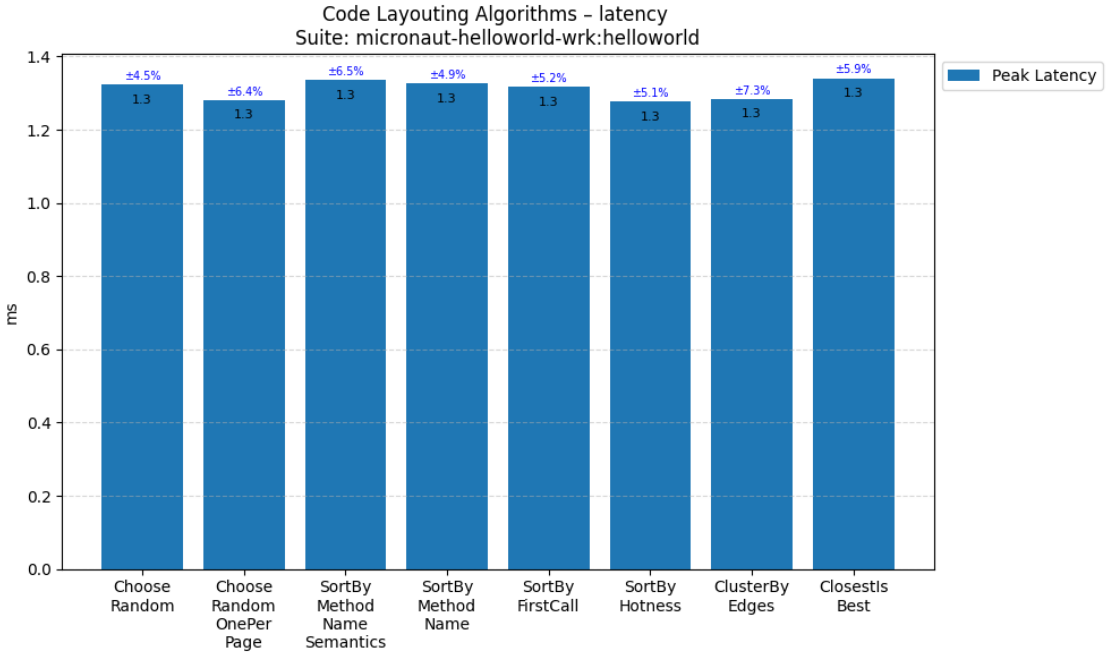
Слика А.24: Резултат мерења кашњења за апликацију *Tika*



Слика А.25: Резултат мерења кашњења за апликацију *Petclinic*



Слика А.26: Резултат мерења кашњења за апликацију *Helloworld* коришћењем радног оквира *Quarkus*



Слика А.27: Резултат мерења кашњења за апликацију *Helloworld* коришћењем радног оквира *Micronaut*

Биографија аутора

Милоје Јоксимовић рођен је 21. марта 1998. године у Шапцу. Основну школу започиње у ОШ „Ната Јелићић“ у Шапцу, наставља у ОШ „Николај Велимировић“ у истом граду, а потом уписује и завршава Математичку гимназију у Београду. Године 2017. уписује Математички факултет Универзитета у Београду, смер Рачунарство и информатика, где је дипломирао у јануару 2021. године са просечном оценом 9.8.

Након завршетка студија запошљава се у *Microsoft Development Center Serbia*, где до октобра 2024. године ради углавном као дата инжењер на развоју производа *Bing Maps*. Истовремено, у октобру 2024. године уписује мастер студије на Математичком факултету и бива изабран за сарадника у настави. Након тога, у марту 2025. почиње истраживачку праксу у компанији *Oracle Labs*. Тренутно изводи наставу на предметима *Увод у Пајтон* на Биолошком факултету, као и на предметима *Програмске парадигме*, *Рачунарске мреже*, *Програмирање 2* и *Објектно оријентисано програмирање* на Математичком факултету.