УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Милица Карличић

ПОБОЉШАЊЕ ЕФИКАСНОСТИ КЛИЈЕНТ-СЕРВЕР АПЛИКАЦИЈА ДИНАМИЧКИМ ПОДЕШАВАЊЕМ ПАРАМЕТАРА СКУПЉАЧА ОТПАДАКА У СИСТЕМУ GRAALVM

мастер рад

Београд, 2024.

Ментор:

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

Чланови комисије:

проф. др Филип Марић, редовни професор
Универзитет у Београду, Математички факултет

доц. др Весна Маринковић, доцент
Универзитет у Београду, Математички факултет

Датум одбране: Септембар, 2024.

Захвалница

Захваљујем се менторки свог рада, проф. др Милени Вујошевић Јаничић, на свим сугестијама, правовременим коментарима и подршци када су ми били најпотребнији. Хвала јој што ми је огромна подршка на факултету као и на истраживачкој пракси у компанији Oracle Labs.

Захваљујем се компанији Oracle Labs, посебно Војину Јовановићу на потребним ресурсима и прилици да радим са најбољим тимом. Посебну захвалност дугујем колеги асистенту Ивану Ристовићу на стрпљењу, свим савјетима, инспирацији, помоћи у вези са имплементацијом као и у вези са писањем самог рада.

Највећу захвалност дугујем својој породици која је увијек била уз мене. Хвала вам на разумијевању, подршци и мотивацији.

Наслов мастер рада: Побољшање ефикасности клијент-сервер апликација динамичким подешавањем параметара скупљача отпадака у систему *GraalVM*

Резиме: Скупљач отпадака (енгл. *garbage collector*) је механизам који у фази извршавања програма аутоматски идентификује и ослобађа меморију објеката који више нису у употреби. Програмски језици који се ослањају на скупљач отпадака олакшавају развој апликација, омогућавајући програмерима да се фокусирају на логику програма умјесто на управљање меморијом. Међутим, скупљач отпадака утиче неповољно на перформансе програма јер већина имплементација функционише по механизму комплетног заустављања извршавања ради ослобађања меморије (енг. *stop the world*). У клијент-сервер апликацијама сервер обрађује захтјев који је послат од стране клијента. Узастопни захтјеви су најчешће слични и та повезаност се може искористити за прилагођавање скупљача отпадака наредним захтјевима у циљу смањења фреквенција позивања самог скупљача. На тај начин се убрзава обрада захтјева, повећава број обрађених захтјева у секунди и самим тим побољшавају перформансе програма. Циљ рада је пратити и анализирати употребу меморије у захтјевима клијент-сервер апликације у облаку ради ефикаснијег скупљања отпадака. На основу анализе историје захтјева, величина младе генерације се динамички подешава тако да се избјегне скупљање отпадака током захтјева, или тако да се фаворизује скупљање које је брже и мање захтјевно од комплетног. Имплеменација решења је урађена у оквиру система *GraalVM* и евалуирана на стандардном скупу референтних програма. Добијени резултати показују значајна смањења меморијског заузећа и максималног кашњења евалуираних апликација.

Кључне речи: програмски језик Јава, клијент-сервер апликације, скупљач отпадака, инкрементално и комплетно скупљање, систем *GraalVM*

Садржај

Глава 1

Увод

Аутоматско управљање меморијом присутно је код великог броја модерних програмских језика како би програмирање било удобније. Оно подразумијева брисање објеката када више нису потребни програму тј. када више нико не реферише на њих. Ослобађање меморије врши скупљач отпадака [14]. Већина скупљача отпадака заустави програм док врши чишћење меморије што значи да његово позивање неповољно утиче на перформансе апликације. У циљу побољшања перформанси, потребно је прилагодити скупљач датој апликацији како би извршавање било ефикасније.

Клијент-сервер апликације представљају модел дистрибуираних апликација у којима се задаци дијеле између сервера који обезбјеђују ресурсе или услуге и клијената који траже те услуге [16]. Клијенти и сервери најчешће комуницирају преко компјутерске мреже на одвојеном хардверу, али могу бити смјештени и у истом систему. Клијенти и сервери размењују поруке у виду комуникацијског шаблона **захтјев-одговор**: клијент пошаље захтјев, а сервер врати одговор. Најчешће, захтјеви су међусобно слични и независни што се може искористити на различите начине како би се побољшале перформансе оваквих апликација.

У оквиру овог рада развијена је методологија у систему *GraalVM* [6] која има за циљ да прилагоди начин скупљања отпадака у клијент-сервер апликацијама ради побољшања ефикасности таквих апликација. Скупљач отпадака система *GraalVM* је генерацијски тј. објекте распоређује у оквиру младе и старе генерације на основу животног вијека. Промоција објекта подразумијева пребацивање објекта из једне генерације у другу или једног дијела једне генерације у други дио. Промовисање објеката кроз младу генерацију је јеф-

тиније јер је млада генерација у општем случају значајно мања од старе. Идеја система је максимално искористити простор младе генерације како би промовисање било брже и једноставније. Како су захтјеви клијент-сервер апликација слични и међусобно независни то се величина младе генерације динамички може прилагодити за смјештање објеката који су алоцирани у појединачним захтјевима.

Рад је организован на следећи начин. Поглавље 2 описује скупљаче отпадака и алгортиме по којима скупљачи раде, са посебном пажњом посвећеном начину на који ради Јавин скупљач отпадака. У поглављу 3 детаљно је описан систем *GraalVM* и његове компоненте. У поглављу 4 описана је имплементација система за ефикасније скупљање отпадака у клијент-сервер апликацијама као и начин на који систем може да се користи приликом превођења Јава апликација. Поглавље 5 садржи резултате тестирања клијент-сервер апликација добијене коришћењем имплементираног система. На крају, у поглављу 6 изнијет је закључак рада.

Глава 2

Скупљач отпадака

Хип (енг. *heap*) је дио меморије у којем се складиште објекти који се динамички алоцирају. Објекти који се алоцирају на хипу нису именовани већ им се приступа искључиво преко адреса. У случају програмских језика са аутоматским управљањем меморијом, објекат на који више нико не реферише аутоматски се ослобађа. Ова операција ослобађања зове се **скупљање** или **колекција отпадака** (енг. *garbage collection*). Већина скупљача отпадака (енг. *garbage collector*) ради по механизму заустављања свијета (енг. *stop the world*). То значи да се све нити апликације заустављају док се операција скупљања објеката не заврши [21]. Најважније предности аутоматског скупљања отпадака су:

заштита од цурења меморије — меморија се ослобађа онда када алгоритам који се користи за скупљање отпадака то одреди о чему програмер не треба да води рачуна,

повећана продуктивност програмера — уместо да води рачуна о ослобађању меморије, програмер више времена може посветити писању логике програма,

стабилност апликације — аутоматским чишћењем меморије смањују се грешке попут *segmentation fault* које утичу на стабилност апликације,

прилагођавање различитим окружењима — програмер одабиром одговарајућег алгоритма може прилагодити начин скупљања отпадака конкретној апликацији.

Коришћење скупљача отпадака може у потпуности елиминисати главне проблеме са меморијском алокацијом и деалокацијом, али за то се плаћа цијена. Аутоматским скупљањем отпадака жртвује се дио меморијске ефикасности зарад повећања угодности програмирања. Највећи проблем скупљача отпадака је потенцијално успоравање рада апликације због времена које се троши на анализирање меморије. Зато језици који користе скупљач отпадака нису препоручљиви за програмирање система који раде у реалном времену.

2.1 Алгоритми за скупљање отпадака

Одабир алгоритма за скупљање отпадака је од великог значаја јер може утицати на перформансе конкретне апликације. У наставку ће бити описани алгоритми које скупљачи отпадака најчешће користе. У пракси се често користе и њихове комбинације.

Пребројавање референци (енг. *reference counting*) — алгоритам се заснива на пребројавању референци објекта. Алгоритми овог типа имају инваријанту: објекат је **жив** ако и само ако је број објеката који реферишу на њега строго већи од нуле. Да би било могуће одредити да ли је објекат жив или не, алгоритам за сваки објекат чува бројач референци тј. број објеката који реферишу на тај објекат [3]. Када год се дода нова референца или показивач који се везује за тренутни ресурс бројачи се повећавају. Бројачи се смањују када год се обрише референца или показивач која се везује за дати ресурс. Ресурс је могуће ослободити када бројач има вриједност 0.

Постоје двије главне мане овог алгоритма: честа ажурирања бројача референци и показивача за сваки објекат (која захтијевају синхронизовање операција када год их треба извршити и самим тим лоше утичу на ефикасност извршавања) и цикличне референце (бројач референци никада није 0 и самим тим имамо објекте који у програму никада неће бити спремни за брисање).

Постоји неколико начина за смањивање учесталости ажурирања референци:

- одлагање ажурирање референци до одређеног тренутка — на тај начин се могу смањити проблеми са синхронизацијом,

- спајање — претпоставка је да је бројач референци од значаја само у одређеним тренуцима извршавања програма па се само у тим стањима врши инкрементирање односно декрементирање,
- баферовано пребројавање — чување међурезултата у баферу којима се приступа касније.

За рјешавање проблема цикличних референци користи се техника пробног брисања. Идеја је да се објекти за које постоји сумња да праве циклус фиктивно обришу. Уколико се испостави да је погођен објекат који прави циклус, алгоритам наставља даље са радом, у супротном се налази нови кандидат техником враћања уназад.

Праћење референци (енг. *reference tracing*) — алгоритам се заснива на обиласку графа објеката и означавању објеката који су достижни. Уколико је објекат достижан сматра се да је жив, у супротном се брише. Праћење референци се често назива и алгоритмом **означи и обриши**. Проблем са овим алгоритмом је то што није могуће мијењати граф повезаности објеката током скупљања отпадака.

Надоградња овог алгоритма је апстракција алгоритам бојења графа са три боје. Сваком од објеката се придружује неко од следећих стања тј. боја:

- бијела — објекат није достижан (иницијално стање свих објеката);
- црна — објекат који не реферише на објекте у бијелом стању. Ови објекти нису кандидати за скупљање;
- сива — објекти који још увијек садрже референце ка бијелим објектима. Ни ови објекти нису кандидати за скупљање (у неком тренутку могу прећи у црно стање).

Алгоритам бојења графа са три боје на почетку означава све коријене објекте сивом бојом, док остале означава бијелом. Након тога, докле год постоје објекти у сивом скупу, узима се један и пребацује у црни скуп након чега се сви објекти на које дати објекат реферише убацују у сиви скуп. Објекти из бијелог скупа се третирају као отпад који је потребно скупити. Инваријанта алгоритма је следећа: ниједан црни објекат не реферише директно на бијели.

Претходни алгоритам се може побољшати тако што скупљач на основу текућег стања објекта као и обрисаног објекта може истовремено ажурирати стање објекта и скупљати бијеле објекте.

Cheney алгоритам[1] (енг. *Cheney algorithm*) је такође једна варијанта алгоритма за праћење референци. Алгоритам се у литератури може наћи и под неким другим називима као што су: алгоритам копирања скупљања (енг. *copy garbage collection algorithm*) или алгоритам заустављања приликом копирања скупљања (енг. *stop and copy garbage collection algorithm*). У овом алгоритму, хип се дијели на пола, при чему се у једном тренутку само једна половина може користити. Техника се заснива на томе да се сви живи објекти једне половине, која се назива **полазни простор** (енг. *from space*), пребаце у другу половину тј. **циљни простор** (енг. *to space*), који након тога постаје нови хип.

Хибридни алгоритми или алгоритми подјеле хипа (енг. *partitioned/-hybrid algorithms*) — алгоритми се заснивају на дијељењу хипа на мање цјелине и на сваку појединачну цјелину се примењује жељени алгоритам скупљања[2]. Постоји неколико подјела хипа од којих можемо издвојити неке:

- подјела објеката на основу цијене њиховог помјерања. Уколико је објекат велики онда се не исплати копирати га и премјештати са једне позиције на другу,
- подјела објеката на основу неког заједничког својства.

Најзначајнији алгоритам подјеле хипа је генерацијски. Идеја је да се објекти групишу на основу својих година тј. броја скупљања које су преживјели. Хип се у том случају најчешће дијели на два дијела и то:

младу генерацију — служи за чување објеката који кратко живе,

стару генерацију — служи за објекте који су преживјели већи број скупљања.

[1] Алгоритам носи назив по *C.J.Cheney* који га је описао у научном раду објављеном 1970. године [4].

[2] *Cheney* алгоритам није хибридни јер се подјела хипа врши у циљу чувања објеката на које постоје референце тј. који се још увијек користе у програму.

Млада генерација се много брже пуни него стара и увијек је мање димензије. Стога, смањење пауза током рада апликације се може постићи честим чишћењем младе и повременим чишћењем старе генерације која је значајно већа па је самим тим обилазак објеката ове генерације временски скупљи [3].

2.2 Скупљач отпадака Јава виртуелне машине

Програмски језик Јава настао је 1995. године у оквиру компаније *Sun Microsystems*. Од 2010. године је у власништву компаније *Oracle*. Јава је објектно-оријентисани, строго типизирани језик опште намјене [21].

Стандардно превођење Јава програма се заснива на компилацији до међурепрезентације (тзв. бајткода) за Јава виртуелну машину[3] (JVM) компилатором *javac*, а затим се добијени бајткод интерпретира на Јава виртуелој машини Јава интерпретером [21].

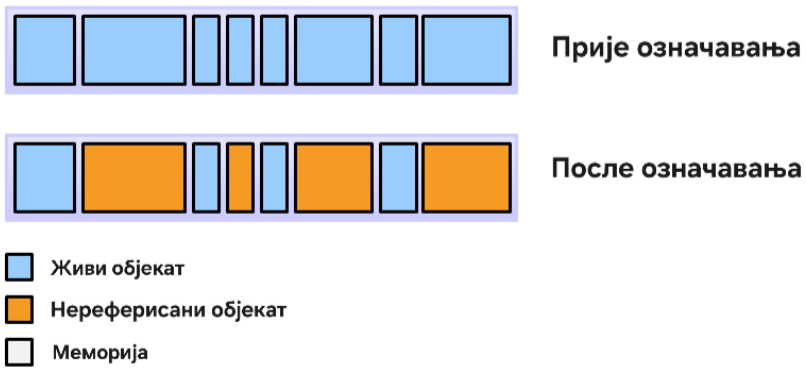
Имплементација Јава виртуелне машине компаније *Oracle* позната је под називом *HotSpot*. Постојање бајткода омогућава већу преносивост Јава апликација јер је за њихово покретање на одређеној архитектури неопходно једино постајање Јава виртуелне машине које ће интерпретирати генерисани бајткод [21]. Како би се побољшале перформансе апликација, на Јава виртуелној машини, умјесто интерпретације користи се JIT (енг. *Just In Time*) компилација која подразумијева парцијално превођење бајткода праћено оптимизацијама за конкретну архитектуру. Поред JIT компилације постоји и AOT (енг. *Ahead of Time*) компилација која подразумијева генерисање објектног кода за циљну машину прије почетка извршавања програма. У случају AOT компилације познати су детаљи конкретне машине и самим тим оптимизација може бити боља.

У програмском језику Јава, као и у многим другим програмским језицима, процес управљања меморијом извршава се аутоматски, тј. програмер не мора ручно да ослобађа меморију. Процес скупљања отпадака у општем случају користи алгоритам праћења референци у комбинацији са хибридним генерацијским алгоритмом.

[3]Рачунар је машина која може да извршава кôд на машинском језику. Слично, виртуелне машине представљају машине које постоје само у меморији и служе за извршавање кода на одређеном језику. У случају Јава виртуелне машине тај језик је бајткод.

Алгоритам праћења референци који користи Јавин скупљач се може описати следећим корацима:

1. **Означавање** — скупљач отпадака одређује који дијелови меморије се користе а који не (Слика 2.1).

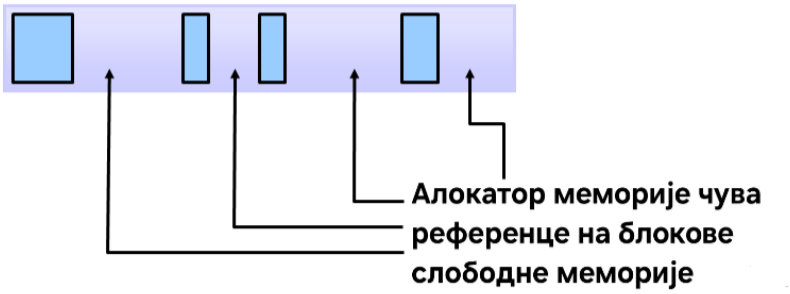


Слика 2.1: Означавање објеката

Сви објекти се скенирају у овој фази како би она била детерминистичка. Овај процес може потрајати дуго, у зависности од количине објеката које је потребно скенирати.

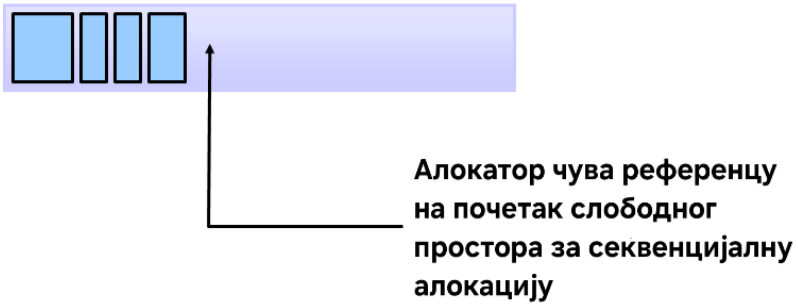
2. **Брисање** — скупљач отпадака брише објекте који се не користе на неки од наредна два начина:

   Нормално брисање — брисање нереферисаних објеката. Алокатор меморије чува референцу на блок слободне меморије у који је могуће смјестити нове објекте (Слика 2.2).



Слика 2.2: Нормално брисање

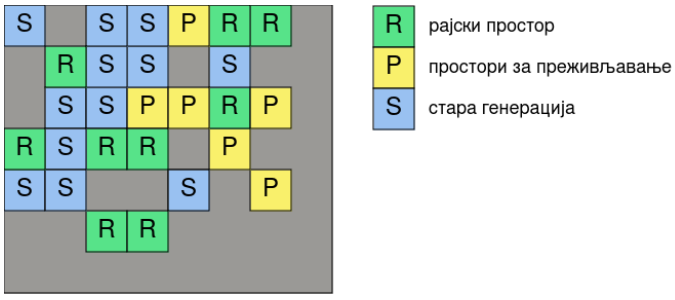
Брисање са груписањем — да би се побољшале перформансе, након брисања нереферисаних објеката, могуће је груписати референсане објекте тако да заузимају секвенцијални дио меморије. Груписањем објеката значајно се олакшава и убрзава процес меморијске алокације (Слика 2.3).



Слика 2.3: Брисање са груписањем

Као што је поменуто раније, означавање и груписање свих објеката је неефикасно. Што се више објеката алоцира, то је листа објеката већа и самим тим потребно је више времена за скупљање. Емпиријска истраживања су показала да већина објеката кратко живи.

Хибридни алгоритам који користи Јавин скупљач хип дијели на мање дијелове које називамо генерацијама. Разликујемо три генерације: младу генерацију, стару или радну генерацију и трајну генерацију. Логички приказ хипа се може видјети на слици 2.5. У реалности, меморија је подијељена на мање дијелове фиксне дужине који служе за смјештање објеката што се може видјети на слици 2.4.



Слика 2.4: Реални приказ хип меморије

Млада генерација је простор за новоалоциране објекте и од тог тренутка се мјери старост објекта. Старост или године објекта одговарају скупљањима које је преживио. Када објекат достигне одређени број година или уколико у младој генерацији нема више мјеста, пребацује се у стару генерацију. Трајна генерација садржи основне податке који су неопходни Јава виртуелној машини за описивање класа и метода које се користе у апликацији тј. податке који се ријетко бришу. Млада генерација се састоји из **рајског простора** за новоалоциране објекте (енг. *eden space*) и више **простора за преживљавање** (енг. *survivor spaces*).

Описани алгоритам за праћење референци се користи за брисање и промоцију[4] објеката кроз генерације хибридног алгоритма.

Слика 2.5: Структура хипа код Јава виртуелне машине

Свака покренута нит у програмском језику Јава има мали дио рајског простора који не дијели са другим нитима — **локални бафер за алокацију TLAB** (енг. *Thread Local Allocation Buffer*). На почетку рада апликације објекти се смјештају у локални бафер за алокацију док се он не попуни, након чега се објекти алоцирају у оквиру преосталог слободног дијела рајског простора. Величина локалног бафера варира али углавном је мала, око `512KB` или `1MB`. Када се рајски простор младе генерације попуни, дешава се **мало скупљање** (енг. *minor garbage collection*). Живи објекти се, након првог малог скупљања, пребацују у први простор за преживљавање. Проблем настаје уколико постоје објекти ван младе генерације који реферишу на објекте младе генерације. У том случају, објекти ван младе генерације могу да реферишу на старе локације објеката у младој генерацији. Овај проблем се рјешава коришћењем структуре за чување додатних информација која се назива **запамћени скуп** (енг. *remembered set*). Једна врста запамћеног скупа је **табела**

[4]Промовисање подразумијева пребацивање објекта из једног дијела хипа у други.

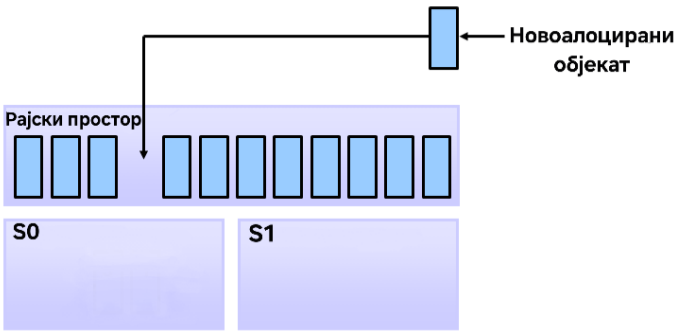
карата (енг. *card table*) која је на Јава виртуелној машини имплементирана као низ битова. Сваки бит низа одговара одређеном дијелу хипа који се назива **карта** [5]. Уколико је вриједност бита 1 тј. уколико је карта означена, онда у дијелу хипа којем карта одговара постоји објекат из старе или трајне генерације који реферише на неки објекат из младе генерације. Након малог скупљања, неопходно је провјерити и по потреби ажурирати референце објеката из свих означених карата. Најчешће, већина карата није означено па је довољно проћи само кроз објекте младе генерације.

Велико скупљање (енг. *major garbage collection*) се користи за анализу и чишћење објеката старе генерације. Како је млада генерација у уопштем случају значајно мања од старе генерације, то је велико скупљање спорије од малог што значи да механизам заустављања свијета може значајно успорити рад апликације уколико је потребно сакупити велику количину отпадака. Велико скупање је најчешће праћено малим скупљањем. Поред малог и великог скупљања постоји и **потпуно скупљање** (енг. *full garbage collection*) приликом којег долази до скупљања отпадака у оквиру свих генерација [21]. Дакле, поред скупљања у оквиру младе и старе, врши се и скупљање објеката из трајне генерације.

Следећи кораци описују начин промоције објеката кроз претходно поменуте дијелове хипа [11]:

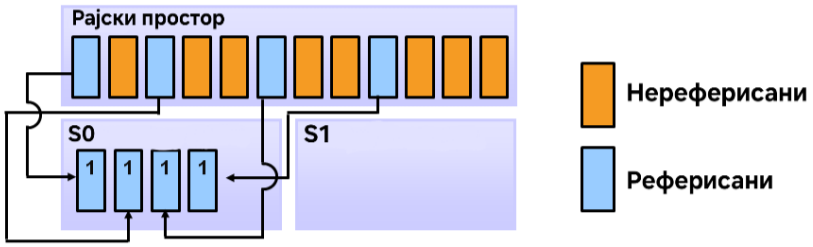
1. Алоцирани објекти се убацују у рајски простор младе генерације. Оба простора за преживљавање на почетку су празна.

2. Када се рајски простор попуни дешава се мало скупљање. На слици 2.6 приказано је алоцирање објекта који ће попунити рајски простор.



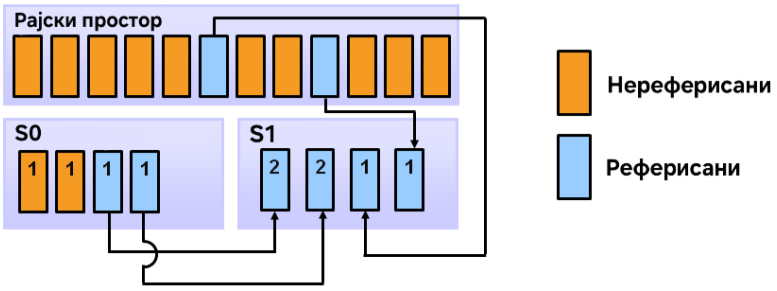
Слика 2.6: Додавање новоалоцираног објекта у рајски простор

3. Приликом малог скупљања у рајском простору нереферисани објекти се бришу, док се они на које постоји референца пребацују у први простор за преживљавање (S0) (Слика 2.7). Бројеви унутар објеката на слици означавају године објекта.



Слика 2.7: Брисање и промовисање објеката

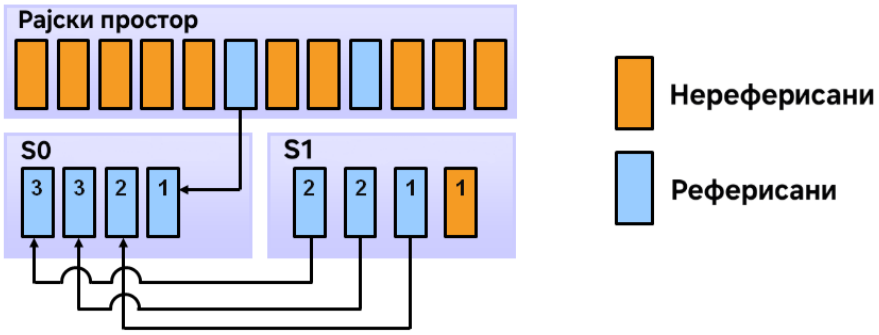
4. Након следећег малог скупљања дешава се иста ствар са рајским простором. У овом случају, реферисани објекти се убацују у други простор за преживљавање (S1). Такође, објекти из S0 се пребацују у S1, при чему се ажурира њихова старост. Дакле, у простору за преживљавање могуће је имати објекте различите старости (Слика 2.8).



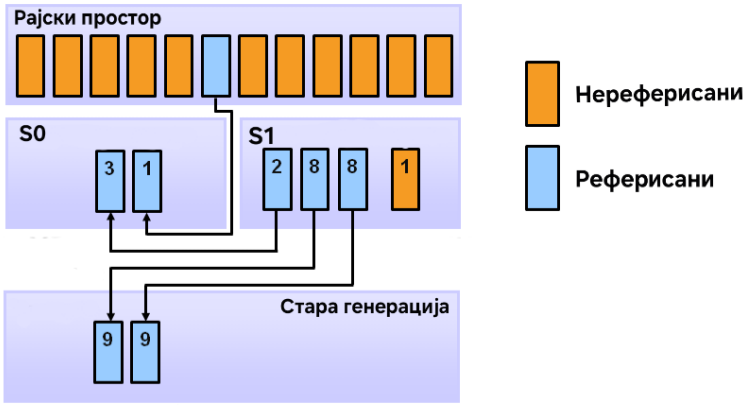
Слика 2.8: Промовисање објеката у просторима за преживљавање

5. Након следећег малог скупљања поступак се понавља. Једина разлика у односу на претходни поступак је замјена мјеста простора S0 и S1 у самом алгоритму (Слика 2.9).

6. На слици 2.10 се може видјети поступак промоције објеката. Након малог скупљања, када објекти досегну одређене године[5] промовишу се



Слика 2.9: Полазни и циљни простор



Слика 2.10: Промовисање објеката у стару генерацију

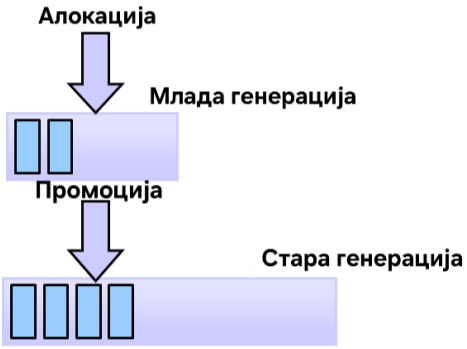
из младе у стару генерацију.

7. Како мало скупљање наставља да се дешава објекти настављају да се пребацују у стару генерацију (Слика 2.11).

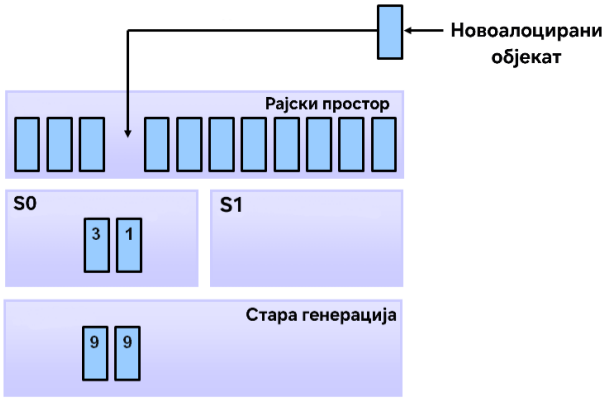
8. Велико скупљање служи за чишћење и груписање објеката у оквиру старе генерације (Слика 2.12).

Тренутно подразумијевани скупљач отпадака Јава виртуелне машине назива се **G1** (енг. *garbage first*) и представља вишенитни скупљач са редукованим паузама узрокованим заустављањем свијета. Заустављања се врше у одређеним дијеловима у току скупљања као што је процесирање референци и груписање живих објеката како би се спријечила фрагментација меморије. Са

[5]Старосна граница код које долази до промоције објеката из младе у стару генерацију се дефинише у имплементацији младе генерације.



Слика 2.11: Пут промоције објекта



Слика 2.12: Стање хипа након великог скупљања

друге стране, означавање објеката тј. одређивање да ли је објекат жив или не, врши се конкурентно. Таква имплементација смањује кашњење и побољшава ефикасност апликације [5]. Поред овог скупљача, на Јава виртуелној машини постоје и [12]:

- *Serial GC* — серијски скупљач који у једној нити скупља отпад. Скупљање отпадака је ефикасно јер нема синхронизације нити. Додатно, може користити метод груписања током скупљања.
- *Parallel GC* — паралелни скупљач који мало скупљање извршава вишенитно у циљу побољшања ефикасности док се велико скупљање извршава у једној нити. На уређају са N језгара обично користи N нити за мало скупљање. Могуће је и велико скупљање извршавати вишенитно укључивањем опције `-XX:+UseParallelOldGC`.
- *Z GC* — скупљач који ради у потпуности конкурентно без заустављања

извршавања апликације. Користи се код апликација које захтијевају минимално кашњење.

Избор скупљача отпадака зависи од потреба конкретне апликације и капацитета машине на којој се апликација покреће. Нпр. за апликације које треба да раде у реалном времену бољи избор су скупљачи који раде паралелно. Са друге стране, уколико машина има мало језгара боље је користити серијски скупљач отпадака.

Глава 3

Систем *GraalVM*

Систем *GraalVM* [7] представља Јава развојно окружење које омогућава превођење улазног програмског кода написаног у различитим програмским језицима у машински кôд који одговара различитим циљним архитектурама, као и извршавање генерисаног машинског кода употребом заједничких компоненти система, независно од конкретног изворног језика или циљне архитектуре.

3.1 Компилатор *GraalVM Native Image*

Компилатор *GraalVM Native Image* [8] је написан у програмском језику Јава и користи се за АОТ компилацију Јава бајткода. Компилацијом се добија извршива датотека или дијељена библиотека.[1] Извршива датотека може бити стандардно или статички повезана. Статички повезана извршива датотека не зависи од дијељених библиотека на систему на којем се покреће већ се сав библиотечки кôд налази у самој датотеци, док се код стандардно повезане извршиве датотеке потребан библиотечки кôд динамички учитава на почетку самог извршавања датотеке. Резултат компилације се назива изворна слика (енг. *native image*) или једноставно слика (енг. *image*).

Извршива датотека укључује класе које користи апликација, њихове зависности и статички линковане методе из Јава алата за развој. Слика се не покреће на Јава виртуелној машини, али укључује неопходне компоненте као што је управљање меморијом, расподјела нити и слично, захваљујући вир-

[1] У наставку текста ћемо се једноставности и читљивости ради референсати само на извршиву датотеку, подразумевајући притом и дијељене библиотеке.

туелној машини *SubstrateVM*. *SubstrateVM* представља назив за компоненте присутне у току извршавања програма, попут скупљача отпадака, распоређивача нити итд. Апликације компилиране коришћењем *GraalVM Native Image* технологије имају смањену потребу за меморијским ресурсима у односу на апликације које се извршавају на Јава виртуелној машини *HotSpot*.

Генератор изворне слике (енг. *native-image generator*) је алат у оквиру компилатора *GraalVM Native Image* који обрађује класе апликације и друге мета податке и након тога креира извршиву датотеку за конкретни оперативни систем и архитектуру. На почетку, алат користи технике статичке анализе над кодом како би се одредиле класе и методе које су достижни у току извршавања апликације. Након тога, врши се компилација класа, метода и других ресурса у извршиву датотеку. Овај процес се назива **фаза превођења** или **вријеме изградње** (енг. *build time*) како би се раздвојио од процеса компилације Јава изворног кода до бајткода.

Извршива датотека

Да би AOT компилацијом направио високо оптимизовану извршиву датотеку, *GraalVM Native Image* користи агресивну статичку анализу која ради по принципу **затвореног свијета** (енг. *closed-world*), што значи да све класе и бајткодови који су достижни у вријеме извршавања морају бити познати у фази превођења. Дакле, то значи да у фази извршавања није могуће учитати нове класе које нису биле доступне у току AOT компилације.

Анализа достижности је дио статичке анализе [9] у којој се, без покретања саме апликације, одређује који елементи програма тј. које класе, методе и поља се користе у апликацији. Ови елементи се називају **достижним**. Анализа достижности у систему *GraalVM* се састоји из два дијела:

- анализа бајткода методе како би се одредили елементи који су из те методе достижни,
- анализа статички иницијализованих објеката, како би се одредиле класе које су достижне из њих.

Анализа креће од методе *main* и достижност се рачуна итеративно све док анализа допуњава достижност елемената. У коначној извршивој датотеци се могу наћи само елементи за које анализа утврди да су достижни.

Image хип и Јава хип

Компилатор *Native Image* у фази изградње врши иницијализацију класа тј. извршавање неких метода класа и уписивање вредности одређених поља у извршиву датотеку. Иницијализатор класе је метода *clinit* у бајткоду. Иницијализатор класе се у изворном коду може дефинисати експлицитно, коришћењем статичког блока за иницијализацију или имплицитно, дефиницијом статичких поља и метода класе [19].

Приликом изградње извршиве датотеке иницијализовани објекти се чувају у посебном дијелу који се назива изворни хип слике (енг. *Native Image heap*), а често и само хип слике (енг. *image heap*). На овај начин се убрзава рад апликације — одређени објекти су већ иницијализовани и њима се може директно приступати у вријеме извршавања апликације. Све класе које не зависе од објеката који се креирају у вријеме извршавања се иницијализују у вријеме изградње. За класе корисник сам може одлучити када их жели иницијализовати коришћењем кључних ријечи *static* и *final*.

Објекти који се налазе у хипу слике су **бесмртни** — скупљач отпадака их неће брисати већ их користи као **коријене** објекте од којих провјерава достижност осталих. Дакле, хип слике има сличну улогу као трајна генерација Јава виртуелне машине. Прецизније, хип слике садржи:

- објекте који се креирају у фази изградње извршиве датотеке и достижни су из кода апликације,
- објекте класа *java.lang.Class* које се користе у извршивој датотеци,
- константне објекте који су уграђени у дефиниције достижних метода.

Поред хипа слике постоји и хип извршавања (енг. *Runtime heap*) или једноставно Јава хип који служи за складиштење објеката приликом извршавања апликације. Овај хип се креира када се извршива датотека покрене. Када се одређени дио Јава хипа попуни, започиње процес скупљања отпадака.

Изградња извршиве датотеке

На листингу 3.1 се налази једноставна Јава апликација на којој ће бити приказана AOT компилација као и детаљно описан процес изградње извршиве датотеке. Компилација изворног програмском кода до бајткода, а затим изградња извршиве датотеке постиже се наредбама из листинга 3.2.

```java
public class HelloWorld {
    public static void main(String[] args) {
        // Print "Hello, World" to the console
        System.out.println("Hello, World! :)");
    }
}
```

Листинг 3.1: *HelloWorld.java*

```
$ javac HelloWorld.java
$ native-image HelloWorld
```

Листинг 3.2: Превођење Јава програма до бајткода (прва команда), а затим изградња извршиве датотеке (друга команда)

Изградња извршиве датотеке пролази кроз неколико фаза. У листингу 3.3 је приказан стандардни излаз настао приликом изградње за кôд са листинга 3.1. У наставку су дата објашњења појединачних фаза и осталих дијелова излаза који се генерише сваки пут приликом изградње извршиве датотеке [10].

```
========================================================================================
GraalVM Native Image: Generating 'helloworld' (executable)...
========================================================================================
[1/8] Initializing...                                                    (2.6s @ 0.11GB)
 Java version: 23+17, vendor version: Oracle GraalVM 23-dev+17.1
 Graal compiler: optimization level: 2, target machine: x86-64-v3, PGO: ML-inferred
 C compiler: gcc (linux, x86_64, 11.4.0)
 Garbage collector: Serial GC (max heap size: 80% of RAM)
 1 user-specific feature(s):
 - com.oracle.svm.thirdparty.gson.GsonFeature
----------------------------------------------------------------------------------------
Build resources:
 - 13.48GB of memory (43.9% of 30.69GB system memory, determined at start)
 - 20 thread(s) (100.0% of 20 available processor(s), determined at start)
[2/8] Performing analysis...  [*****]                                    (3.0s @ 0.27GB)
    2,085 reachable types   (58.6% of    3,555 total)
    1,868 reachable fields  (38.4% of    4,861 total)
    8,823 reachable methods (35.0% of   25,194 total)
      758 types,    36 fields, and   328 methods registered for reflection
       49 types,    34 fields, and    48 methods registered for JNI access
        4 native libraries: dl, pthread, rt, z
[3/8] Building universe...                                               (0.7s @ 0.26GB)
[4/8] Parsing methods...      [*]                                        (0.7s @ 0.29GB)
[5/8] Inlining methods...     [***]                                      (0.3s @ 0.32GB)
[6/8] Compiling methods...    [***]                                      (7.8s @ 0.28GB)
[7/8] Laying out methods...   [*]                                        (0.4s @ 0.30GB)
[8/8] Creating image...       [*]                                        (0.6s @ 0.33GB)
   3.04MB (44.01%) for code area:     4,029 compilation units
   3.19MB (46.22%) for image heap:   52,304 objects and 52 resources
```

```
 691.66kB ( 9.78%) for other data
   6.91MB in total
--------------------------------------------------------------------------------
Top 10 origins of code area:         Top 10 object types in image heap:
  1.44MB java.base                     792.84kB byte[] for code metadata
  1.33MB svm.jar (Native Image)        710.19kB byte[] for java.lang.String
 82.79kB com.oracle.svm.svm_enterprise 365.91kB java.lang.String
 35.04kB org.graalvm.nativeimage.base  353.05kB java.lang.Class
 33.25kB jdk.proxy2                    141.63kB java.util.HashMap$Node
 30.33kB jdk.proxy1                    14.52kB char[]
 21.73kB org.graalvm.collections       87.17kB java.lang.Object[]
 21.28kB jdk.internal.vm.ci            81.88kB byte[] for reflection metadata
 14.23kB jdk.graal.compiler            81.45kB com.oracle.svm.core.hub.DynamicHubCompanion
  7.79kB jdk.proxy3                    70.98kB heap alignment
389.00B for 1 more packages            470.80kB for 523 more object types
                        Use '-H:+BuildReport' to create a report with more details.
--------------------------------------------------------------------------------
Security report:
 - Binary includes Java deserialization.
 - Use '--enable-sbom' to embed a Software Bill of Materials (SBOM) in the binary.
--------------------------------------------------------------------------------
Recommendations:
 G1GC: Use the G1 GC ('--gc=G1') for improved latency and throughput.
 PGO:  Use Profile-Guided Optimizations ('--pgo') for improved throughput.
 HEAP: Set max heap for improved and more predictable memory usage.
 CPU:  Enable more CPU features with '-march=native' for improved performance.
 QBM:  Use the quick build mode ('-Ob') to speed up builds during development.
--------------------------------------------------------------------------------
            1.1s (6.4% of total time) in 300 GCs | Peak RSS: 1.14GB | CPU load: 13.54
--------------------------------------------------------------------------------
Build artefacts:
 /home/milica/bitbucket/graal-enterprise/substratevm-enterprise/helloworld (executable)
================================================================================
Finished generating 'helloworld' in 16.7s.
```

Листинг 3.3: Фазе изградње извршиве датотеке `helloworld`

Иницијализација (енг. *Initializing*). Ово је почетна фаза изградње у оквиру које се врши иницијализација одређених класа. Након тога се приказује врста датотеке која се гради (стандардна извршива датотека, дијељена библиотека или статичка извршива датотека), верзија Јаве као и врсте компилатора и скупљача отпадака. Поред информације о врсти скупљача отпадака налази се информација о максималној величини Јава хипа.

Анализа (енг. *Performing analysis*). У овој фази се врши статичка анализа кода и достижних типова, поља и метода.

Изградња универзума (енг. *Building universe*). У овој фази прави се универзум са свим типовима, пољима и методама које се користе у апликацији који ће касније бити коришћен за формирање извршиве датотеке.

Парсирање метода (енг. *Parsing methods*). У овој фази се парсирају достижне методе.

Уметање или инлајновање метода (енг. *Inlining methods*). У овој фази се врши уметање једноставних метода.[2]

Компилација метода (енг. *Compiling methods*). У овој фази *Graal* компилатор преводи све достижне методе до машинског кода.

Распоређивање метода (енг. *Laying out methods*). У овој фази се фиксира редослед претходно преведених метода у извршивој датотеци.

Изградња извршиве датотеке (енг. *Creating image*). У овој фази се креира извршива датотека која се чува на диску. У овој фази се *debug* информације уписују у извршиву датотеку уколико је то тражено.

Након завршених фаза приказани су предлози за побољшање перформанси дате апликације, статистике у вези са скупљачем отпадака, меморијским заузећем и употребом процесора. Поред тога, могу се видјети и путање до генерисаних датотека као и вријеме које је утрошено у фазу изградње.

3.2 Организација меморије и серијски скупљач отпадака

У систему *GraalVM*-у могуће је користити неки од следећих скупљача отпадака:

- *Serial GC* — серијски скупљач који се користи као подразумијевани за систем *GraalVM* и о њему ће бити више ријечи у наставку.
- *G1 GC* — тренутно подразумијевани скупљач отпадака Јава виртуелне машине о коме је било ријечи у поглављу 2.2.

[2]Уметање је поступак у коме се на мјесту позива методе уметне тијело методе. Уколико су у питању једноставне методе много је ефикасније на мјесту њиховог позива исписати кôд методе како би се избјегло креирање стек овира и преношење параметара и повратне вриједности.

- *Epsilon GC* — скупљач који не скупља ништа. Намијењен је за једноставне апликације које врло мало меморије алоцирају.

Серијски скупљач отпадака [13] је једноставни скупљач који, као што му и само име каже, ради серијски и по принципу заустављања свијета. Користи се као подразумијевани скупљач у систему *GraalVM*. У овом раду биће приказан побољшани алгоритам за скупљање отпадака у клијент-сервер апликацијама.

Организација Јава хипа

Као што је већ речено у поглављу 2.1, најчешћа подјела хипа је на младу и стару генерацију што је случај и са Јава хипом. Свака генерација састављена је од једнаких дјелова — **комада** (енг. *chunk*). Комад представља непрекидни дио виртуелне меморије. Ове комаде скупљач користи као основну јединицу мјере приликом алокације и реалокације меморије. Постоје двије врсте комада: **поравнати** (енг. *aligned*) и **непоравнати** (енг. *unaligned*). Поравнати комади служе за складиштење мањих објеката и њихова величина је фиксна и износи `512KB`. Непоравнати комади се користе за чување великих објеката који не могу да стану у један поравнати комад. Сваки простор младе и старе генерације садржи повезану листу комада.

Рајски простор младе генерације је резервисан за нове објекте. Када се овај дио младе генерације попуни покреће се **инкрементално скупљање** или **инкрементална колекција**. Објекти се у том случају пребацују у полазни простор за преживљавање. Када се тај простор попуни, објекти се даље пребацују у стару генерацију. Инкрементално скупљање обавља промоцију објеката кроз младу генерацију као и из младе у стару генерацију тј. има исту улогу као и мало скупљање код скупљача Јава виртуелне машине. Чишћење старе генерације се ради помоћу **комплетног скупљања** или **комплетне колекције** која је значајно скупља од инкременталне, али је важна јер спречава задржавање објеката који након одређеног времена постану непотребни. Комплетно скупљање код серијског скупљача има исту улогу као и велико скупљање код скупљача Јава виртуелне машине. Пандан потпуног скупљања не постоји јер улогу трајне генерације има *Image* хип.

Приликом промоције објеката, уколико се објекат налази у поравнатом комаду онда се он копира у нови поравнати комад циљног простора. За разлику од њих, објекти који се налазе у непоравнатим комадима се не копирају јер

могу бити произвољно велики, већ се преусмјере показивачи[3] — обрише се показивач из текућег простора у коме се непоравнати комад налази и дода се показвач на дати комад из цильног простора.

3.3 Опције превођења и извршавања

Опције које се користе у вријеме изградње (енг. *hosted options*) и опције које се користе у вријеме извршавања (енг. *runtime options*) омогућавају прецизирање жељеног понашања приликом превођења односно приликом извршавања програма.

Опције изградње задају се приликом изградње извршиве датотеке и њихов начин задавања је:

`-H:±opcija` — уколико је тип вриједности која се крије иза опције типа *boolean* тј. уколико опција може бити укључена и искључена. Нпр. опцијом `-H:+VerifyReferences` се врши провјера валидности постојећих референци програма.

`-H:opcija=vrijednost` — уколико тип вриједности која се крије иза опције може имати више од двије вриједности. Нпр. опцијом `-H:-MaxSurvivorSpaces=4` задајемо да се могу користити највише четири простора за преживљавање.

Опције извршавања најчешће се задају приликом покретања извршиве датотеке и једина разлика у односу на начин задавања претходно дефинисаних опција је то што се умјесто слова `H` користи слово `R` или `XX`. Неке опције извршавања је неопходно послиједити приликом изградње извршиве датотеке. Разлог због ког су те опције извршиве је то што се њихова вриједност може промијенити у фази извршавања. Примјери опција које се користе у фази извршавања су:

`MaximumYoungGenerationSizePercent` — опција којом задајемо максималну процентуалну величину младе генерације у односу на хип. Нпр.

```
-R:MaximumYoungGenerationSizePercent=40
```

[3]Показивач (енг. *pointer*) представља кориснички дефинисан тип у систему *GraalVM* који постоји како би се олакшало управљање меморијом.

```
Hello world! :)
GC summary
  Collected chunk bytes: 0.00M
  Collected object bytes: 0.00M
  Allocated chunk bytes: 1.50M
  Allocated object bytes: 0.12M
  Incremental GC count: 0
  Incremental GC time: 0.000s
  Complete GC count: 0
  Complete GC time: 0.000s
  GC time: 0.000s
  Run time: 0.000s
  GC load: 0%
```

Листинг 3.4: Употреба опције `PrintGCSummary`

`PrintGCSummary` — уколико је опција укључена, исписују се статистике скупљача, попут броја скупљених и сачуваних бајтова, вријеме проведено у скупљању итд.

Излаз укључивањем опције `-R:+PrintGCSummary` се може видјети на листингу 3.4. Како је програм веома једноставан, скоро сви бројачи су једнаки нули.

Корисничке опције задају параметре компилације или извршавања за које не треба користити префиксе `-R` и `-H`, неке од њих се могу видјети позивом команде `native-image --help`. Корисничке опције често обједињују у себи више других опција или представљају скраћенице за често коришћене опције. Неке од често коришћених корисничких опција су:

`-Xmx` — опција којом задајемо максималну величину хипа коју може да користи апликација. Нпр. за хип величине `64MB` употреба опција би имала облик `-Xmx64m`.

`-Xmn` — опција којом задајемо величину младе генерације у бајтовима. Нпр. `-Xmn32m` поставља величину младе генерације на `32MB`.

Примјер коришћења неких опција[4]:

```
$ native-image -H:+UnlockExperimentalVMOptions -H:MaxSurvivorSpaces=4
    -R:+PrintGCSummary HelloWorld
$ ./helloworld -Xmx64m
```

Листинг 3.5: Први ред — фаза изградње извршиве датотеке, други ред — фаза покретања извршиве датотеке

[4]Опција за промјену броја простора за преживљавање је експериментална па је неопходно укључити и опцију `-H:+UnlockExperimentalVMOptions` која дозвољава коришћење експерименталних опција.

Глава 4

Систем за ефикасније скупљање отпадака

Систем који је развијен у оквиру овог рада има за циљ да побољша перфомарнсе клијент-сервер апликација ефикаснијим скупљањем отпадака. Покретањем система над клијент-сервер апликацијама чувају се статистике извршавања програма попут текућег заузећа меморије, броја урађених захтјева у секунди итд. Те статистике омогућавају прилагођавање скупљања отпадака датој апликацији. Такође, оне ће бити коришћене и за одређивање разлике између ефикасности подразумијеваног и развијеног система.

Клијент-сервер апликације

Велики број модерних рачунарских система, попут интернет сајтова и мобилних апликација, представља моделе клијент-сервер система. Клијент-сервер систем је дистрибуиран рачунарски систем између два типа независних ентитета познатијих као сервери и клијенти који комуницирају преко мреже. У овим системима разликујемо физичке и логичке компоненте [15]. Физичке компоненте су сервери и клијент уређаји, улазни и излазни уређаји, док су логичке компоненте интернет стране, подаци, протоколи итд.

Клијент представља уређај који може да шаље захтјеве серверу и да прима информације. Сервер је уређај који прима и обрађује захтјеве које шаље клијент. Сервер може да опслужује више клијената истовремено. Након укључивања, сервер чека да пристигну захтјеви клијената на одређеном порту[1].

[1]Порт је број у мрежи који се користи за успостављање везе између уређаја.

Да би се клијент повезао са сервером неопходно је да зна порт на ком сервер слуша и IP адресу[2] сервера. Када клијент пошаље захтјев серверу, сервер може или да прихвати или да одбаци примљени захтјев. Уколико је линк прихваћен сервер успоставља конекцију са клијентом преко посебног протокола. Протокол у овом случају представља унапријед дефинисан начин комуникације између клијента и сервера [16].

У клијент-сервер апликацијама узастопни захтјеви су најчешће слични што се може искористити за предвиђање меморијског заузећа захтјева који пристижу у будућности. Поред ове, битна претпоставка система чији ће опис бити дат у наставку, је то да сервер у једном тренутку обрађује захтјев једног клијента.

4.1 Основне идеје система

Праћењем алокације меморије у претходних неколико захтјева могуће је „припремити" хип за следећи захтјев. Како је инкрементално скупљање најчешће јефтиније, то је идеја да се што чешће примењује, као и да се води рачуна да се оно примењује ван времена обраде самог захтева. У клијент-сервер апликацијама објекти најчешће кратко живе па ће инкрементално скупљање бити довољна да почисти скоро читаву меморију јер ће већина објеката бити у младој генерацији.

Идеја система је да се динамички смањи величина рајског простора тако да и даље буде довољно простора за алокацију једног захтјева али да сам простор буде мањи, како би скупљања на њему била бржа и како би меморијско заузеће било мање. У општем случају иницијални рајски простор може смјестити објекте из више од једног захтјева што повећава меморијско заузеће апликације и продужава вријеме скупљања. У случају мале величине хипа, уколико рајски простор није довољан за смјештање објеката једног захтјева, динамички се врши његово повећање. На тај начин се спречава скупљање током извршавања захтјева.

Прије промјене величине рајског простора неопходно је урадити инкрементално скупљање. На тај начин, избјегава се скупљање отпадака у захтјеву што би повећало кашњење апликације. Такође, смањује се и меморијско заузеће јер су простори младе генерације мањи и често се чисте. Поред тога,

[2]IP адреса представља јединствену адресу појединачног уређаја у мрежи.

избјегава се позивање комплетног скупљања које је скупље и неповољно утиче на перформансе апликације.

Систем одређује наредне ставке:

1. када се динамички мијења величина младе генерације и
2. када се позива инкрементално, а када комплетно скупљање.

Прва ставка се одређује на основу максималног броја алокацираних бајтова у претходних n захтјева. Уколико се ажурира величине младе генерације, позива се инкрементално скупљање. Такође, уколико би предвиђени број бајтова наредног захтјева премашио величину рајског простора, позива се инкрементално скупљање.

Како је циљ побољшати ефикасност апликације то се комплетно скупљање отпадака ради само онда када проценимо да ће имати повољан утицај на ефикасност апликације. Најчешће метрике које се користе за мјерење перформанси апликације су:

пропусност (eng. *throughput*) — у контексту клијент-сервер апликација број захтјева који се обраде у датој јединици времена. Скупљања отпадака негативно утичу на ову вриједност јер се вријеме умјесто на обраду захтјева, троши на скупљање отпадака.

кашњење (eng. *latency*) — брзина одзива апликације. Паузе које се праве приликом скупљања отпадака негативно утичу на ову брзину.

траг (eng. *footprint*) — количина ресурса које користи систем, процес или апликација. Скупљач отпадака чисти меморију и самим тим спречава цурење меморије. Ипак, након одређеног времена се може појавити фрагментација меморије тј. постојање меморијских блокова који су превише мали да су неупотребљиви.

RSS (енг. *resident set size*) — представља количину меморије коју програм заузима у извршавању. Позивање скупљача отпадака чисти меморију што значи да повољно утиче на дату метрику. Уколико објекти кратко живе, то је инкрементално скупљање довољно за смањење меморијског заузећа тј. за побољшање вриједности ове метрике.

Ефикасност апликације може да се рачуна као количник урађених захтјева у секунди који обиљежавамо са T (скраћено од *throughput*) и заузете меморије коју обиљежавамо са R (скраћено од RSS). Да би добили позитиван утицај на ефикасност апликације, дефинишемо наредну неједнакост која треба да буде тачна:

$$\frac{T_{current}}{R_{current}} \leq \frac{T_{predicted}}{R_{predicted}} \tag{4.1}$$

Значења симбола у формули дефинишемо на следећи начин:

- $T_{current}$ — тренутни број урађених захтјева у једној секунди и рачуна се по формули
$$T_{current} = \frac{n}{t_n - t_0}$$
гдје је t_i вријеме почетка i-тог захтјева,

- $R_{current}$ — тренутно меморијско заузеће тј. укупан број бајтова који се налазе у младој (рајски и простори за преживљавање) и старој генерацији,

- $T_{predicted}$ — предвиђени број урађених захтјева у једној секунди уколико претпоставимо да је урађено једно комплетно скупљање. У том случају рачунамо број $n_{skipped}$ који представља број прескочених захтјева који су могли бити урађени да није било скупљања отпадака. Рачуна се по формули
$$T_{predicted} = \frac{n - n_{skipped}}{t_n - t_0}$$
гдје је t_i вријеме трајања i-тог захтјева,

- $R_{predicted}$ — предвиђено меморијско заузеће тј. укупан број бајтова који се налазе у младој (рајски и простори за преживљавање) и старој генерацији након једног комплетног скупљања. Рачуна се по формули
$$R_{predicted} = R_{current} - R_{freed}$$
гдје је R_{freed} просјек ослобођених бајтова приликом претходних скупљања.

Начин израчунавања поменутих вриједности биће детаљније описан у поглављу 4.2.

4.2 Имплементација система

Оптимизациони систем ради на побољшању клијент-сервер апликација чији се захтјеви обрађују секвенцијално и у једној нити. Дакле, меморија хипа у сваком тренутку припада тачно једном захтјеву.

У вријеме писања рада коришћене верзије Јава виртуелне машине и *native-image* се могу видјети на листингу 4.1.

```
$ native-image --version
native-image 24 2025-03-18
Java(TM) SE Runtime Environment Oracle GraalVM 24-dev+2.1 (build 24+2-jvmci-b01)
Java HotSpot(TM) 64-Bit Server VM Oracle GraalVM 24-dev+2.1 (build 24+2-jvmci-b01,
    mixed mode, sharing)
```

Листинг 4.1: Коришћене верзије

Технички детаљи

У имплементацији система користи се и агент за инструметализацију и интерфејс *Feature*.

Агент за инструментализацију омогућава *инструметализацију* кода. Инструментализација је убацивање новог кода у постојећи кôд. Нови кôд има за циљ да омогући пребројавање одређених догађаја односно прикупљање података који су од интереса [20].

Јава агент је специјално направљена *.jar*[3] (енг. *Java Archive*) датотека која се користи за инструментализацију бајткода. Јава бајткод, као што је већ речено, представља репрезентацију ниског нивоа која се извршава на виртуелној машини. За функционисање агента, неопходно је дефинисати методе:

- `premain` — за статичко учитавање агента коришћењем параметара који су задати опцијом `-javaagent` и
- `agentmain` — за динамичко учитавање агента.

У имплементацији система коришћен је агент који користи библиотеку ASM. ASM (eng. *Java bytecode manipulation and analysis framework*) је популарна библиотека за анализирање и модификацију Јава бајткода прије

[3]Један од формата који се користи за чување више датотека у једној.

или током извршавања програма. Библитека ASM омогућава трансформацију бајткода, пружајући разне алате за оптимизацију и инструментализацију бајткода [2].

Имплементирани агент може да инструментује више метода произвољног Јава пројекта на основу прослијеђених аргумената. Од инструментованог бајткода прави се извршива датотека. Приликом позивања агента могуће је користити следеће опције за задавање релевантних информација које описују инструментализацију једне методе:

- `instr` — служи за задавање путање до методе коју желимо да инструментујемо,
- `begin` — служи за задавање путање до методе чији кôд желимо да уметнемо на самом почетку методе *instr*,
- `end` — служи за задавање путање до методе чији кôд желимо да уметнемо на сам крај методе *instr*.

Уколико се инструментује више метода, као сепаратор између навођења опција за различите методе користи се карактер `;`. Пример аргумената[4] агента који се задају приликом изградње извршиве датотеке могу се видјети у коду 4.2.

```
agentArgs="/home/milica/agent/src/agent.jar=\
instr:\$ShopController\$Definition\$Exec.dispatch,\
begin:org/graalvm/nativeimage/GCHints.beginRequest()V,\
end:org/graalvm/nativeimage/GCHints.endRequest()V;\
instr:InvocationHandler.handle,\
begin:org/graalvm/nativeimage/GCHints.beginRequest()V,\
end:org/graalvm/nativeimage/GCHints.endRequest()V"
```

Листинг 4.2: Аргументи агента за инструментализацију

Опцијама из кода 4.2 задато је да се методе `beginRequest` и `endRequest` умећу на почетак односно на крај тијела методе `dispatch` која је дефинисана у класи `$ShopController$Definition$Exec`[5]. У другом случају, задато је уметање кода на почетак и крај методе `handle` класе `InvocationHandler`.

[4]Потпис методе `beginRequest()V` означава да је у питању метода која нема аргументе и повратна вриједност је типа `void`. У питању је JNI (енг. *Java Native Interface*) потпис методе који је облика `nazivMetode(tipoviArgumenata)tipPovratneVrijednosti`.

[5]Карактер \ у дефинисању опција користи се за одузимање специјалног значења карактера $.

Интерфејс *Feature* омогућава дефинисање класа чији се кôд извршава приликом изградње извршиве датотеке или у току њеног извршавања. Те класе представљају имплементацију интерфејса `org.graalvm.nativeimage.hosted.Feature` и њихове јединствене (енг. *singleton*) инстанце[6] се креирају коришћењем конструктора без аргумената тј. подразумијеваног конструктора.

У интерфејсу `Feature` декларисане су разне методе за руковање (енг. *handlers*) кодом који треба да се изврши у одређеној фази изградње (нпр. прије анализе или након компилације) или покретања извршиве датотеке (нпр. вријеме поставке). На примјер, ако је је за свако покретање програма потребно генерисати датотеку са јединственим именом за запис информација, то се може урадити чувањем времена у милисекундама на почетку извршавања, коришћењем класе која имплементира `Feature`.

Кратак преглед класа и интерфејса система

Најважније класе и интерфејси система описани су у наставку.

GCHintsSupport — интерфејс који садржи методе за инструментовање захтјева и испис прикупљених статистика.

AbstractGCHintsSupport — апстрактна класа која наслеђује интерфејс `GCHintsSupport` и имплементира његове методе. Због специфичне организације класа по пакетима, у овој класи дефинисане су апстрактне методе за дохватање меморијских бројача и статистика везаних за скупљање отпадака које постоје у систему *GraalVM* и користе се од стране подразумијеване полисе.

SerialGCHintsSupport — класа која наслеђује класу `AbstractGCHintsSupport` и имплементира све њене апстрактне методе.

RequestInfo — класа која садржи значајне информације једног захтјева.

GCHintsCollectionPolicy — класа која омогућава коришћење имплементираног система. Ова класа омогућава промјену величине младе генерације као и контролу тј. избор начина за скупљање отпадака.

[6] Образац *singleton* представља софтверско решење за инстанцирање јединственог објекта одређене класе.

Детаљнији опис наведених класа и интерфејса биће дат у поглављу 4.2.

Структура система

Серијски скупљач отпадака у оквиру система *GraalVM* дозвољава дјелимичну контролу процеса скупљања отпадака имплементацијом полиса за скупљање отпадака. Полисе представљају класе које имплементирају интерфејс `CollectionPolicy`. Подразумијевано понашање тј. начин позивања серијског скупљача отпадака дефинисано је класом `AdaptiveCollectionPolicy`[7] која имплементира претходно поменути интерфејс. За потребе система имплементирана је нова полиса `GCHintsCollectionPolicy` или савјетодавна полиса тј. класа која имплементира интерфејс `CollectionPolicy` која ће нам омогућити жељену промјену параметара хипа и контролисано позивање скупљача.

Имплементиране опције за рад са системом су:

- `EnableSerialGCHints` — омогућава инструментализацију агентом, прикупљање жељених статистика приликом извршавања
- `GCHintsOverrideDefaultGCPolicy` — омогућава коришћење савјетодавне полисе умјесто подразумијеване
- `LogGCHints` — омогућава испис информација о појединачним захтјевима и тренутном заузећу генерација
- `DumpGCHints` — омогућава чување информација о појединачним захтјевима у формату *json*
- `DumpGCHintsPath` — служи за задавање путање до директоријума за чување датотека у формату *json*

Такође, имплементирана је класа `GCHints` као и интерфејс `GCHintsSupport`. Класа `GCHints` омогућава кориснику да комуницира са интерно имплементираним системом. Она обезбеђује методе које се користе за инструментализацију као и методу за прикупљање статистика.

Интерфејс `GCHintsSupport` (Листинг 4.3) садржи потписе метода чија имплементација омогућава рад система и прикупљање одговарајућих статистика приликом извршавања програма. Метода `dumpJson` се користи за прикупљање

[7]Једноставности ради, у наставку ћемо користити термин подразумијевана полиса при чему мислимо на класу којом је дефинисано подразумијавано понашање серијског скупљача.

статистика у вези са извршавањем програма и њиховим записом у *.json* датотеци. Методе `beginRequest` и `endRequest` садрже кôд који се умеће на почетак односно на крај методе која се интрументује агентом.

```
public interface GCHintsSupport {
    void beginRequest();
    void endRequest();
    void dumpJson();
}
```

Листинг 4.3: Интерфејс *GCHintsSupport*

Имплементација интерфејса `GCHintsSupport` се може искористити за побољшање перформанси рада било којег скупљача отпадака. У овом раду интерфејс `GCHintsSupport` имплементира класа `AbstractGCHintsSupport` која профајлира апликације и подешава параметре серијског скупљача отпадака.

У коду 4.4 се налази имплементација класе која имплементира интерфејс `InternalFeature`[8]. Метода `initTimeStamp` се покреће на самом почетку програма гдје чува текуће вријеме за потребе генерисања назива *.json* датотеке за запис информација. Метода `dumpJson` се позива на самом крају извршавања програма и обезбјеђује испис сачуваних информација о захтјевима. Претходне методе се извршавају само у случају да је укључена опција `DumpGCHints` за испис информација што провјерава метода `isInConfiguration`.

```
class DumpGCHintsFeature implements InternalFeature {
    @Override
    public boolean isInConfiguration(IsInConfigurationAccess access) {
       return GCHintsOptions.DumpGCHints;
    }

    @Override
    public void duringSetup(DuringSetupAccess access) {
       RuntimeSupport.getRuntimeSupport().addShutdownHook(e -> dumpInstance.
           initTimeStamp());
       RuntimeSupport.getRuntimeSupport().addStartupHook( e -> dumpInstance.dumpJson());
    }
}
```

Листинг 4.4: Упрошћени кôд класе `DumpGCHintsFeature` која имплементира интерфејс `InternalFeature`

Класа `AbstractGCHintsSupport` чува информације о последњих n обрађених захтјева које су дефинисане класом `RequestInfo`. Класа `RequestInfo` садржи релевантне информације једног захтјева као што су:

[8] `InternalFeature` представља проширење интерфејса `Feature`.

- број алоцираних бајтова на почетку и крају захтјева — како би се израчунао укупан број алоцираних бајтова у захтјеву,
- вријеме на почетку и крају захтјева — како би се израчунало вријеме трајања једног захтјева,
- број инкременталних и комплетних скупљања — како би се графички приказало стање меморије прије и након урађеног скупљања.

Класа `RequestStructure` омогућава рад са последњих n обрађених захтјева. Интерно информације чува у виду низа у оквиру којег захтјеве додаје циклично, тј. у случају да је капацитет низа попуњен елементи се додају на почетак. Један елемент низа је типа `RequestInfo`.

Најважније методе класе `AbstractGCHintsSupport` а уједно и комплетног система су методе:

`beginRequest` која се помоћу агента умеће на почетку захтјева апликације и поставља одговарајућа поља која чувају информације са почетка захтјева. Поред тога, логују се и исписи о величини младе, старе генерације и хипа.

`endRequest` која се помоћу агента умеће на крај захтјева апликације, поставља одговарајућа поља која чувају информације са краја захтјева и рачуна укупно меморију алоцирану у захтјеву као и вријеме његовог трајања. Поред тога, логују се и исписи о величини младе, старе генерације и хипа. Кључни дијелови методе су услови у којима се провјерава да ли ће се ажурирати величина младе генерације и да ли ће се урадити неко скупљање. Главна израчунавања система дефинисана су овом методом:

1. Да ли ће се величина младе генерације ажурирати зависи од више фактора. Први услов који мора бити испуњен јесте да је укључена опција за савјетодавну полису. Даље се рачуна максимални број алоцираних бајтова по захтјеву у последњих n захтјева. Добијени број се користи као оптимална величина рајског простора. Величина младе генерације се рачуна као три пута величина рајског простора зато што једна трећина припада рајском, а преостале двије трећине просторима за преживљавање.

Метода `shouldUpdateYoungGenerationSize` враћа нову процентуалну величину младе генерације или -1 уколико савјетодавна полиса није укључена или је оптимална величина младе генерације иста као и прије. Приликом рачунања оптималне величине води се рачуна да проценат буде у интервалу $[1, 50]$ зато што млада генерација не би требала да буде већа од старе. Упрошћени кôд ове методе дат је на листингу 4.5.

```
public long shouldUpdateYoungGenerationSize(long allocatedInRequest,
    RequestStructure requests) {
       if (!GCHintsOptions.GCHintsOverrideDefaultGCPolicy) {
           return -1;
       }

       long edenSize = calculateOptimalEdenSize();
       long newYoungGenSizePercent = calculateOptimalYoungGenSizePercent(
           edenSize);

       return newYoungGenSizePercent != SerialGCOptions.
           MaximumYoungGenerationSizePercent ?
           newYoungGenSizePercent : -1;
    }
```

Листинг 4.5: Упрошћени кôд методе која рачуна оптималну величину младе генерације

2. Да ли ће се позвати инкрементално или комплетно скупљање зависи од испуњења услова дефинисаног неједнакошћу 4.1. Формула предвиђа ефикасност апликације приликом комплетног скупљања. Уколико је израчуната ефикасност боља од тренутне, врши се комплетно скупљање, у супротном, инкрементално. На листингу 4.6 приказан је упрошћени кôд методе из савјетодавне полисе која одређује врсту скупљања.

```
public boolean shouldCollectCompletely() {
       long currentRSS = computeRSS();
       double predictedRSS = currentRSS - getAvgFreedBytesDuringGC();
       long maxRequestDuration = requestStructure.maxRequestDuration();
       int numberOfReqThatWillBeSkipped = (int) Math.ceil(
           avgCompleteGCDurationTime / maxRequestDuration);

       int structureCapacity = requestStructure.getCapacity();
       return currentRSS * (structureCapacity - numberOfReqThatWillBeSkipped)
            / (predictedRSS * structureCapacity) >= 1;
    }
```

Листинг 4.6: Упрошћени кôд методе која одређује врсту скупљања

Начин употребе савјетодавне полисе

За коришћење савјетодавне полисе у произвољној клијент-сервер апликацији потребно је урадити следеће кораке:

1. задавање аргумената агенту:

```
agentArgs="putanjaDoAgentJarDatoteke=\
instr:putanjaDoZeljenogMetodaZaInstrumentalizaciju,\
begin:org/graalvm/nativeimage/GCHints.beginRequest()V,\
end:org/graalvm/nativeimage/GCHints.endRequest()V;"
```

   који се потом укључују коришћењем опције у фази изградње:
   `-J-javaagent:$agentArgs`

2. укључивање опција за коришћење полисе у фази изградње:
   `-H:+EnableSerialGCHints`
   `-H:+GCHintsOverrideDefaultGCPolicy`

3. укључивање опције у фази изградње за чување статистика у току извршавања програма у датотеци:
   `-H:+DumpGCHints`

Примјер коришћења претходно дефинисаних опција:

```
$ javac HelloWorld.java
$ native-image java.base/jdk.internal.org.objectweb.asm=ALL-UNNAMED -javaagent:"
    $agentargs" -H:+EnableSerialGCHints -H:+GCHintsOverrideDefaultGCPolicy -H:+
    DumpGCHints HelloWorld
```

Опцијом `java.base/jdk.internal.org.objectweb.asm=ALL-UNNAMED` укључују се интерни пакети из *java.base* модула за коришћење ASM агента.

Уколико се искључи опција `GCHintsOverrideDefaultGCPolicy` програм се у том случају извршава користећи подразумијевану полису. То је од значаја како би се прикупиле статистике употребом подразумијеване полисе и како би се могло направити поређење са савјетодавном полисом.

Глава 5

Евалуација система

За тестирање система користе се већ постојећи програми који представљају клијент-сервер апликације. Поређење подразумијеване полисе и савјетодавне полисе биће приказано на основу меморијских и временских мјерења појединачних захтјева која су сачувана у *json* датотекама.

5.1 Референтни програми

Бенчмарк или референтни програм (енг. *benchmark*) представља скуп стандардизованих програма и алата за мјерење перформанси који се користе за евалуацију ефикасности, брзине и других значајних карактеристика софтвера, библиотека или апликација.

Micronaut, Quarkus и *Spring* су радни оквири који садрже скупове програма који се користе за мјерење перформанси Јава микросервисних апликација система *GraalVM*. Приликом извршавања апликација прати се брзина покретања апликације, меморијска искоришћеност, број урађених операција по меморији и времену, вријеме компилације итд. По завршетку извршавања ови програми генеришу датотеку *bench-results.json* у којој су записане вриједности метрика за мјерење перформанси апликације.

Неки од најпопуларнијих бенчмарка Јава апликација су *DaCapo* [1], *Renaissance* [18], *Shopcart* [17], *Tika* [17] и *Petclinic* [17]. Евалуација имплементираног система у овом раду биће приказана на бенчмарку *Shopcart* и бенчмарку *Tika* јер су у питању микросервисне клијент-сервер апликације. Програми из бенчмарка *DaCapo* и *Renaissance* немају архитектуру клијент-сервер апликација па зато нису коришћени за евалуацију. Бенчмарк *Petclinic* је такође клијент-

сервер апликација али је због својих специфичности остављен за даљи рад. У наставку су дати кратки описи свих бенчмарка.

DaCapo [1] — скуп Јава апликација отвореног кода. Апликације се користе за мјерење перформанси скупљања отпадака, управљања нитима и управљања меморијом.

Renaissance [18] — скуп програма који су дизајнирани за тестирање перформанси Јава виртуелне машине на савременом хардверу који садржи више језгара.

Shopcart [17] — скуп програма који представљају клијент-сервер апликацију за куповину и дефинисани су у радном оквиру *Micronaut*. Апликација управља клијентима као и производима које купују.

Tika [17] — скуп програма који представљају клијент-сервер апликацију и користе се за анализу и чување података из различитих формата за записивање као што су PDF (енг. *Portable Document Format*), ODT (енг. *Open Document*) и други. Бенчмарк је дефинисан у радном оквиру *Quarkus*.

Petclinic [17] — скуп програма који представљају клијент-сервер апликацију клинике за кућне љубимце. Апликација управља власницима као и љубимцима које посједују. Бенчмарк је дефинисан у радном оквиру *Spring*.

Бенчмарци *Shopcart* и *Tika* се покрећу са великим бројем сличних захтјева. Због тога, ради боље прегледности, на графицима који садрже резултате ће бити приказани резултати једног сегмента захтјева[1].

5.2 Метрике за мјерење перформанси програма

Након што референтни програми заврше извршавање генерисана датотека *bench_results.json* садржи вриједности наредних метрика:

вријеме покретања (енг. *app-startup*) — вријеме које је потребно за покретање апликације,

[1]Графички приказ резултата било ког другог сегмента је сличан.

највећа пропусност (енг. *peak-throughput*) — максимална количина података или у контексту клијент-сервер апликација број захтјева који се обраде у датој јединици времена,

кашњење (енг. *latency*) — брзина одзива апликације,

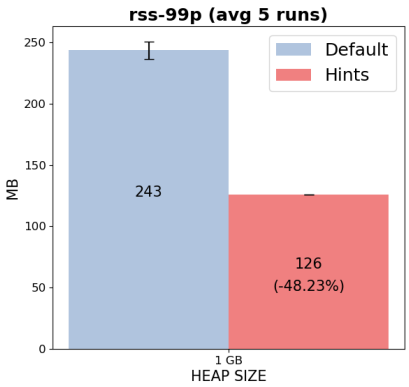
максимални RSS (енг. *resident set size*) — максимална количина меморије коју заузима програм у извршавању. Уколико се бенчмарк покрене са опцијом `--tracker=rsspercentiles` у датотеци се чувају подаци о заузећу меморије у перцентилима[2].

Одабир величина дијелова Јава хипа представља компромис између претходно наведених метрика. На примјер, уколико се већи дио хипа придружи млађој генерацији пропусност ће бити велика, али са друге стране и кашњење и траг (дефинисани у поглављу 4.1) могу бити велики.

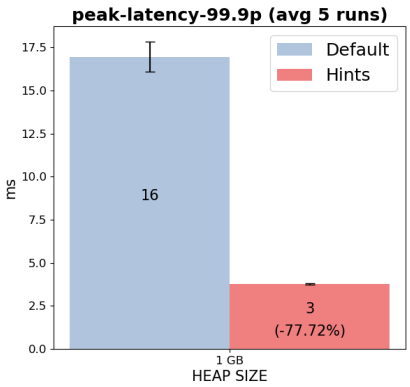
5.3 Резултати за бенчмарк *Shopcart*

На слици 5.1 су приказани графици који приказују разлике у вриједностима заузећа меморије, максималног кашњења и броја извршених операција по меморији и времену код подразумијеване и савјетодавне полисе на бенчмарку *Shopcart*. Максимално кашњење се мјери у перцентилима. На графицима су приказане и линије грешке које представљају стандардну девијацију тј. одступање вриједности појединачних покретања у односу на просјек вриједности тих покретања. Мјерења су добијена као просјек у `5` покретања програма за хип величине `1GB`. За друге величине хипа добијени су слични резултати и они неће бити приказани у овом раду. Добијени резултати показују следеће:

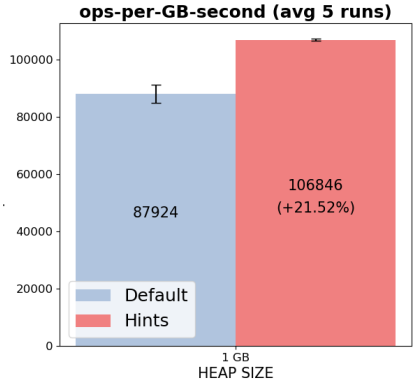
- максимално меморијско заузеће се смањило за 39.25% (Слика 5.1a),
- кашњење апликације се смањило за преко 48.95% (Слика 5.1b),
- број извршених операција по меморији и времену се повећао за 21.52% (Слика 5.1c). Овакав резултат се могао очекивати с обзиром на то да су и меморија и одзив апликације смањени.



(a) Меморијско заузеће, 99-и перцентил



(b) Максимално кашњење, 99.9-и перцентил



(c) Број извршених операција по меморији и времену

Слика 5.1: Вриједности метрика добијене коришћењем бенчмарка *Shopcart*

Графици на слици 5.1 показују да су мјерења меморије и операција по меморији и времену стабилна тј. вриједности током различитих покретања програма су приближно једнаке коришћењем савјетодавне полисе. Са друге стране, у случају мјерења кашњења апликације, подразумијевана полиса је стабилнија.

На слици 5.2 приказано је меморијско заузеће коришћењем подразумијеване и савјетодавне полисе током извршавања бенчмарка *Shopcart*. Одабрани сегмент за приказ резултата обухвата 1000 захтјева почевши од захтјева под

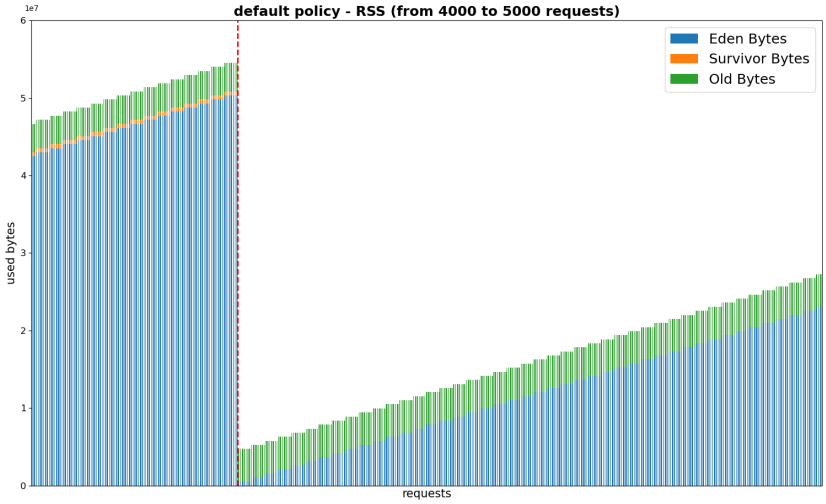
[2]Перцентил је број који представља проценат вриједности неког скупа вриједности који су мањи од задатог броја. Нпр. ако 99.9 перцентил има вриједност 1, то значи да је 99.9% вриједности датог скупа бројева мање од 1.

редним бројем 4000. Обојена линија представља меморијско заузеће текућег захтјева. Плавом бојом су означени бајтови рајског простора, наранџастом бојом бајтови простора за преживљавање и зеленом бојом бајтови из старе генерације. Значење вертикалне линије на графику:

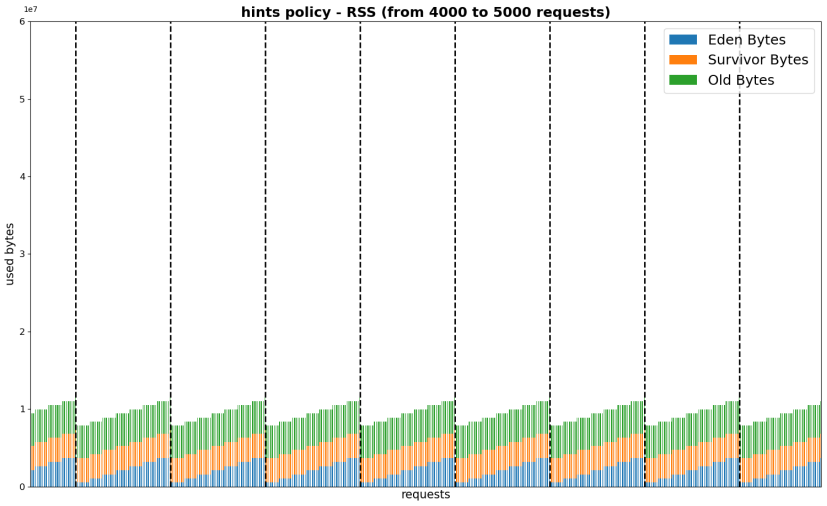
- испрекидана црна линија — десило се инкрементално скупљање ван захтјева,
- испрекидана црвена линија — десило се комплетно скупљање ван захтјева.

Подразумијевана полиса пусти да се напуни рајски простор након чега се дешава инкрементално скупљање ван захтјева. Бенчмарк је такав да се врло мало објеката налази у старој генерацији.

Код подразумијеване полисе много чешће се дешава инкрементално скупљање, зато што је смањена димензија рајског простора, али самим тим је и заузеће меморије смањено. Меморијско заузеће сваког појединачног захтјева подразумијеване полисе не прелази $5 \cdot 10^7$ бајтова док је код савјетодавне полисе заузеће значајно мање и не прелази $1.2 \cdot 10^7$ бајтова. У случају подразумијеване полисе је једном било урађено инкрементално скупљање док је у случају савјетодавне полисе било урађено осам колеција тј. однос урађених скупљања је $1 : 8$. С обзиром на то да је млада генерација смањена, инкрементално скупљање мање кошта тако да кашњење апликације неће бити погоршано.



(a) Меморијско заузеће са подразумијеваном полисом

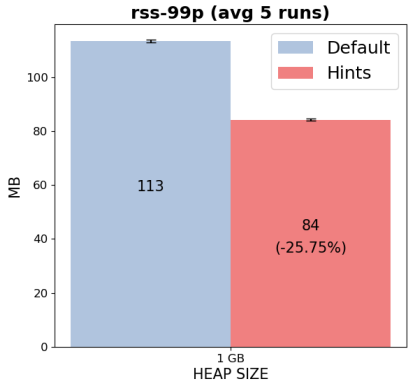


(b) Меморијско заузеће са савјетодавном полисом

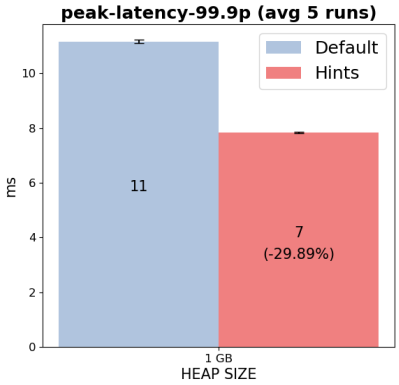
Слика 5.2: Вриједности метрика добијене коришћењем бенчмарка *Shopcart*

5.4 Резултати за бенчмарк *Tika*

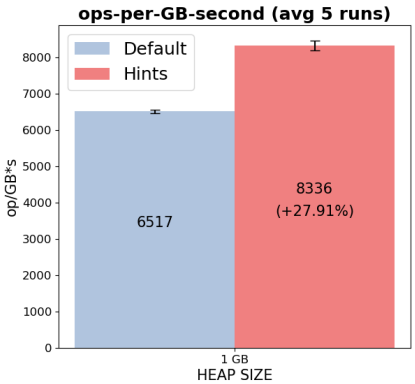
На слици 5.3 су приказани графици који приказују разлике у вриједностима заузећа меморије, максималног кашњења и броја извршених операција по меморији и времену код подразумијеване и савјетодавне полисе на бенчмарку *Tika*. Као и код бенчмарка *Shopcart* на графицима су приказане линије грешке и метрике за заузеће меморије и кашњење као мјерну јединицу користе перцентиле. Мјерења су добијена као просјек у `5` покретања програма на хипу величине `1GB`.



(a) Меморијско заузеће, 99-и перцентил



(b) Максимално кашњење, 99.9-и перцентил



(c) Број извршених операција по меморији и времену

Слика 5.3: Вриједности метрика добијене коришћењем бенчмарка *Tika*

Резултати показују следеће:

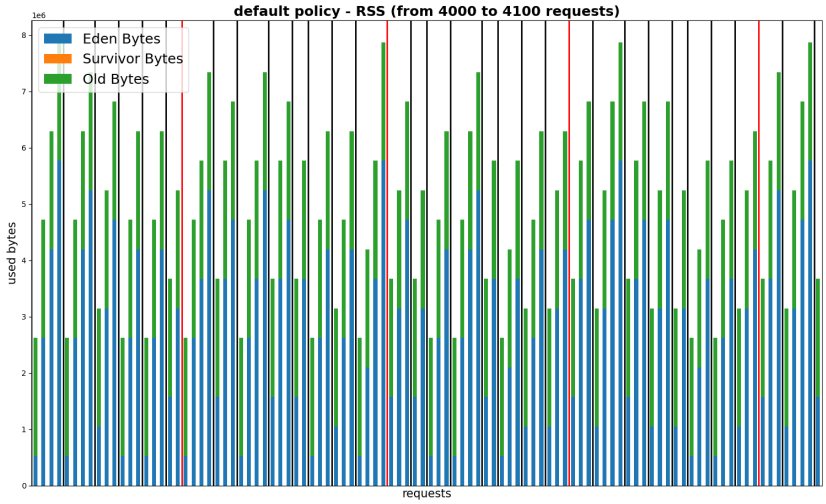
- максимално меморијско заузеће се смањило за 25.75% мјерено у перцентилима (Слика 5.3a),
- кашњење апликације се смањило за 27.62% мјерено у перцентилима, (Слика 5.3b)
- број извршених операција по меморији и времену се повећао за 27.91% (Слика 5.3c).

Такође, графици показују да су спроведена мјерења у случају и једне и друге полисе стабилна.

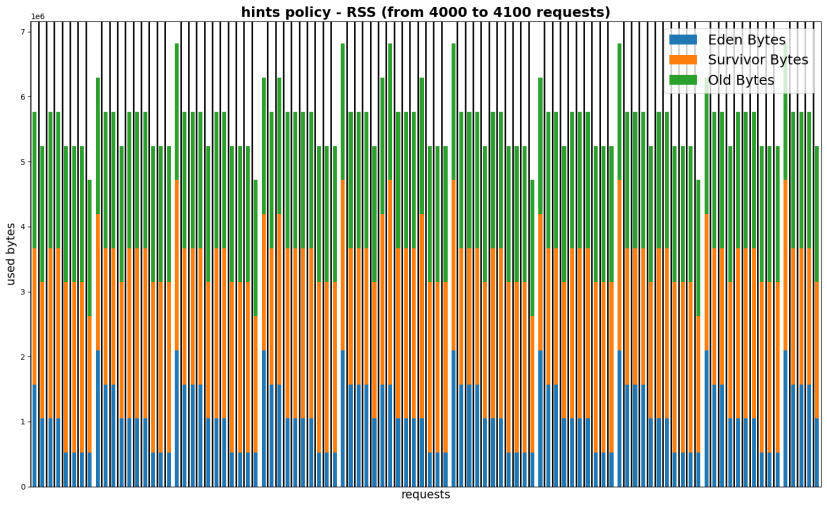
На графику меморијског заузећа (Слика 5.4) приказано је дешавање инкременталне, односно комплетног скупљања у току захтјева коришћењем подразумијеване полисе и инкременталног скупљања током захтјева коришћењем савјетодавне полисе. Одабрани сегмент за приказ резултата обухвата 100 захтјева почевши од захтјева под редним бројем 4000. Слично као на графику 5.2, обојена линија представља меморијско заузеће текућег захтјева (плавом бојом су означени бајтови рајског простора, наранџастом бојом бајтови простора за преживљавање и зеленом бојом бајтови из старе генерације). Значење вертикалних линија на графику:

- пуна црна линија — десило се инкрементално скупљање у захтјеву,
- пуна црвена линија — десило се комплетно скупљање у захтјеву.

Позивање комплетног скупљања приликом коришћења подразумијеване полисе у току захтјева може бити разлог због којег савјетодавна полиса има мање кашњење и више операција по меморији и времену.



(a) Меморијско заузеће са подразумијеваном полисом



(b) Меморијско заузеће са савјетодавном полисом

Слика 5.4: Вриједности метрика добијене коришћењем бенчмарка *Shopcart*

Глава 6

Закључак

У оквиру овог рада описан је начин скупљања отпадака у програмском језику Јава на Јава виртуелној машини као и у систему *GraalVM*. За потребе рада коришћен је компилатор *GraalVM Native Image* као и агент за инструментализацију Јава бајткода. Главни предмет рада је побољшање алгоритма за скупљање отпадака у клијент-сервер апликацијама динамичким подешавањем параметара младе генерације у систему *GraalVM*. Подешавање параметара се врши на основу меморијске анализе претходних захтјева у циљу максималног искоришћења младе генерације како би се минимизовало позивање комплетног скупљања.

Основни допринос ове тезе је развијање савјетодавне полисе, односно методологије предвиђања правих тренутака за позив скупљача отпадака код клијент-сервер апликација. Резултат развијене методологије је смањена количина меморије као и повећан број захтјева који се ураде у секунди у клијент-сервер апликацијама које се покрећу коришћењем система *GraalVM*. Смањење меморијског заузећа у случају бенчмарка *Shopcart* износи 39.25% а у случају бенчмарка *Tika* износи 25.75% на хипу величине `1GB`. Смањена количина меморије је од великог значаја јер апликација у том случају употребљава мање ресурса. Самим тим, смањује се цијена покретања клијент-сервер апликација. Кашњење апликације у случају бенчмарка *Shopcart* је смањено за 48.95%, а у случају бенчмарка *Tika* за 27.62% док је број извршених операција по меморији и времену за бенчмарк *Shopcart* смањен за 34.41% али у случају бенчмарка *Tika* повећан за 27.91%.

Постоји неколико праваца даљег развоја. Један од праваца је манипулација величином простора за преживљавање како би се смањила употреба

старе генерације и самим тим још више смањило позивање комплетног скупљања. Динамички се поред величине младе генерације и хипа може мијењати и број и величина простора за преживљавање. Поред тога, алгоритам је могуће побољшати тако да се може примјењивати и на клијент-сервер апликацијама код којих захтјеви за обраду пристужу паралелно. У том случају, мијења се начин приступања меморијским бројачима јер тада у једном тренутку ресурси од више захтјева могу бити присутни истовремено на хипу.

Тренутна имплементација система функционише за серијски скупљач отпадака. Да би савјетодавна полиса могла да се користи и у случају других скупљача, довољно је имплементирати интерфејс `GCHintsSupport`.

Евалуација резултата урађена је над представницима радних оквира *Micronaut* и *Quarkus*. Поменути представник *Petclinic* радног оквира *Spring* такође представља клијент-сервер апликацију која се може користити за даље тестирање и развој овог система.

Библиографија

[1] 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.

[2] Eric Bruneton. Asm 3.0 a java bytecode engineering library. *URL: http://download. forge. objectweb. org/asm/asmguide. pdf*, 2007.

[3] Rodrigo Bruno and Paulo Ferreira. A study on garbage collection algorithms for big data environments. *ACM Computing Surveys (CSUR)*, 51(1):1–35, 2018.

[4] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, nov 1970.

[5] Oracle Company. Garbage-First Garbage Collector, 2023. on-line at: `https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1_gc.html`.

[6] Oracle Company. GraalVM, 2023. on-line at: `https://docs.oracle.com/en/java/javase/12/gctuning/available-collectors.html#GUID-45794DA6-AB96-4856-A96D-FDE5F7DEE498`.

[7] Oracle Company. GraalVM, 2023. on-line at: `http://gcc.gnu.org/`.

[8] Oracle Company. GraalVM Native Image, 2023. on-line at: `https://www.graalvm.org/latest/reference-manual/native-image/`.

[9] Oracle Company. GraalVM Native Image Basics, 2023. on-line at: `https://docs.oracle.com/en/graalvm/jdk/21/docs/reference-manual/native-image/basics/#native-image-basics`.

[10] Oracle Company. GraalVM Native Image Build Output, 2023. on-line at: `https://docs.oracle.com/en/graalvm/jdk/21/docs/reference-manual/native-image/overview/BuildOutput/#build-stages`.

[11] Oracle Company. Java Garbage Collection Basics, 2023. on-line at: `https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html`.

[12] Oracle Company. Java Garbage Collection Basics, 2023. on-line at: `https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html`.

[13] Oracle Company. Memory Managment at Image Runtime, 2023. on-line at: `https://docs.oracle.com/en/graalvm/enterprise/20/docs/reference-manual/native-image/MemoryManagement/#serial-garbage-collector`.

[14] L Peter Deutsch and Daniel G Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, 1976.

[15] Santosh Kumar. A review on client-server based applications and research opportunity. *International Journal of Recent Scientific Research*, 10(7):33857–3386, 2019.

[16] Haroon Shakirat Oluwatosin. Client-server model. *IOSR Journal of Computer Engineering*, 16(1):67–71, 2014.

[17] 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.

[18] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, 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 *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 31–47, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wogerer, Peter B. Kessler, Oleg Pliss, and Thomas Wurthinger. Initialize Once, Start Fast: Application Initialization at Build Time, 2019. on-line at: `http://www.christianwimmer.at/Publications/Wimmer19a/Wimmer19a.pdf`.

[20] Милена Вујошевић Јаничић. Верификација софтвера, 2023. on-line at: `http://www.verifikacijasoftvera.matf.bg.ac.rs/vs//verifikacija_softvera.pdf`.

[21] Александар Картељ, Владимир Филиповић, and Душан Тошић. Објектно оријентисано програмирање, 2022.

Биографија аутора

Милица Карличић рођена је 18.01.2000. године у Бијелом Пољу (Црна Гора). Основну школу Ристо Ратковић у Бијелом Пољу завршила је 2014. године. Упоредо је похађала и државну школу за основно музичко образовање гдје је свирала флауту. И у једној и у другој школи остварила је титулу ђака генерације. Након тога, похађала је општи смјер у гимназији Милоје Добрашиновић, коју је завршила 2018. године. Исте године, уписала је Математички факултет Универзитета у Београду, смјер Рачунарство и математика у оквиру модула Математика. На истом је дипломирала у септембру 2022. година са просјечном оцјеном 9.40. Након дипломирања, уписала је мастер студије на истом факултету, на смјеру Математика и рачунарство, на ком је успјешно положила све испите са просјечном оцјеном 10.0. Исте године изабрана је за сарадника у настави на Катедри за рачунарство и информатику. Држала је вјежбе из предмета: Објектно-оријентисано програмирање, Лексичка анализа и њене примене и Компилација програмских језика. Од октобра 2023. године запослена је на истраживачкој пракси у компанији *Oracle*. Области интересовања укључују системски софтвер, преводиоце и програмске језике.