Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Nikolaj Janko

Predelava javanskega navideznega stroja za štetje ukazov zložne kode

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Tomaž Dobravec

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Programski jezik Java je eden od najbolj priljubljenih in uporabljenih sodobnih programskih jezikov. Uporaba javanskega navideznega stroja omogoča prenosljivost programov med različnimi sistemi, hkrati pa delno upočasni izvajanje. Še posebej so te zakasnitve pomembne pri natančnem merjenju časa izvajanja, saj med izvajanje lahko pride do nepredvidljivih okoliščin, ki spremenijo časovne rezultate. Boljša mera za časovno analizo je zato število ukazov zložne kode, ki se izvedejo med delovanjem programa. V diplomskem delu preučite možnosti za štetje ukazov zložne kode izvajanega programa. Poiščite odprtokodni javanski navidezni stroj in popravite njegovo programsko kodo tako, da bo omogočal štetje uporabe posameznih ukazov. Preučite tudi možnost za štetje kumulativne porabe pomnilnika med izvajanjem programa. V jeziku Java izdelajte knjižnico, ki uporabniku omogoča preprost dostop do podatkov, ki jih ustvari prirejen navidezni stroj.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Nikolaj Janko, z vpisno številko **63100248**, sem avtor diplomskega dela z naslovom:

Predelava javanskega navideznega stroja za štetje ukazov zložne kode

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Tomaža Dobravca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 15. septembra 2014 Podpis avtorja:

Na prvem mestu se zahvaljujem svojemu mentorju doc. dr. Tomažu Dobravcu, ki me je s svojimi napotki in nasveti vodil pri izdelavi diplomske naloge.

Posebno se zahvaljujem materi, družini in bližnjim za vso podporo, ki sem je bil deležen v času študija.

Zahvaljujem se tudi vsem sošolcem, ki so mi tako ali drugače pomagali pri študiju, in Teji Cetinski, ki je s svojimi lektorskimi sposobnostmi popravila nenamerne napake v diplomski nalogi.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
JVM	Java virtual machine	Javanski navidezni stroj
JRE	Java runtime environment	Javansko izvajalno okolje
JDK	Java development kit	Javanska razvojna orodja
JNI	Java native interface	Javanski domorodni vmesnik
JIT	Just in time	Le v pravem času
AUR	Arch user repository	Uporabniški repozitorij Arch
VMEP	Virtual machine entry point	Vstopna točka navideznega stroja
QS	Quick sort	Hitro urejanje

Povzetek

Cilj, ki smo ga zastavili pri izdelavi diplomske naloge, je implementirati ali predelati že obstoječi javanski navidezni stroj (JVM) tako, da bo uporabniku za pognan program omogočal vpogled v statistiko izvedenih javanskih vmesnih ukazov. Ta funkcionalnost bo med drugim omogočala lažjo analizo izvajanja algoritmov v javanskem okolju. Po preučevanju teorije jave in javanskega navideznega stroja smo se odločili za predelavo obstoječega javanskega navideznega stroja. Izbrali smo JamVM, preprostejši, odprtokodni javanski navidezni stroj pod licenco GNU [1]. Idejo smo razdelili na minimalno predelavo JamVM-ja in implementacijo javanske knjižnice. Javanska knjižnica bo znotraj pognanega uporabniškega programa omogočila dostop do informacij in statistike o izvedenih javanskih vmesnih ukazih. Na koncu smo implementirali in prikazali še enostaven primer uporabe naše rešitve.

Ključne besede: Javanski navidezni stroj, Vstopna točka navideznega stroja, Java.

Abstract

The objective of the thesis was to implement or modify an existing Java virtual machine (JVM) in a way that it will allow insight into statistics of the executed Java instructions of an executed user program. The functionality will allow analysis of the algorithms in Java environment. After studying the theory of Java and Java virtual machine, we decided to modify an existing Java virtual machine. We chose JamVM which is a lightweight, open-source Java virtual machine under GNU license [1]. The idea was divided into minimal modification of JamVM and implementation of own Java library. The Java library will, within the user application program, allow access to information and statistics of executed Java instructions. At the end, we have implemented and presented a simple use of our solution.

Keywords: Java virtual machine, Virtual machine entry point, Java.

Poglavje 1

Uvod

Časovna in prostorska zahtevnost sta kljub vedno večji zmogljivosti računalnikov v svetu računalništva zelo pomembni. Vsak, ki se večkrat sreča s programiranjem, kmalu ugotovi, da se lahko že dokaj enostaven program izvaja dlje časa, kot bi to od njega pričakovali. Da bi rešili težavo časovne ali prostorske zahtevnosti, preden napišemo program, običajno poiščemo čim bolj optimalen algoritem za dano nalogo.

Ko govorimo o algoritmu, govorimo o načrtu programa. Cilj je torej napisati tak algoritem, ki se bo na splošno izvajal najhitreje in porabil čim manj prostora v pomnilniku. Tukaj je poudarek na "na splošno", saj algoritem običajno ni vezan na določen procesor ali določeno arhitekturo računalnika, na katerem se bo izvajal končni program. Vsekakor velja, da bo na splošno izpiljen algoritem dober približek najbolj optimalni rešitvi, zato ga lahko vzamemo kot izhodišče in pri končnem piljenju upoštevamo še sámo arhitekturo sistema, na katerem se bo izvajal končni program.

Ko želimo napisati čim bolj optimalen program, ki se bo izvajal hitro in s katerim bomo imeli velik nadzor nad pomnilnikom, izberemo nizkonivojski programski jezik. Nizkonivojski programski jezik, kot je C, se pred izvajanjem prevede v niz strojnih ukazov, ki jih lahko izvedemo na določenem fizičnem procesorju. Kljub temu se vse pogosteje uporabljajo visokonivojski programski jeziki, ki se po večini izvajajo na navideznem procesorju oziroma

v navideznem izvajalnem okolju. Med fizičnim procesorjem in programom, ki se izvaja, se pojavi vmesni programski sloj, ki predstavlja navidezno izvajalno okolje. Zaradi vmesnega sloja se takšni programi izvajajo počasneje, vendar pa prednosti pretehtajo to slabost. Glavni dve prednosti sta sistemska neodvisnost in varnost.

Pri tem projektu smo se osredotočili na izvajanje končnega programa na namenski arhitekturi. Naš cilj je napisati orodje, ki bi pomagalo analizirati že delujoči program in bi tako pripomoglo k še boljši optimizaciji. Iščemo programsko orodje, ki požene napisan program in izpiše statistiko izvedenih strojnih ukazov. Če poleg statistike izvedenih strojnih ukazov poznamo tudi to, kateri strojni ukazi so najbolj časovno ali prostorsko potratni, lahko ugotovimo, zakaj je izvajanje programa počasno. Tako lahko skupaj z nekaj znanja o prevajalnikih oziroma prevajanju programa ali pa z malo poskušanja zreduciramo frekvenco časovno zahtevnih strojnih ukazov. S tem dejanjem lahko zelo pohitrimo izvajanje programa.

Za arhitekturo smo izbrali dobro poznano javo. Java je razvojno okolje s programskim jezikom in svojim programskim izvajalnim okoljem. Prav tako je zelo dobro dokumentirana in veliko uporabljena. Zelo pomemben del javanskega izvajalnega okolja (JRE) predstavlja javanski navidezni stroj, ki izvaja javanske vmesne ukaze. Če torej želimo šteti izvedene javanske vmesne ukaze, jih moramo šteti prav v javanskem navideznem stroju. Implementirati ali predelati je potrebno JVM tako, da bo poleg izvajanja tudi štel javanske vmesne ukaze in statistiko prikazal uporabniku. Odločili smo se za predelavo obstoječega javanskega navideznega stroja, ker bi bila implementacija novega preobsežna naloga. Tako smo izbrali eno od minimalističnih implementacij javanskega navideznega stroja, JamVM, in mu s predelavo dodali potrebno funkcionalnost.

V prvem delu diplomske naloge bomo opisali nekaj teorije o javi. Pogledali si bomo, kakšen je princip jave, kako cel sistem deluje in sámo strukturo oziroma sestavo. V drugem delu pa si bomo ogledali implementacijo rešitve in kako smo dosegli cilj, ki smo si ga zastavili.

Poglavje 2

Java

Poznamo več visokonivojskih razvojnih okolij, kot so Python, .NET, Java in tudi LaTeX. Java je poleg .NET edini bolj znani programski jezik, ki se prevede v vmesno kodo, ki se nato izvaja na za to namenjenem navideznem stroju. Drugi visokonivojski programski jeziki, kot so JavaScript, Python, ..., se po večini interpretirajo. Programi, napisani v jezikih, ki se interpretirajo, so običajno počasnejši, ker gre za potratne operacije, kot je obdelava nizov. Tudi pri jezikih, kot je java, pride do interpretacije, vendar ta ne vključuje zahtevnih operacij, kot je obdelava nizov.

Java je predmetno orientiran programski jezik, podoben je C++, a brez kompleksnih in nevarnih lastnosti. Ker je bila java že prvotno namenjena prenašanju preko različnih medijev in med različnimi arhitekturami, je prevedena koda prilagojena temu namenu.

2.1 Struktura

Da se bomo lažje pogovarjali o javi, si bomo v tem razdelku pogledali vse pomembnejše javanske gradnike. Najpomembnejši del jave je javanski navidezni stroj (angl. Java virtual machine, JVM). Javanski navidezni stroj je program, ki poganja javanske programe; je neke vrste navidezni računalnik, ki ima vse od pomnilnika do procesorja. Kljub temu javanski navidezni stroj

nima vseh potrebnih sestavin, da bi poganjal programe. Manjkajo mu še javanske sistemske knjižnice. Te knjižnice običajno niso del javanskega navideznega stroja, kar pa se lahko razlikuje od implementacije do implementacije. Naslednji pomemben gradnik so torej javanske sistemske knjižnice, ki jih običajno najdemo v datoteki rt.jar. Če združimo ta dva gradnika, dobimo tako imenovano javansko izvajalno okolje (angl. Java runtime environment, JRE). Med napisanim in pognanim programom manjka samo še prevajalnik. Javanski prevajalnik najdemo v javanskem razvojnem paketu, na kratko JDK (Java Development Kit). JDK vsebuje vse od javanskega navideznega stroja in javanskih sistemskih knjižnic do prevajalnika in različnih pripomočkov za razvijalce. Celotno strukturo lahko vidimo na sliki 2.1.

Slika 2.1: Struktura JDK [2]

Podrobnejše informacije o strukturi jave je mogoče najti v specifikaciji "The Java Language Specification" [6].

2.2 Zgodovina

V zgodnjih devetdesetih letih prejšnjega stoletja je ena od ekip iz podjetja Oracle verjela, da je prihodnost v združenju različnih digitalnih naprav in računalnikov. Skupina, ki jo je vodil `James Goslin`, je tako razvila programski jezik java. Leta 1995 je skupina predstavila na novo razviti programski jezik. Prvotno so ciljali na kabelske televizijske sisteme, vendar je bila ideja za tisti čas preveč napredna. Velik potencial se je našel v internetu, ki se je v tistem času ravno dobro razcvetel. Tako je java postala in je še danes eno izmed obveznih orodij, medtem ko brskamo po multimedijsko bogatih spletnih straneh.

Java se je do današnjega dne zelo razvila in jo najdemo ne samo na internetu, vendar tudi v mobilnih napravah, za pogon igric, v navigacijskih sistemih, v poslovnih sistemih in na mnogo drugih področjih računalništva.

2.3 Primer uporabe jave

Na primeru si bomo pogledali, kako napišemo in poženemo v javi napisan program. Primer bomo izvedli na sistemu `Linux` v lupini `bash`. Na sistemu je potrebno imeti naložen paket JDK.

Najprej ustvarimo navadno tekstovno datoteko s končnico `.java`. V našem primeru smo ustvarili datoteko `Program.java`. Datoteka, ki bo izhodišče za zagon programa, mora vsebovati javni razred z istim imenom, kot je ime datoteke. Ta razred mora vsebovati še javno statično metodo `main`, v kateri se bo tok programa začel. Argumente programa prejmemo v argumentu `args` metode `main`.

```
1 class Program{
2   public static void main(String[] args){
3     System.out.println(”Pozdravljen svet!!”);
4   }
5 }
```

Koda 2.1: Datoteka `Program.java`

Primer kode 2.1 prikazuje preprost v javi napisan program. Sledi prevajanje programa. V paketu JDK najdemo prevajalnik datotek `.java`, imenuje se javac. Kot argument sprejme ime datoteke `.java`, jo prevede in v delovnem imeniku ustvari datoteko `.class`, ki ji rečemo razredna zbirka. Če program vsebuje sintaktično napako ali pomanjkljive reference, se prevajalnik zaključi in javi napako. Razredna zbirka je rezultat prevajanja in vsebuje vse potrebne informacije o prevedenih razredih, reference na druge razrede in, seveda najbolj pomembno, javansko vmesno kodo vseh metod. Kaj vse vsebuje razredna zbirka in bolj natančen opis njene strukture lahko najdemo v podpoglavju 2.4.2 na strani 10.

```
1 javac Program.java
```

Koda 2.2: Prevajanje programa

Po prevajanju, ko pridobimo vse potrebne razredne zbirke, lahko poženemo program. Ob zagonu javanskega navideznega stroja moramo posredovati direktorije, ki vsebujejo razredne zbirke. Direktoriji, po katerih javanski navidezni stroj išče razredne zbirke oziroma razrede, se imenujejo classpath. Običajen javanski navidezni stroj vzame za classpath tudi trenutni delovni direktorij. Da bo javanski navidezni stroj vedel, kje začeti izvajanje, mu moramo posredovati še ime razreda, ki vsebuje metodo `main`.

Če se nahajamo v direktoriju, v katerem smo prevedli program, lahko program poženemo s preprostim ukazom, kot je prikazan v primeru 2.3.

```
1 java Program
```

Koda 2.3: Zagon programa

Po zagonu programa se standardni izhod in vhod lupine `bash` povežeta s pognanim programom, kar nam omogoča, da vnašamo podatke in opazujemo izpis programa. V našem primeru se izpiše “Pozdravljen svet”.

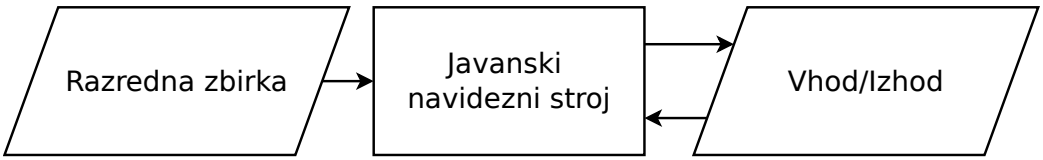
2.4 Javanski navidezni stroj

Ker želimo predelati javanski navidezni stroj, si bomo še podrobneje ogledali njegovo delovanje in sestavo. Na spletni strani Oracle najdemo specifikacijo, ki zelo dobro opisuje delovanje in sestavo javanskega navideznega stroja [5]. Ker je specifikacija zelo obsežna, bomo tukaj povzeli le najpomembnejše stvari.

Kot smo to že omenili v uvodu poglavja 2, je javanski navidezni stroj zelo pomemben del jave. Je tisti del, ki premosti javansko okolje z operacijskim sistemom in arhitekturo sistema, na katerem poganjamo javanske programe. Torej, javanski navidezni stroj je tisti del jave, ki naredi javanske programe sistemsko neodvisne, sam javanski navidezni stroj pa ostaja sistemsko odvisen.

Poleg sistemske neodvisnosti nam javanski navidezni stroj nudi še varnost. Ker programi, pognani preko javanskega navideznega stroja, tečejo izolirano od operacijskega sistema, ne morejo posegati in škoditi samemu sistemu ali zlorabljati pomanjkljivo zaščitenih administrativnih operacij. Prav tako nam java nudi tudi izolacijo med pognanimi javanskimi programi.

Glavna naloga javanskega navideznega stroja je, da zna prebrati razredno zbirko in izvesti javansko vmesno kodo. Pri tem lahko uporablja različne vhode in izhode.



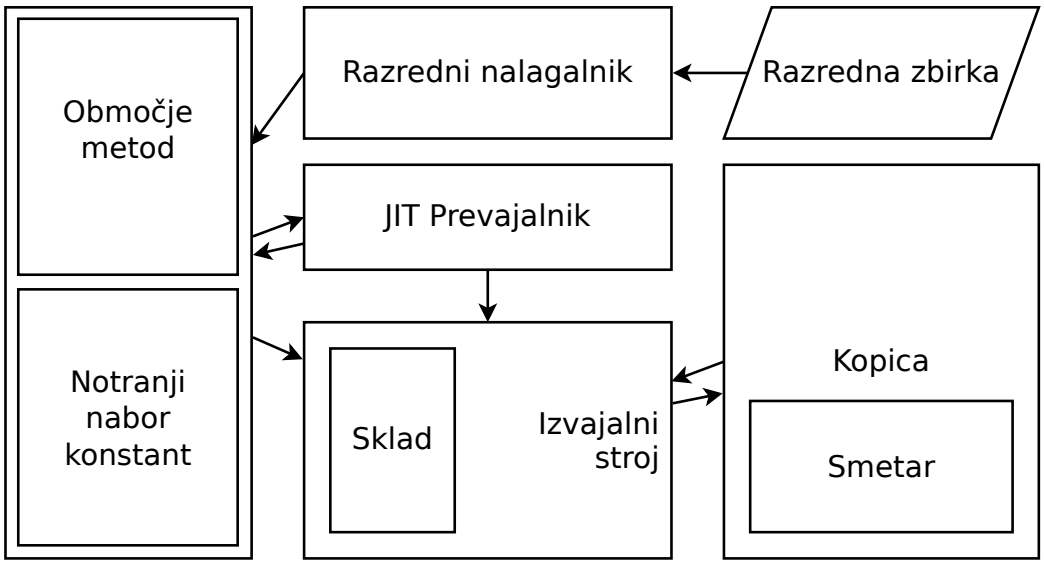
Slika 2.2: Diagram JVM

2.4.1 Struktura javanskega navideznega stroja

Struktura javanskega navideznega stroja opisuje zasnovo in ne dejansko implementacijo stroja. Vsaka implementacija javanskega navideznega stroja mora slediti specifikaciji. Delujoča implementacija javanskega navideznega stroja mora znati prebirati razredne zbirke in pri določenem vhodu programa vrniti pravilen in konsistenten rezultat.

Glavni gradniki javanskega navideznega stroja

Diagram na sliki 2.3 prikazuje glavne gradnike javanskega navideznega stroja in povezave med njimi.



Slika 2.3: Diagram glavnih gradnikov javanskega navideznega stroja

Opis in vloge gradnikov iz slike 2.3:

Razredna zbirka

Razredna zbirka (angl. Class file) je datoteka, ki nosi opis javanskega razreda ali vmesnika in javansko vmesno kodo. Njeno strukturo si lahko natančneje ogledamo v poglavju 2.4.2 na strani 10.

Razredni nalagalnik

Naloga razrednega nalagalnika (angl. Class loader) je, da prebere razredno zbirko, jo razčleni in preveri njeno veljavnost. Iz prebrane razredne zbirke sestavi notranjo predstavitev posameznega razreda in jo shrani v notranji nabor konstant.

Območje metod

Območje metod (angl. Method area) je del pomnilnika v javanskem navideznem stroju, kjer je shranjena javanska vmesna koda metod ali že prevedena in sistemsko odvisna strojna koda, pripravljena za izvajanje.

Notranji nabor konstant

Notranji nabor konstant (angl. Internal constant pool) hrani notranjo predstavitev vseh naloženih razredov in vmesnikov. Vsebuje lahko tudi vse konstante nizov. Oblika ali struktura predstavitve razredov je lahko od implementacije do implementacije drugačna.

Prevajalnik JIT

Prevajalnik JIT (angl. JIT Compiler) je del javanskega navideznega stroja, ki prevaja še neprevedeno metodo iz javanskega vmesnega jezika v sistemsko odvisen strojni jezik. S tem pridobimo večjo hitrost izvajanja, saj se prevedena metoda izvede na fizičnem procesorju brez potratnih skokov, ki so običajno prisotni pri interpretaciji posameznih javanskih vmesnih ukazov. Obstajajo tudi javanski navidezni stroji, ki samo interpretirajo javanske vmesne ukaze in nimajo prevajalnika JIT.

Izvajalni stroj

Izvajalni stroj (angl. Execution engine) skrbi za tok izvajanja programa. V njem se nahaja sklad, nad katerim se izvajajo operacije. Upravlja s skladom in drugimi pomnilniki, skrbi za klice metod in dejansko izvaja strojne ukaze. Z drugimi besedami bi lahko izvajalnemu stroju rekli kar procesor, saj se v njem izvaja procesiranje ukazov.

Kopica

Kopica (angl. Heap) je del pomnilnika, v katerem so shranjeni objekti oziroma primerki razredov in polja različnih tipov. Da se kopica redno prazni, skrbi Smetar.

Smetar

Smetar (angl. Garbage collector) skrbi, da se sprosti tisti del pomnilnika v kopici, do katerega je izgubljena referenca (do katerega ne moremo več dostopati).

`Primer:` Recimo, da v metodi ustvarimo lokalno spremenljivko in vanjo shranimo referenco pravkar ustvarjenega objekta. Ker smo objekt pravkar ustvarili, lahko do njega dostopamo le preko naše lokalne spremenljivke. Ob koncu metode se lokalne spremenljivke pobrišejo in takrat ostanemo brez dostopa do našega objekta, saj je edina referenca do njega obstajala v lokalni spremenljivki. Smetar po enem od algoritmov (odvisno od implementacije) ugotovi, da del pomnilnika, v katerem je shranjen objekt, ne more biti več dostopen, zato tisti del pomnilnika sprosti.

2.4.2 Format razredne zbirke

Prevedena koda, pripravljena za izvajanje, je shranjena v posebnem binarnem formatu. Ta format se imenuje format razredne zbirke (angl. Class file format). Format razredne zbirke točno definira javanski razred ali vmesnik skupaj z javansko vmesno kodo (angl. Java byte code). V večini primerov so za shranjevanje razredov ali vmesnikov uporabljene razredne zbirke (datoteke s končnico `.class`), zato bomo v našem primeru govorili kar o razrednih zbirkah.

Cela razredna zbirka je sestavljena iz serije bajtov (8 bitov). Torej so tudi vse večje vrednosti večkratniki enega bajta oziroma osmih bitov. Za lažjo predstavitev strukture bomo definirali tri tipe: u1 (1 bajt), u2 (2 bajta) in u4 (4 bajti), ki bodo predstavljali določene velikosti tipov.

```
RazrednaZbirka{
        u4              magično_število;
        u2              mala_različica;
        u2              velika_različica;
        u2              število_nabora_konstant;
        nk_info          nabor_konstant[število_nabora_konstant−1];
        u2              zastavice_dostopnosti;
        u2              ta_razred;
        u2              nadrazred;
        u2              število_vmesnikov;
        u2              vmesniki[število_vmesnikov];
        u2              število_atributov;
        atri_info        atributi[število_atributov];
        u2              število_metod;
        meto_info          metode[število_metod];
        u2              število_lastnosti;
        last_info        lastnosti[število_lastnosti];
}
```

Koda 2.4: Struktura razredne zbirke

Predmeti iz strukture razredne zbirke 2.4.2:

magično_število

Magično število pomaga identificirati razredno zbirko. Za razredno zbirko mora magično število nositi vrednost `0xCAFEBABE`.

mala_različica in velika_različica

Vrednosti iz `mala_različica` in `velika_različica` določata različico razredne zbirke.

število_nabora_konstant

Vrednost predstavlja število vnosov v naboru konstant (constant pool).

nabor_konstant[ ]

Nabor konstant predstavlja tabelo različnih vrednosti kot so: konstante nizov, opise razredov, opise vmesnikov, opise metod, opise atributov in

še vse druge konstante, na katere se sklicuje razred/vmesnik iz razredne zbirke.

zastavice_dostopnosti

Vrednost predstavlja masko zastavic, ki določajo pravice dostopa do razreda ali vmesnika, ki ga predstavlja ta razredna zbirka.

ta_razred

Vrednost tega polja je indeks v naboru konstant, kjer se nahaja ime razreda, ki ga predstavlja razredna zbirka.

nadrazred

Vrednost predstavlja indeks v naboru konstant, kjer se nahaja ime razreda, iz katerega je ta razred razširjen. Vrednost 0 pomeni, da razred ni razširjen iz drugega razreda.

število_vmesnikov

Vrednost pove število vnosov v polju vmesnikov.

vmesniki[ ]

Vrednost predstavlja polje vmesnikov, ki jih ta razred razširja. Vsaka vrednost polja nosi indeks v naboru konstant, na katerem mestu je ime oz. predstavitev razširjenega vmesnika.

število_atributov

Vrednost pove število vnosov v polju atributov.

atributi[ ]

Tabela, ki predstavlja vse atribute (spremenljivke) razreda. Vsaka vrednost v tabeli predstavlja atribut (ime in tip) razreda.

število_metod

Vrednost pove število vnosov v polju metod.

metode[ ]

Vsak vnos v tabeli `metode`, predstavlja eno metodo.

število_lastnosti

Vrednost pove število vnosov v polju lastnosti.

lastnosti[ ]

Polje vseh lastnosti razreda.

2.4.3 Od javanskega razreda do objekta

Pogledali si bomo postopek oziroma zaporedje korakov za pripravo razreda ali vmesnika na uporabo. Cel postopek je potrebno pred uporabo izvesti nad vsakim razredom ali vmesnikom.

Prevajanje razreda (angl. Compiling)

Ko programer napiše izvorno kodo razreda, jo je potrebno z javanskim prevajalnikom prevesti v format razredne zbirke. Vsak razred v izvorni kodi se prevede v ločeno razredno zbirko. Kako izvedemo prevajanje, je opisano v poglavju 2.3. Ko enkrat dobimo vse potrebne razredne zbirke, je postopek prevajanja končan.

Nalaganje razreda (angl. Class loading)

Naslednji korak, prvi v postopku zagona programa, je nalaganje razreda z razrednim nalagalnikom. Razredni nalagalnik začne s prebiranjem in z razčlenjevanjem izbrane razredne zbirke, iz katere ustvari notranjo predstavitev javanskega razreda. Notranja predstavitev javanskega razreda je lahko poljubne oblike, lahko je tudi ista kot predstavitev v formatu razredne zbirke. Pomembno je le, da jedro javanskega navideznega stroja lahko dostopa do naloženih razredov in prepozna vse njihove lastnosti.

Povezovanje (angl. Linking)

Povezovanje razredov je postopek, ki ga je potrebno izvesti pri razreševanju razredov, ko sta dva razreda v sorodu ali ko se prvi razred sklicuje na drugega. Dva razreda sta v sorodu, ko prvi razred razširja

drugi razred ali pa ko razred implementira vsaj enega od vmesnikov. Povezovanje se lahko izvaja samo med že naloženimi razredi.

Inicializacija (angl. Initializing)

Inicializacija se izvede ob prvi uporabi razreda, kot je dostop do statičnih atributov razreda ali ustvarjanju novih primerkov razreda. Ob inicializaciji se izvede ustrezna inicializacijska metoda.

V vseh opisanih korakih in v samem izvajanju uporabniške kode lahko ob sklicevanju na določen razred ali vmesnik pride do še enega postopka, povezanega z razredi in vmesniki. Temu postopku rečemo razreševanje.

Razreševanje (angl. Resloving)

Postopek razreševanja se izvede ob sklicu na razred ali vmesnik, ki še ni bil naložen. V postopku razreševanja javanski navidezni stroj poskuša v classpath-u poiskati razredno zbirko, ki ustreza sklicanemu/iskanemu razredu ali vmesniku. Nalogo razreševanja opravlja razredni nalagalnik, iz katerega je bil naložen razred, ki se sklicuje na iskan razred oziroma vmesnik. Ko razredni nalagalnik najde ustrezno razredno zbirko, sproži nalaganje in vse nadaljnje potrebne korake. Če se zgodi, da razreda/vmesnika ni mogoče razrešiti (razredne zbirke razreda/vmesnika ni mogoče najti, v enem od postopkov je prišlo do napake), se v toku programa vrže ustrezno izjemo.

Poznamo dva načina glede na čas razreševanja. Prvi način je, da se razreševanje izvede ob trenutku sklica na razred. Drugi način pa je, da se vse razreševanje izvede takoj ob inicializaciji javanskega navideznega stroja. Če v katerem koli od načinov razreševanja pride do napake, se izjema vrže nič prej kot ob trenutku sklica na nerazrešljiv razred oziroma vmesnik.

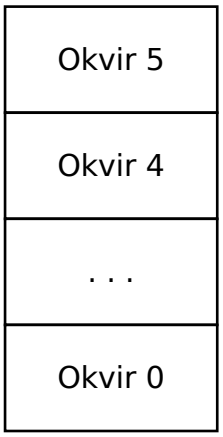
2.4.4 Sklad JVM

Javanski navidezni stroj za svoje delovanje ne uporablja notranjih registrov, vendar za izvajanje operacij uporablja le preprost sklad. Zato rečemo, da je javanski navidezni stroj skladovni navidezni stroj. Sklad se razdeli na okvire

(frames). Vsak od okvirov se nanaša na klic metode in je tudi ustvarjen ob novem klicu metode.

Ob normalnem končanju metode se zadnji okvir pobriše, rezultat metode se zapiše v prejšnji okvir. Prejšnji okvir vedno pripada metodi, ki je klicala trenutno metodo.

Če se v izvajanju trenutne metode vrže izjemo in je trenutna metoda ne ujame, se metoda konča ter pobriše se zadnji okvir. Ob takem koncu metode se nikoli ne vrača rezultat, vendar se izjema posreduje prejšnji metodi.

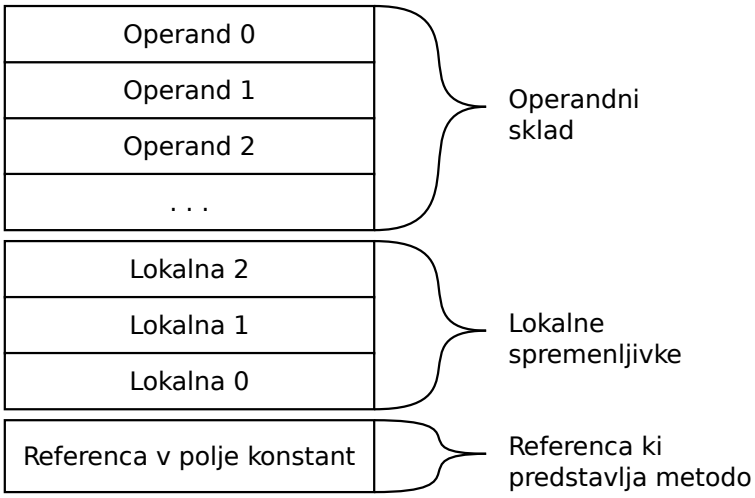


Slika 2.4: Zgradba JVM sklada

Slika 2.4 nam prikazuje sklad programa v trenutku, ko je klicana peta gnezdena metoda. Peti okvir pripada zadnji klicani metodi in se bo pobrisal, ko se bo zadnja metoda končala.

Okvir (Frame) sklada JVM

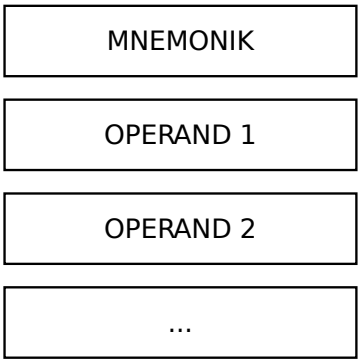
Okvir se uporablja za shranjevanje lokalnih podatkov, delnih rezultatov, vračanje rezultatov, upravljanje z izjemami, skratka za upravljanje z vsemi lokalno vezanimi lastnostmi. Okvir je sestavljen iz operandnega sklada, lokalnih spremenljivk in reference v naboru konstant, ki predstavlja klicano metodo. Struktura okvira je prikazana na sliki 2.5.



Slika 2.5: Okvir sklada JVM

2.4.5 Ukazi javanske vmesne kode

Javanski vmesni ukazi so ukazi, ki jih razume javanski navidezni stroj in jih zna pravilno interpretirati. Sestavljeni so iz zaporedja 8-bitnih bajtov. Struktura je prikazana na sliki 2.6. Prvi bajt je vedno ukaz oz. mnemonik. Sledeči bajti so ukazni operandi. Ker se vsak ukaz izvaja nad skladom javanskega navideznega stroja, lahko uporablja tudi operande s sklada (na primer: prva dva bajta s sklada). Koliko ukaznih ali skladovnih operandov ima ukaz, je odvisno od tega, za kateri ukaz gre.



Slika 2.6: Struktura javanskega vmesnega ukaza

Poglejmo si nekaj primerov javanskih strojnih ukazov in korake pri izvajanju:

iadd (0x60)

Celoštevilčno seštevanje zadnjega in predzadnjega števila s sklada.

Operandi:

2 skladovna operanda

Izvajanje:

1. Potegne zadnje celo število s sklada.
2. Potegne predzadnje število s sklada.
3. Sešteje števili.
4. Rezultat potisne na sklad.

new (0xBB)

Ustvari nov objekt.

Operandi:

2 ukazna operanda

0 skladovnih operandov

Izvajanje:

1. Prebere 2 bajta, ki sledita za mnemonikom (ixbyte1, ixbyte2).
2. Iz prebranih bajtov sestavi indeks: $(ixbyte1 << 18) \mid ixbyte2$.
3. Iz nabora konstant na v prejšnjem koraku izračunanem indeksu prebere predstavitev razreda.
4. Razreši razred.
5. Od upravljalca kopice pridobi dovolj velik prostor, pripravljen za nov objekt.
6. Inicializira objekt ustreznega tipa.
7. Referenco na objekt potisne na sklad.

Poglavje 3

JamVM

V tem poglavju si bomo podrobneje pogledali JamVM, ki je ena od odprtokodnih implementacij javanskega navideznega stroja, avtorja **Roberta Lougherja**. Za JamVM smo se odločili zato, ker je minimalna in polno delujoča implementacija javanskega navideznega stroja. JamVM je napisan v programskem jeziku C in z zelo malo zbirne kode, zato ga je možno prevesti za veliko različnih platform. Zadnjo verzijo in izvorno kodo lahko prenesemo s spletne strani JamVM [3].

3.1 Lastnosti JamVM

Nekaj lastnosti JamVM [3].

- Izvajalni stroj podpira več stopenj optimizacije, od navadnega interpreterja do posebne interpretacije na mestu (hitrost na nivoju JIT prevajalnika) s skladovnim predpomnjenjem.
- Uporablja niti `posix`. Polna implementacija niti (angl. Thread).
- Reference objektov so neposredni kazalci.
- Podpira razredne nalagalnike.

- Izvajalni stroj podpira preklopni interpreter (s programskim preklapljanjem simuliramo različne niti) in nitni interpreter (za vsako javansko nit se utvari tudi dejanska nit v operacijskem sistemu).
- Smetar je implementiran na način označevanja in čiščenja (angl. mark/sweep).
- Pri upravljanju s spanci niti v nitnem interpreterju uporablja signale (ni preverjanja, ali določena nit spi ali ne).
- Podpora zaključevanju objektov v smetarju.
- Polna podpora mehkih referenc.
- Polna podpora smetarja in razlaganja (Unloading).
- Smetar lahko teče sinhrono ali asinhrono v ločeni niti.
- Konstante nizov iz razrednih zbirk so shranjene v razpršilni tabeli.
- Podpira JNI in dinamično nalaganje za uporabo s standardnimi knjižnicami.
- Uporablja svoj domorodni vmesnik za notranje domorodne metode.

3.1.1 Vrste interpretacij

Kot smo že rekli ima JamVM interpreter več načinov in lastnosti interpretiranja.

Direct threaded (Neposredno): Pri neposredni interpretaciji ima vsak ukaz naslov, na katerem se nahaja koda določenega ukaza. Ob interpretaciji določenega ukaza se izvede skok na naslov. Neposredna interpretacija je hitra, vendar pa je težje preklapljanje med različnimi implementacijami ukazov.

Indirect threaded (Posredno): Pri posredni interpretaciji vsak ukaz kaže na polje v iskalni tabeli (angl. lookup table). V iskalni tabeli se za vsak ukaz nahajajo naslovi na sekvenco kode. Pri posredni interpretaciji je dodan še dodaten skok, kar upočasni izvajanje. Dobra lastnost pri posredni interpretaciji je, da lahko preklapljamo med različnimi iskalnimi tabelami. Zaradi tega nam za različne implementacije ukazov ni potrebno vedno znova prepisovati izvorne kode in posledično izvajati prevajanja.

Inlined instructions (Prevedeni ukazi): Pri interpretaciji lahko uporabimo prevedene ukaze, kar omogoča, da se vsak ukaz ali sekvenca javanskih ukazov prevede in sestavi v sekvenco domorodnih ukazov. S tem pohitrimo izvajanje, saj ga ne prekinjamo z zamudnimi skoki. Prevajanje ukazov lahko pripeljemo do takega nivoja, da prevedemo celo metodo; takrat interpreter dobi pomen prevajalnika JIT.

3.2 Uporaba JamVM

V korakih bomo prikazali prenos izvorne kode, prevajanje izvorne kode in uporabo oziroma izvajanje JamVM. Za demonstracijo bomo uporabili sledeči sistem:

Okolje Oracle VirtualBox

Operacijski sistem Manjaro Linux x64 (Arch based Linux distribution)

Lupina Bash

3.2.1 Prenos izvorne kode

Za prenos zadnje verzije izvorne kode iz repozitorija potrebujemo orodje git.

- Prenos in namestitev orodja git:

```
1 pacman -S git
```

- Da ne bi prišlo do zmešnjave, ustvarimo strukturo direktorijev za prevajanje in namestitev.

```
mkdir JamVM && cd JamVM && mkdir installdir
```

- Prenos izvorne kode JamVM iz repozitorija s pomočjo orodja git.

```
git clone git://git.code.sf.net/p/jamvm/code jamvm-code
```

3.2.2 Namestitev odvisnosti in orodij za prevajanje JamVM

Za prevajanje izvorne kode JamVM so potrebna sledeča orodja: gcc, gcc-gcj, make, autoreconf, linux-headers. JamVM ima še odvisnost "GNU classpath", ki je odprtokoden skupek domorodnih knjižnic jave.

- Nalaganje paketa classpath.

```
pacman -S classpath
```

- Večino za prevajanje potrebnih orodij lahko najdemo v skupini paketov "base-devel".

```
sudo pacman -S base-devel
```

Orodij gcc-gcj in gcc-gcj-ejc ni na voljo v uradnih repozitorijih manjaro, zato ju lahko prevedemo sami ali pa namestimo s pomočjo repozitorija AUR (Arch user repository). Za orodji gcc-gcj in gcc-gcj-ejc je postopek namestitve iz AUR enak, razlika je le v imenih datotek.

- Prenos namestitvene datoteke iz AUR za gcc-gcj-ejc.

```
wget https://aur.archlinux.org/packages/gc/gcc-gcj-ecj/gcc-gcj-ecj.tar.gz
```

  in za gcc-gcj

```
wget https://aur.archlinux.org/packages/gc/gcc-gcj/gcc-gcj.tar.gz
```

- Razširimo preneseni arhiv.

```
tar xzf gcc-gcj
```

- Pomaknemo se v razširjen direktorij in poženemo orodje za ustvarjanje programskega paketa. Stikalo `-s` naroča, naj se namestijo vse odvisnosti paketa.

```
makepkg -s
```

- Ob koncu pakiranja programa z orodjem makepkg se v delovnem direktoriju ustvari paket. Ta paket lahko namestimo s pomočjo orodja pacman. Stikalo `-U` naroča postopek posodabljanja iz datoteke.

```
pacman -U gcc-gcj.pkg.tar.xz
```

3.2.3 Prevajanje

Za prevajanje izvorne kode JamVM so potrebna orodja gcc, make tool, linux-headers in skupek domorodnih knjižnic "classpath".

- Pomik v podimenik jamvm-code, kjer se nahaja izvorna koda.

```
cd jamvm-code
```

- Poženemo orodje autoreconf, ki ustvari vse potrebne konfiguracijske datoteke. Stikalo `--install` pove, da želimo prevajati za namestitev.

```
autoreconf --force --install
```

- Poženemo generirano skripto za konfiguracijo in generiranje make datotek. V prvem parametru podamo pot do knjižnic "classpath". V drugem parametru, `--prefix`, podamo pot do direktorija, v katerega se bo orodje jamvm namestilo. Ker mora biti pot do namestitvenega direktorija absolutna, si pomagamo z ukazom readlink.

```
./configure --with-classpath-install-dir=/usr --prefix=$( readlink -m ../
    installdir)
```

- Če med konfiguracijo ni prišlo do napak, je programska koda pripravljena za prevajanje. Prevajanje izvedemo z orodjem make. Ukaz make install namesti prevedeno kodo.

```
make && make install
```

- Ob koncu prevajanja in namestitve lahko izvajalno datoteko JamVM najdemo na lokaciji: `./installdir/bin/jamvm`.

3.2.4 Izvajanje

Način izvajanja javanskih programov na navideznem stroju JamVM je popolnoma enak postopku izvajanja javanskih programov na katerem drugem javanskem navideznem stroju. Glede na prejšnjo trditev si je postopek izvajanja mogoče ogledati v poglavju 2.3 (`Primer uporabe jave`) na strani 5, le da namesto ukaza java uporabimo `./installdir/bin/jamvm`.

3.3 Datotečna struktura izvorne kode

Tukaj si bomo ogledali grobo strukturo izvorne kode JamVM. Pokrili ne bomo vseh datotek, vendar le najpomembnejše za dosego našega cilja.

- Direktorij z izvorno kodo.

```
./src
```

- Direktorij, ki vsebuje vse sistemsko odvisne dele programa.

```
./src/arch
```

- Del programske opreme, ki na kopici pridobiva prostor za primerke novih razredov.

```
./src/alloc.c
./src/alloc.h
```

- Glavna datoteka glav jezika C.

```
./src/jam.h
```

- Direktorij, ki vsebuje vmesnike za različne implementacije javanskega izvajalnega okolja. JamVM trenutno polno podpira le GNU classpath. Podpora za OpenJDK je še v povojih. Tukaj se nahajajo tudi nekateri prepisani razredi JRE, ki za delovanje skupaj z JamVM potrebujejo nekoliko drugačno implementacijo.

```
./src/classlib/
./src/classlib/gnuclasspath/ # GNU Classpath
./src/classlib/openjdk # OpenJDK
```

- Direktorij, ki vsebuje implementacijo interpreterja oziroma izvajalnega stroja.

```
./src/interp
```

- Datoteki, ki vsebujeta programsko kodo za klice domorodnih metod.

```
./src/natives.h
./src/natives.c
```

Poglavje 4

Dodelava JamVM

V zadnjem vsebinskem poglavju si bomo pogledali cilj, ki smo si ga zastavili, in kako smo ga s pomočjo znanja iz prejšnjih poglavij realizirali.

4.1 Od ideje do ideje

Ideja se je pojavila ob zanimanju po statistiki izvedenih vmesnih ukazov na navideznem stroju. Prvotna ideja je bila implementirati navidezni stroj tako, da bi po koncu izvajanja na standardni izhod izpisal statistiko izvedenih vmesnih ukazov. Ker bi tak način zelo otežil kasnejšo obdelavo statističnih podatkov, smo začeli razmišljati, da bi statistične podatke vrnili v definirani strukturi. Tako bi uporabnik podatke prejel v že znani strukturi in jih zato lažje obdelal. Na koncu smo se odločili, da podatke vrnemo uporabniku kar v programski strukturi znotraj izvajanja javanskega navideznega stroja. Končna ideja nas je potegnila v implementacijo javanske knjižnice, ki bo v ozadju komunicirala s samim jedrom javanskega navideznega stroja in vračala statistične podatke izvajanja.

4.2 Javanska knjižnica VMEP

Da bi dosegli naš cilj in olajšali uporabo naše rešitve, smo se odločili za implementacijo javanske knjižnice. Knjižnico smo poimenovali Vstopna Točka Navideznega Stroja/Virtual Machine Entry Point (VMEP). Ker želimo s knjižnico pridobivati podatke izven javanskega izvajalnega okolja, in sicer iz sistemsko odvisnega javanskega navideznega stroja (v našem primeru JamVM), potrebujemo domorodne metode.

Domorodne metode: Domorodne metode so javanske metode, ki jih definiramo v javi, implementiramo pa v sistemsko odvisnem programskem jeziku. Uporabljajo se za premostitev javanskega izoliranega izvajalnega okolja s sistemom, na katerem teče javanski navidezni stroj. Ko želimo implementirati javansko knjižnico z domorodnimi metodami, običajno uporabimo Javanski domorodni vmesnik/Java native interface (JNI). V našem primeru potrebujemo premostitev med javanskim izvajalnim okoljem in javanskim navideznim strojem, zato lahko samo knjižnico in domorodne metode implementiramo kar znotraj javanskega navideznega stroja.

V drugi izdaji knjige B. Vennersa, "Inside the Java Virtual Machine", najdemo vodiče in primere uporabe domorodnih metod z vmesnikom JNI [4].

4.2.1 Struktura in opis knjižnice VMEP

Vsi gradniki knjižnice VMEP se nahajajo v javanskem paketu `jamvm.vmep`.

Monitor

Abstraktni razred, ki predstavlja orodje, s katerim začnemo in končamo opazovanje izvedenih javanskih vmesnih ukazov.

`public void start()` Metoda, ki začne opazovanje.

`public void stop()` Metoda, ki ustavi opazovanje.

`public boolean isRunning()` Preveri, ali poteka opazovanje.

`protected abstract void onExecute()` Metoda je klicana ob vsaki izvedbi javanskega vmesnega ukaza znotraj opazovanja. Metoda še ni povsem uporabna. Za večjo uporabnost manjka še dodelava, ki bo omogočala pridobitev različnih informacij o trenutnem stanju jedra JVM (kateri ukaz se izvaja in katere parametre ima, katera metoda se izvaja, vsebina javanski sklada ...).

`public void addRuntimeFilter (RuntimeFilter filter)` Metoda doda filter, ki se uporabi ob izvajanju opazovanja. Pravilna uporaba filtra lahko zelo pohitri izvajanje programa med opazovanjem, saj lahko z njim preskočimo opazovanja metod, katerih rezultati nas ne zanimajo.

`public void removeRuntimeFilter (RuntimeFilter filter)` Metoda odstrani določen filter.

InstructionMonitor

Razširjen iz razreda `Monitor`, namenjen opazovanju javanskih vmesnih ukazov.

`public int[] getCounts()` Metoda vrne tabelo statističnih podatkov vseh prejšnjih opazovanj. Pod vsakim indeksom tabele, ki predstavlja številko ukaza, najdemo celoštevilsko vrednost, ki pove, kolikokrat med opazovanjem je bil ukaz izveden.

`public int[] getArrayCount(Method m)` Metoda deluje enako kot `getCounts()`, le da vrne samo podatke o izvajanju v podani metodi.

`public int[] getArrayCount(Package p)` Metoda deluje enako kot `getCounts()`, le da vrne samo podatke o izvajanju v vseh metodah iz podanega javanskega paketa.

MemoryMonitor

Razširjen iz razreda `Monitor`, namenjen opazovanju porabe pomnilnika.

`int getMemUse()` Metoda vrne vsoto velikosti vseh pridobitev pomnilnika.

`int getMemUseFor(Method m)` Metoda vrne vsoto velikosti vseh pridobitev pomnilnika, ki so se zgodile znotraj podane metode.

`int getMemUseFor(Package m)` Metoda vrne vsoto velikosti vseh pridobitev pomnilnika, ki so se zgodile znotraj vseh metod iz podanega javanskega paketa.

RuntimeFilter

RuntimeFilter je abstraktni razred, ki je namenjen implementaciji filtra za monitor. Če filter s pomočjo metode `addRuntimeFilter` dodamo v `Monitor`, se opazovanje začne filtrirati. S tem lahko veliko pridobimo na hitrosti izvajanja, saj s filtrom omejimo opazovanje le na tisti del programa, ki nas zanima.

Opcode

Razred `Opcode` vsebuje konstante ki nosijo vrednosti, katere predstavljajo javanske vmesne ukaze.

`String getNameFor(int opcode)` Metoda spremeni vrednost, ki predstavlja javanski vmesni ukaz, v zbirno ime javanskega vmesnega ukaza.

4.2.2 Kratek primer uporabe knjižnice VMEP

V izvorni kodi 4.1 je v javi napisan program, ki prikazuje primer uporabe knjižnice VMEP.

V prvem delu `main` metode naredimo nov primerek razreda `InstructionMonitor`, s pomočjo katerega bomo izvajali opazovanje.

V drugem delu s pomočjo metode `start` zaženemo opazovanje. Opazovanje poteka skozi celotno izvajanje gnezdenih zank `for`, do klica metode `stop`.

V tretjem delu pridobimo rezultate s pomočjo metode `getCounts` in jih izpišemo na zaslon. Pri izpisu si pomagamo še s statično metodo `getNameFor` razreda `Opcode` za pretvorbo številke ukaza v ime ukaza. Tako je izpis (Koda 4.3) bolj prijazen in tudi lažje ga je interpretirati.

```
1  import jamvm.vmep.InstructionMonitor;
2  import jamvm.vmep.Opcode;
3
4  public class VmepTest{
5    public static void main(String[] args){
6      InstructionMonitor m=new InstructionMonitor();
7
8      m.start(); // Zacetek opazovanja
9
10     int result=0;
11
12     for(int i=0;i<100;i++){
13       for(int j=0;j<100;j++){
14         result+=i*j;
15       }
16     }
17
18     m.stop(); // Konec opazovanja
19
20     // Izpis rezultatov na zaslon
21     int[] instFreq=m.getCounts();
22     for(int i=0;i<instFreq.length;i++){
23       if(instFreq[i]>0){
24         System.out.println(Opcode.getNameFor(i) + " - "+instFreq[i]);
25       }
26     }
27   }
28 }
```

Koda 4.1: Kratek primer uporabe VMEP knjižnice

Preden lahko izvorno kodo 4.1 poženemo, jo je potrebno prevesti v javansko razredno zbirko. Prevajanje izvedemo z enim od orodij za prevajanje javanske izvorne kode. V našem primeru smo uporabili orodje javac iz skupka orodij OpenJDK. Pri prevajanju programa prevajalnik potrebuje razredne zbirke vseh referenc. V našem primeru potrebuje tudi ustrezen skupek razrednih zbirk za knjižnico VMEP. Arhiv razrednih zbirk javanskega navideznega stroja JamVM, ki vsebuje tudi vse razredne zbirke knjižnice VMEP, najdemo med binarnimi datotekami javanskega navideznega stroja na lokaciji `share/jamvm/classes.zip`.

```
1 javac -classpath share/jamvm/classes.zip VmepTest.java
2 jamvm -classpath VmepTest
```

Koda 4.2: Prevajanje in zagon kratkega primera

```
1 ICONST_0 - 102
2 BIPUSH - 10201
3 ILOAD - 20100
4 ILOAD_2 - 10000
5 ILOAD_3 - 10101
6 ALOAD_0 - 1
7 ALOAD_1 - 1
8 ISTORE - 100
9 ISTORE_2 - 10001
10 ISTORE_3 - 1
11 IADD - 10000
12 IMUL - 10000
13 IINC - 10100
14 IF_ICMPGE - 10201
15 GOTO - 10100
16 RETURN - 1
17 INVOKEVIRTUAL - 1
18 INVOKESPECIAL - 1
```

Koda 4.3: Izpis pognane kode 4.1.

4.2.3 Definicija in implementacija javanskega dela knjižnice VMEP

Kot smo že omenili, za povezavo javanskega dela in sistemskega dela naše razširitve potrebujemo javanske domorodne metode. Domorodne metode javanskih knjižnic običajno definiramo s pomočjo vmesnika JNI. Zaradi odvisnosti razširitve od implementacije JamVM smo lahko v našem primeru domorodne metode definirali kar v implementaciji samega navideznega stroja JamVM.

Definicije domorodnih metod za uporabo skupaj z gnuclasspath se nahajajo v datoteki `src/classlib/gnuclasspath/natives.c`. V to datoteko smo dodali še definicije naših javanskih razredov in domorodnih metod. Pri vsaki od domorodnih metod je potrebno podati še referenco na sistemsko metodo, ki bo implementacija in sistemski del javanske domorodne metode.

```
1  /* VMEP Monitor */
2  VMMethod vmep_monitor[]={
3   {"mstart",  NULL,vmepMonitorStart},
4   {"mstop", NULL,vmepMonitorStop},
5   {"initMonitors",NULL,vmepMonitorInit},
6   {NULL,NULL,NULL}
7  };
8
9  VMClass native_methods[] = {
10    {"java/lang/VMClass", vm_class},
11    ...
12    {"jamvm/vmep/Monitor", vmep_monitor}, // Definicija razreda Monitor, knjižnice
        VMEP.
13    ...
14    {NULL, NULL}
15 };
```

Koda 4.4: Definicija domorodnih metod VMEP knjižnice v `natives.c`.

V kodi 4.4 lahko opazimo da, se pri vsaki definiciji domorodne javanske metode poda še referenca oziroma ime funkcije, ki predstavlja sistemski del javanske domorodne metode. V knjižnici VMEP imamo 3 domorodne

metode: “mstart”,“mstop” in “initMonitors”. Vsaka od njih ima implementirano sistemsko funkcijo v jedru razširitve VMEP, ki se nahaja v datoteki `vmep.c`.

4.3 Poseg v JamVM izvorno kodo

V prejšnjih poglavjih je bila z uporabniške strani predstavljena knjižnica VMEP. Za realizacijo naše razširitve potrebujemo poseg v izvorno kodo interpreterja javanskega navideznega stroja (JamVM). JamVM vsebuje več vrst interpretacij, ki jih lahko po izbiri vključimo ali izključimo. Zaradi tega se pojavi težava, saj moramo, če želimo podpreti vse vrste interpretacije, tudi našo predelavo prilagoditi za vse vrste interpretacij. Druga možnost, za katero smo se tudi odločili, je, da podpremo le del vseh vrst interpretacij in to uporabnikom omejimo ali sporočimo v dokumentaciji.

Izvorno kodo interpreterja najdemo v mapi `src/interp/engine` . V njej se nahaja več datotek s končnico `.h` in še datoteka `engine.c`, ki implementira skoraj vse javanske vmesne ukaze. Ukazi so implementirani s pomočjo predprocesorskih ukazov programskega jezika C.

Naš glavni poseg v izvorno kodo se nahaja v datoteki `interp-inlining.h`, ki vsebuje večino predprocesorske kode za prevajanje ukazov (inlining). V njej najdemo definicijo predprocesorske spremenljivke `DEF_OPC_LBLS` , ki zgradi oznako pred vsakim ukazom. Poseg je prikazan v kodi 4.5. Vanjo smo dopisali predprocesorska ukaza, ki vcepita potrebne funkcije za delovanje razširitve. Zaradi tega se ob izvedbi vsakega javanskega ukaza tudi pokličejo funkcije, ki smo jih s pomočjo predprocesorskih ukazov `VMEP_EXEPRE()` in `VMEP_EXEMEM()` vcepili v interpreter.

```
1 ...
2 #define DEF_OPC_LBLS(opcode, level, PRE, BODY)  \
3  label(opcode, level, START)                    \
4  PAD                                \
5  label(opcode, level, ENTRY)                    \
6  PRE                                \
```

```
7  GUARD(opcode, level)                    \
8  VMEP_EXEPRE()       \
9  BODY                                    \
10 VMEP_EXEMEM()       \
11 label(opcode, level, END)
12 . . .
```

Koda 4.5: Vcepitev funkcije v interpreter, v datoteki `interp-inlining.h`

Definiciji pred-procesorskih ukazov `VMEP_EXEPRE()` in `VMEP_EXEMEM()` se nahajata v novo ustvarjeni datoteki `src/vmep.h` (Koda 4.6). V definiciji teh dveh pred-procesorskih ukazov lahko vidimo klice funkcij, ki izvajajo potrebne funkcionalnosti razširitve.

```
1  #define VMEP_EXEPRE()                    \
2  {                                        \
3   Instruction *inst=(Instruction*)pc;     \
4   vmepExePre(getExecEnv(),inst,frame,mb); \
5  }
6
7  #define VMEP_EXEMEM()                              \
8  {                                                  \
9   Instruction *inst=(Instruction*)pc;               \
10  vmepExeMem(getExecEnv(),inst,frame,mb,ostack); \
11 }
```

Koda 4.6: Definicija pred-procesorskih ukazov, v datoteki `vmep.h`

Implementacija funkcij in vse ostale funkcionalnosti razširitve se nahajajo v datoteki `vmep.c`.

4.3.1 Opis jedra in delovanja razširitve VMEP

Vsa ostala implementacija, ki je tudi večinski del razširitve, se nahaja v jedru. Za lažje vzdrževanje in nadgradnjo razširitve ali navideznega stroja smo želeli čim večji del izvorne kode razširitve ločiti od izvorne kode navideznega stroja JamVM. Zato smo rešitev v večini implementirali v jedru, ki se nahaja v dveh ločenih datotekah, `vmep.c` in `vmep.h`. Tako je presek izvorne kode JamVM

in razširitve VMEP minimalen.

V poglavju 4.2.1, kjer smo predstavili uporabo knjižnice VMEP, smo spoznali razred `Monitor`. S pomočjo primerkov razreda `Monitor` lahko v kateremkoli trenutku izvajanja programa začnemo ali končamo opazovanje.

Postopek izvajanja opazovanja

Začetek opazovanja

Ob klicu metode `start()` nad objektom tipa `Monitor` se referenca primerka objekta shrani v poseben povezan seznam v jedru.

Pred vsako izvedbo javanskega vmesnega ukaza

Preverjanje stanja jedra VMEP

Če med obdelavo enega od javanskih vmesnih ukazov naletimo na izvajanje javanske vmesne kode, je ne obdelamo. V takem primeru gre običajno za izvajanje filtra. Torej, če ugotovimo, da je jedro VMEP že v stanju obdelave, obdelavo trenutnega javanskega vmesnega ukaza prekinemo oziroma preskočimo.

Preverjanje seznama opazovanja

Pred izvedbo vsakega javanskega vmesnega ukaza se preveri, ali je seznam opazovanja prazen. Če seznam ni prazen, pomeni, da je eden od monitorjev v stanju opazovanja. Torej tisti monitorji, katerih reference so shranjene v seznamu opazovanja, so v aktivnem stanju opazovanja.

Preverjanje filtrov

Za vsak monitor se izvede postopek preverjanja filtra. V tem postopku se kliče metoda `doFilter` nad primerki filtrov, ki so bili dodani monitorju. Če kateri od filtrov vrne status `FILTER_BLOCK` (blokiraj) ali `FILTER_RECURSIVE_LOCK` (rekurzivno blokiraj), se obdelava prekine.

Obdelava javanskega vmesnega ukaza

Odvisno od tipa monitorja se izvede potrebna procedura. V pri-

meru, da gre za monitor, ki šteje ukaze, se v tabeli, ki je del monitorja, na ustreznem mestu števec poveča za ena.

Izvedba javanskega vmesnega ukaza

Tok programa prevzame JamVM in izvede javanski vmesni ukaz.

Po izvedbi javanskega vmesnega ukaza

Preverimo, ali je izvedeni ukaz pridobil prostor v kopici. Če je, lahko zdaj pridobimo velikost pridobljenega prostora. To velikost prištejemo v monitor, če je ta tipa za merjenje pridobljenega prostora.

Konec opazovanja

Ko želimo končati opazovanje, pokličemo metodo `stop()` in takrat se referenca objekta monitorja odstrani s seznama opazovanja.

4.4 Primer uporabe razširitve VMEP

S pomočjo razširitve VMEP bomo na primeru primerjali navadno hitro urejanje s hitrim urejanjem na mestu. Hitro urejanje je algoritem za urejanje števil ali veličin. Velikokrat se ga uporablja v praksi, saj se v kombinaciji z drugimi algoritmi zelo dobro odreže. Algoritem je rekurziven in njegova zgornja meja časovne zahtevnosti je prikazana v formuli 4.1.

$$O(n * log(n)) \tag{4.1}$$

4.4.1 Navadno hitro urejanje

Navadno hitro urejanje je sestavljeno iz štirih delov: izstopnega pogoja, izbire delilnega elementa (pivota), porazdelitve glede na delilni element in rekurzivnih klicev. Implementacija navadnega hitrega urejanja v javi je prikazana v sledeči kodi 4.7.

```
1 static List<Integer> quickSort(List<Integer> array){
2  // Izstopni pogoj
3  if(array.size()<=1) return array;
```

```
4
5   // Ustvarjanje seznamov za porazdeljevanje
6   ArrayList<Integer> less=new ArrayList<Integer>();
7   ArrayList<Integer> equal=new ArrayList<Integer>();
8   ArrayList<Integer> great=new ArrayList<Integer>();
9
10  // Izbira delilnega elementa (pivota)
11  int pivot=array.get(array.size()/2);
12
13  // Porazdeljevanje glede na delilni element
14  for(Integer v:array){
15    if(v<pivot) less.add(v);
16    if(v==pivot) equal.add(v);
17    if(v>pivot) great.add(v);
18  }
19
20  // Rekurzivna klica nad seznami manjših in večjih števil
21  less=(ArrayList<Integer>)quickSort(less);
22  great=(ArrayList<Integer>)quickSort(great);
23
24  // Sestavljanje končnega seznama
25  array=new ArrayList<Integer>();
26  array.addAll(less);
27  array.addAll(equal);
28  array.addAll(great);
29
30  return array;
31 }
```

Koda 4.7: Navadno hitro urejanje v programskem jeziku Java

4.4.2 Hitro urejanje na mestu

Pri hitrem urejanju na mestu se porazdeljevanje izvede na mestu v enem seznamu (tabeli). S tem se znebimo treh dodatnih seznamov, prepisovanja med seznami in na koncu združevanja v en seznam. V javi implementirano hitro urejanje na mestu je prikazano v kodi 4.8.

```
1  static Random rand=new Random();
2  static void quickSortI(int[] array,int first,int last){
3    // Izstopni pogoj
4    if(last−first<=1) return;
5
6    // Deklaracija in inicializacija spremenljivk
7    int left,right,pivot,temp=0;
8    left=first;
9    right=last;
10
11   // Naključno izbiranje delilnega elementa (pivot)
12   pivot=array[rand.nextInt(array.length−1)];
13
14   // Zanka porazdeljevanja na mestu
15   while(left<=right){
16     while(array[left]<pivot) left++;
17     while(array[right]>pivot) right−−;
18     if(left<=right){
19       temp=array[left];
20       array[left]=array[right];
21       array[right]=temp;
22       left++;
23       right−−;
24     }
25   }
26
27   // Rekurzivni klici
28   quickSortI(array,first,right);
29   quickSortI(array,left,last);
30 }
```

Koda 4.8: Hitro urejanje na mestu v programskem jeziku Java

4.4.3 Primerjava

V praksi se izkaže da kljub isti časovni zahtevnosti (formula 4.1) je hitro urejanje na mestu hitrejše od navadnega hitrega urejanja. S pomočjo razširitve

VMEP bomo izmerili frekvenco ukazov v izvajanju obeh algoritmov pri istem vhodu in iz rezultatov poskušali ugotoviti, zakaj je izvajanje hitrega urejanja na mestu hitrejše. Pripravili smo program (Koda 4.9), ki bo s pomočjo razširitve VMEP za vsak algoritem štel javanske vmesne ukaze.

```
1  public static void main(String[] args) throws Exception{
2   // Napolnimo tabeli z 100 naključnimi števili
3   ArrayList<Integer> list=new ArrayList<Integer>();
4   int[] array=new int[100];
5   int val=0;
6   for(int i=0;i<100;i++){
7     val=rand.nextInt(99);
8     list.add(val);
9     array[i]=val;
10  }
11
12
13  long timeQS,timeQSI;
14
15  // Deklaracija in inicializacija dveh monitorjev za beleženje frekvence ukazov
16  InstructionMonitor monitorQS=new InstructionMonitor();
17  InstructionMonitor monitorQSI=new InstructionMonitor();
18
19
20  // Začetek merjenja časa za navadno hitro urejanje
21  timeQS=System.nanoTime();
22
23  // Začetek opazovanja prvega monitorja za navadno hitro urejanje
24  monitorQS.start();
25
26  // Klic metode za navadno hitro urejanje
27  list=(ArrayList<Integer>)quickSort(list);
28
29  // Konec opazovanja prvega monitorja za navadno hitro urejanje
30  monitorQS.stop();
31
32  // Končamo merjene časa za navadno hitro urejanje
33  timeQS=System.nanoTime()−timeQS;
```

```
34
35
36  // Začetek merjenja časa za hitro urejanje na mestu
37  timeQSI=System.nanoTime();
38
39  // Začetek opazovanja drugega monitorja za hitro urejanje na mestu
40  monitorQSI.start();
41
42  // Klic metode za hitro urejanje na mestu
43  quickSortI(array,0,array.length-1);
44
45  // Konec opazovanja drugega monitorja za hitro urejanje na mestu
46  monitorQSI.stop();
47
48  // Končamo merjene časa za hitro urejanje na mestu
49  timeQSI=System.nanoTime()-timeQSI;
50
51
52  // Izpis rezultatov
53  System.out.println("Instructions frequencies for quick sort:");
54  printResults(monitorQS.getCounts());
55
56  System.out.println();
57  System.out.println();
58
59
60  System.out.println("Instructions frequencies for in place quick sort:");
61  printResults(monitorQSI.getCounts());
62
63  System.out.println();
64  System.out.println();
65
66
67  // Izpis porabljenih časov za oba algoritma
68  System.out.println("Quick sort: "+timeQS);
69  System.out.println("In place quick sort: "+timeQSI);
70
71 }
```

```
72 // Metoda za izpis frekvenc dolocenega monitorja
73 static void printResults(int[] table){
74   for(int i=0;i<table.length;i++){
75     if(table[i]>0){
76       System.out.println(Opcode.getNameFor(i)+" - "+table[i]);
77     }
78   }
79 }
```

Koda 4.9: Programska koda za testiranje algoritmov

Programsko kodo 4.9 za testiranje prevedemo in poženemo z navideznim strojem JamVM, podobno kot je bilo prikazano na testnem primeru 4.2 v poglavju 4.2.2. Kot rezultat lahko na standardnem izhodu pričakujemo frekvence izvedenih javanskih vmesnih ukazov (4.10, 4.11).

Rezultati: Frekvence izvedenih javanskih vmesnih ukazov

```
Instructions frequencies for quick sort:
ICONST_M1 – 192
ICONST_0 – 367
ICONST_1 – 4368
ICONST_2 – 97
BIPUSH – 192
ILOAD – 3448
ALOAD – 4033
ILOAD_0 – 91
ILOAD_1 – 6701
ILOAD_2 – 102
ILOAD_3 – 49
ALOAD_0 – 28915
ALOAD_1 – 1346
ALOAD_2 – 463
ALOAD_3 – 1213
AALOAD – 1376
ISTORE – 336
ASTORE – 712
ISTORE_2 – 49
ASTORE_0 – 48
ASTORE_1 – 97
ASTORE_2 – 48
ASTORE_3 – 295
AASTORE – 1328
POP – 808
DUP – 3328
DUP_X1 – 1992
IADD – 3281
IMUL – 49
IDIV – 48
IINC – 664
IFEQ – 712
IFNE – 2
IFGE – 192
IFLE – 288
IF_ICMPEQ – 1410
IF_ICMPNE – 2656
IF_ICMPLT – 1376
IF_ICMPGE – 2184
IF_ICMPGT – 97
IF_ICMPLE – 1050
GOTO – 2151
IRETURN – 4234
ARETURN – 3001
RETURN – 4293
GETSTATIC – 2
GETFIELD – 4464
PUTFIELD – 5473
INVOKEVIRTUAL – 4422
INVOKESPECIAL – 4197
INVOKESTATIC – 248
INVOKEINTERFACE – 2713
NEW – 384
ANEWARRAY – 241
ARRAYLENGTH – 857
CHECKCAST – 1050
IFNULL – 4
IFNONNULL – 4

```

Koda 4.10: Frekvence izvedenih ukazov za navadno hitro urejanje

```
Instructions frequencies for in place quick sort:
ICONST_0 – 1011
ICONST_1 – 4042
BIPUSH – 2020
LDC2_W – 3030
ILOAD – 57760
ALOAD – 1
ILOAD_1 – 11111
ILOAD_2 – 6061
ILOAD_3 – 16406
ALOAD_0 – 40094
ALOAD_2 – 2
IALOAD – 32695
ISTORE – 3194
ISTORE_2 – 1010
ISTORE_3 – 2020
IASTORE – 328
IADD – 1010
LADD – 1010
ISUB – 6062
LMUL – 1010
IREM – 1010
INEG – 1010
LUSHR – 1010
IAND – 1010
LAND – 1010
IINC – 29399
L2I – 1010
IFLT – 1010
IFGT – 1010
IF_ICMPNE – 1010
IF_ICMPGE – 9752
IF_ICMPGT – 5317
IF_ICMPLE – 21605
GOTO – 29235
IRETURN – 2020
RETURN – 2022
GETSTATIC – 1010
GETFIELD – 2020
PUTFIELD – 1010
INVOKEVIRTUAL – 2021
INVOKESPECIAL – 1
INVOKESTATIC – 2021
ARRAYLENGTH – 1011

```

Koda 4.11: Frekvence izvedenih ukazov za hitro urejanje na mestu

Rezultati, preneseni v tabelo 4.1.

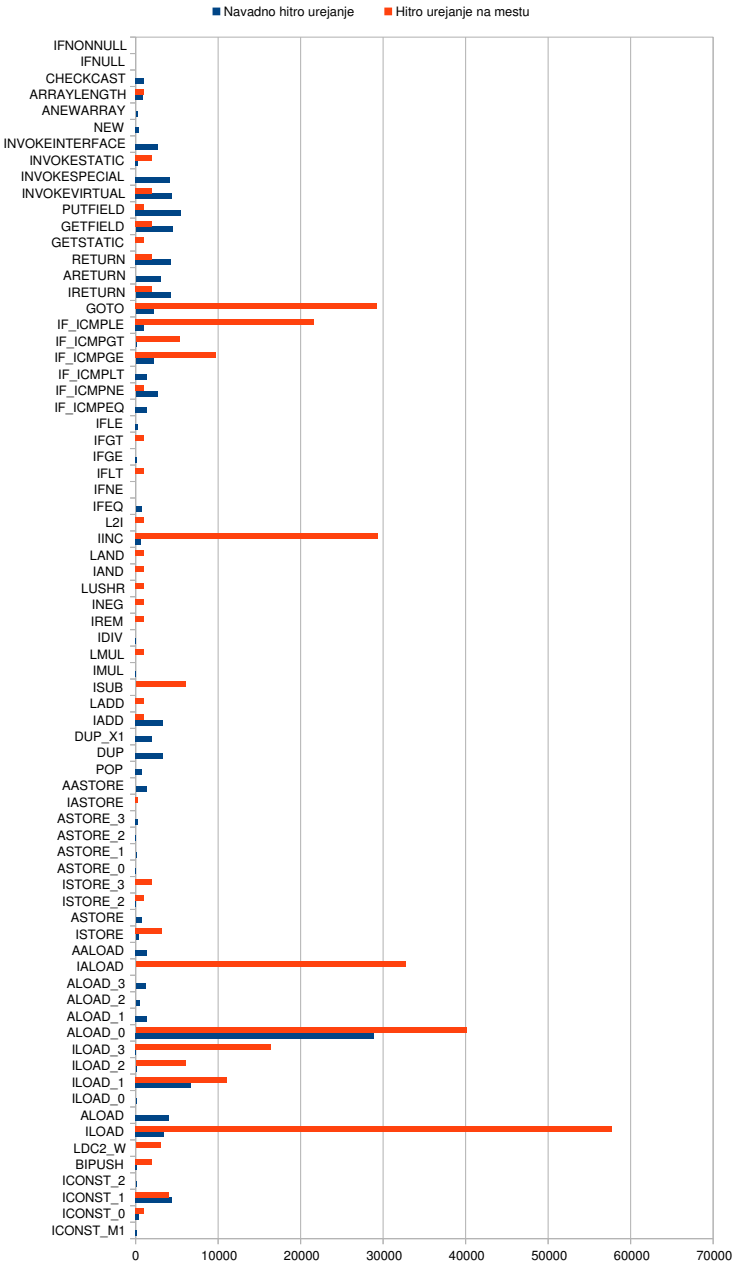
	Navadno hitro urejanje	Hitro urejanje na mestu
ICONST_M1	192	0
ICONST_0	367	1011
ICONST_1	4368	4042
ICONST_2	97	0
BIPUSH	192	2020
LDC2_W	0	3030
ILOAD	3448	57760
ALOAD	4033	1
ILOAD_0	91	0
ILOAD_1	6701	11111
ILOAD_2	102	6061
ILOAD_3	49	16406
ALOAD_0	28915	40094
ALOAD_1	1346	0
ALOAD_2	463	2
ALOAD_3	1213	0
IALOAD	0	32695
AALOAD	1376	0
ISTORE	336	3194
ASTORE	712	0
ISTORE_2	49	1010
ISTORE_3	0	2020
ASTORE_0	48	0
ASTORE_1	97	0
ASTORE_2	48	0
ASTORE_3	295	0
IASTORE	0	328
AASTORE	1328	0
POP	808	0
DUP	3328	0
DUP_X1	1992	0
IADD	3281	1010
LADD	0	1010
ISUB	0	6062
IMUL	49	0
LMUL	0	1010
IDIV	48	0
IREM	0	1010

	Navadno QS	QS na mestu
INEG	0	1010
LUSHR	0	1010
IAND	0	1010
LAND	0	1010
IINC	664	29399
L2I	0	1010
IFEQ	712	0
IFNE	2	0
IFLT	0	1010
IFGE	192	0
IFGT	0	1010
IFLE	288	0
IF_ICMPEQ	1410	0
IF_ICMPNE	2656	1010
IF_ICMPLT	1376	0
IF_ICMPGE	2184	9752
IF_ICMPGT	97	5317
IF_ICMPLE	1050	21605
GOTO	2151	29235
IRETURN	4234	2020
ARETURN	3001	0
RETURN	4293	2022
GETSTATIC	2	1010
GETFIELD	4464	2020
PUTFIELD	5473	1010
INVOKEVIRTUAL	4422	2021
INVOKESPECIAL	4197	1
INVOKESTATIC	248	2021
INVOKEINTERFACE	2713	0
NEW	384	0
ANEWARRAY	241	0
ARRAYLENGTH	857	1011
CHECKCAST	1050	0
IFNULL	4	0
IFNONNULL	4	0
	113741	307411

Tabela 4.1: Primerjave frekvenc po ukazih.

Za lažjo predstavitev smo rezultate prenesli še v stolpčni diagram, ki je

prikazan na sliki 4.1.



Slika 4.1: Frekvence rezultatov v stolpčnem diagramu

	Navadno QS	QS na mestu
ICONST	5024	5053
ILOAD	10391	124033
ALOAD	37346	40097
ISTORE	385	6552
ASTORE	2528	0
STACK_RELATED	6320	5050
I_ARITM	4042	37481
I_LOG	0	2020
L_ARITM	0	2020
L_LOG	0	2020
L2I	0	1010
IF	1202	2020
IF_ICMP	8773	37684
GOTO	2151	29235
RETURN	11528	4042
GET	4466	3030
PUT	5473	1010
INVOKE	11580	4043
NEW	625	0
ARRAYLENGTH	857	1011
CHECKCAST	1050	0
	113741	307411

Tabela 4.2: Primerjava frekvenc izvedenih ukazov po skupinah ukazov

Že na prvi pogled lahko iz diagrama na sliki 4.1 ugotovimo, da navadno hitro urejanje najpogosteje izvede ukaz ALOAD_0 za razliko od hitrega urejanja na mestu, ki najpogosteje izvede ukaz ILOAD.

Ukaz ALOAD_0 prebere referenco iz lokalne spremenljivke 0 na sklad, kar je za navadno hitro urejanje v javi logično, saj implementacija temelji na seznamih, ki pa imajo svoje reference.

Ukaz ILOAD je namenjen nalaganju celega števila iz lokalne spremenljivke na sklad. ILOAD je najpogostejši pri hitrem ujemanju na mestu zato, ker je v implementaciji veliko dostopov do lokalnih spremenljivk tipa int.

Poglejmo si razlog, zakaj je hitro urejanje na mestu kljub več izvedenim ukazom hitrejše. Eden izmed najbolj časovno potratnih ukazov med javanskimi vmesnimi ukazi je ukaz NEW. Ukaz NEW ustvari nov primerek ali objekt določenega razreda. Če primerjamo frekvenci ukaza NEW pri hitrim urejanju in hitrim urejanju na mestu, lahko opazimo, da se ukaz NEW pri

hitrim urejanjem na mestu sploh ne pojavi. Zaradi manjše frekvence ukaza NEW je program veliko hitrejši in to je eden izmed vzrokov, da je hitro urejanje na mestu hitrejše od navadnega hitrega urejanja.

Rezultati: Časovna meritev

Rezultati merjenja časov izvajanja 4.4.3 potrjujejo, da je hitro urejanje na mestu hitrejše od navadnega hitrega urejanja. Hitro urejanje na mestu je bilo v našem primeru za 230890 ms oziroma približno štirikrat hitrejše od navadnega hitrega urejanja.

```
1 Quick sort: 299309000
2 In place quick sort: 68419000
```

Koda 4.12: Rezultati merjenja porabljenega časa algoritmov

Poglavje 5

Zaključek

Glavni cilj, ki smo si ga zastavili, je bil dosežen. Predelali smo že obstoječi javanski navidezni stroj tako, da lahko merimo frekvence javanskih vmesnih ukazov in tako na drugačen način analiziramo različne v javi napisane programe (algoritme). S tem pa naš projekt še zdaleč ni končan, saj se je ob glavnem cilju pojavila veliko večja ideja in še več ciljev ter problemov za reševanje. Razširitev VMEP je še v fazi razvijanja, saj pri razvoju še nismo dosegli končne funkcionalnosti in tudi trenutna implementacija ima že odkrite in lahko tudi še neodkrite hrošče.

Eden od večjih odkritih in še ne rešenih hroščev je odpoved razširitve VMEP ob uporabi grafičnega okolja Swing. V tem primeru se program izvaja normalno, vendar je rezultat opazovanja (meritev) prazna množica.

Najtežji del pri razvoju razširitve do sedaj je bil analizirati in ugotoviti delovanje JamVM. Veliko so nam pomagali komentarji v sami izvorni kodi, vendar pa smo si morali pomagati z ročnim razhroščevanjem (izpisi na standardni izhod).

Zadnjo verzijo implementacije razširitve VMEP skupaj z JamVM je mogoče prenesti iz repozitorija GIT: https://github.com/nikolai5slo/jamvm.

Slike

Literatura

[1] GNU General Public Licence, dostopno na:
https://www.gnu.org/copyleft/gpl.html

[2] Krishna Srinivasan (21.2.2013) What is the difference between JRE,JVM and JDK? dostopno na:
http://www.javabeat.net/what-is-the-difference-between-jrejvm-and-jdk/

[3] JamVM, dostopno na:
http://jamvm.sourceforge.net/

[4] B. Venners, "Inside the Java Virtual Machine", 1. in 2. izdaja, McGraw-Hill Professional, 1999.

[5] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, "The Java Virtual Machine Specification, Java SE 7 Edition", 1. izdaja, Addison-Wesley Professional, 2013, dostopno na:
http://docs.oracle.com/javase/specs/jvms/se7/html/

[6] J. Gosling, B. Joy, G. L. Steele, Jr., G. Bracha, A. Buckley, "The Java Language Specification, Java SE 7 Edition", 1. izdaja, Addison-Wesley Professional, 2013, dostopno na:
http://docs.oracle.com/javase/specs/jls/se7/html/index.html