VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

POROVNÁNÍ VÝKONNOSTI VIRTUÁLNÍHO STROJE CACAO S HOTSPOT JVM

A COMPARISON OF CACAO VIRTUAL MACHINE AND HOTSPOT JVM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **NIKOLAJ MALÍK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. RADEK KOČÍ, Ph.D.**
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/19543/2016/xmalik15

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Malík Nikolaj**
Obor: Informační technologie
Téma: **Porovnání výkonnosti virtuálního stroje CACAO s HotSpot JVM**
A Comparison of CACAO Virtual Machine and HotSpot JVM
Kategorie: Softwarové inženýrství

Pokyny:

1. Nastudujte koncepty a vlastnosti virtuálních strojů pro Javu (JVM) CACAO JVM a HotSpot JVM.
2. Porovnejte vlastnosti virtuálních strojů uvedených v bodu 1.
3. Navrhněte typy aplikací, na kterých lze oba typy virtuálních strojů otestovat. Zaměřte se na použití JVM na desktopech, mobilních zařízeních a serverech.
4. Vytvořte sadu výkonnostních testů (benchmarků), na nichž porovnáte výkonnost JVM HotSpot a CACAO
5. Výsledky výkonnostních testů vhodným způsobem prezentujte a okomentujte.
6. Zhodnoťte dosažené výsledky a navrhněte možnosti dalšího vývoje projektu.

Literatura:

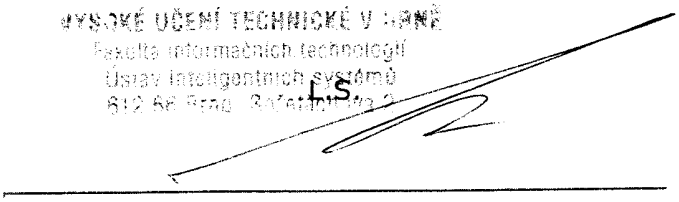
- Bill Venners. Inside the Java Virtual Machine. ISBN 0-07-135093-4
- Tim Lindholm, Frank Yellin. The Java Virtual Machine Specification, Second Edition. ISBN 0-201-43294-3

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kočí Radek, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017



doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Cílem této práce je představit platformu Java a dva vybrané zástupce implementací virtuálních strojů (JVM). V první řadě bude představen rozšířený virtuální stroj JVM HotSpot, v druhé řadě jedna z alternativních variant, JVM CACAO. Práce popisuje vybrané prvky implementace obou strojů, kde uvedené informace jsou vhodné pro všechny, kdo se zabývají optimalizacemi běhu programů. Závěrečná část prezentuje výsledky porovnání vybraných oblastí výše zmíněných strojů z hlediska paměťové a časové náročnosti.

Abstract

The aim of this thesis is introduction of Java platform and two chosen representants of Java Virtual Machine (JVM) implementations. At first JVM HotSpot as widespread one and on the other side JVM CACAO as an alternative option. This thesis describes implementation of chosen areas of both machines. It provides useful information for everybody who deals with runtime program optimalization. Final part presents benchmark results of the match between abovementioned machines by memory and time complexity.

Klíčová slova

JVM, Java, virtuální stroj, HotSpot, CACAO, optimalizace, bajtkód, výkon, test

Keywords

JVM, Java, virtual machine, HotSpot, CACAO, optimalization, bytecode, performance, benchmark

Citace

MALÍK, Nikolaj. *Porovnání výkonnosti virtuálního stroje CACAO s HotSpot JVM*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Kočí Radek.

Porovnání výkonnosti virtuálního stroje CACAO s HotSpot JVM

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Pavla Tišnovského ze společnosti RedHat a pana Ing. Radka Kočího, Ph.D. z Fakulty informačních technologií VUT v Brně. Další informace mi poskytly literární i neliterární prameny a publikace, jejichž vyčerpávající seznam je níže připojen.

.....
Nikolaj Malík
14. května 2017

Poděkování

Tímto bych chtěl poděkovat technickému vedoucímu mé práce, panu. Ing. Pavlu Tišnovskému ze společnosti RedHat, za asistenci při instalaci JVM CACAO.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Počátek vývoje platformy Java se datuje do roku 1990. Přední motivací bylo vytvořit programovací jazyk/technologii, která umožní přenositelnost kódu mezi různými architekturami operačních systémů a hardwarových platforem. Firma Sun Microsystems se vydala cestou vytvoření specifikace virtuálního stroje, jeho rozhraní pro programování aplikací a samotným objektově orientovaným programovacím jazykem Java, který syntakticky vychází z jazyků C/C++. Tyto tři elementy dohromady tvoří základ platformy Java.

První verze Javy byla vydána 23. 5. 1995 (ve stejném rocem byly také vydány jazyky PHP, Javascript nebo RUBY)[9]. Zjistit přesné statistiky využití jednotlivých jazyků je téměř nemožné, většina zdrojů se však shoduje a Java se umisťuje na horních třech stupních nejpopulárnějších jazyků. Magazín IEEE Spectrum v roce 2016 vyhodnotil 12 metrik z 10 různých zdrojů a Java se umístila jako druhý nejlepší programovací jazyk[8]. Více o technologii Java se rozepíšu v kapitole 3, kde především popíšu problematiku běhu programů ve virtuálních strojích.

V následujících kapitolách 4 a 5 rozeberu vlastnosti dvou vybraných virtuálních strojů. JVM HotSpot jako zástupce rozšířené distribuce, který je spravován společností Oracle, a JVM CACAO jako alternativní variantu virtuálního stroje vyvíjenou na Technické univerzitě ve Vídni. Oba stroje porovnám, rozeberu jednotlivé implementace jejich částí, popíšu použité technologie a z nich plynoucí teoretické vlastnosti. V dalších dvou kapitolách 6 a 7 se budu zabývat návrhem výkonnostních testů a prezentací jejich výsledků.

Kapitola 2

Virtuální stroje, jejich význam a historie

2.1 Historie virtuálních strojů

Pro popis a pochopení jednotlivých konkrétních implementací Java virtuálních strojů je nezbytné nejprve nastínit obecný koncept virtuálních strojů a stručně popsat historii jejich vývoje společně s motivací, která vývojáře vedla k rozvoji tohoto konceptu.

Návrh konceptu virtuálního stroje sahá do 50. let 20. století a je úzce spjat s firmou IBM. Formální specifikace se virtuálnímu stroji dostalo až v roce 1974 prací *Formální požadavky pro virtualizaci architektur třetí generace*[14], která definuje virtuální stroj jako *účinný a izolovaný duplikát skutečného počítačového stroje*. Přestože v dnešní době došlo k odklonu od této původní definice, myšlenka zůstává – abstrahování hardwarové vrstvy pro různé účely.

V dalších desetiletích byl původní koncept virtuálního stroje rozpracován a v 70. letech se objevila první verze virtuálního stroje, která vytvářela abstrakci hardwaru pro vývoj v jazyce SmallTalk[12]. Využití virtuálního stroje v tomto případě mělo za cíl kompilovat jednotlivé metody do nativního strojového kódu a významně urychlit překlad programů. Platfroma Java tedy nebyla první, jež přišla s virtuálním strojem, dokázala ale tento koncept masově prosadit a dostat „své“ virtuální stroje na miliony zařízení.

Implementace virtuálního stroje pro jazyk SmallTalk ovlivnila velké množství systémů. Jedním z nich byl i virtuální stroj pro jazyk SELF. Dva vývojáři, kteří se významně podíleli na implementaci stroje pro jazyk SELF, založili v roce 1994 v Palo Alto[1] společnost Animorphic Systems. V rámci této společnosti potom pracovali na vylepšení vlastností tohoto stroje (především vylepšení správy paměti a rychlosti kompilátoru). I přes jejich původní záměr použít tuto technologii na vylepšení stroje pro SmallTalk, v roce 1995 neunikla jejich pozornosti vzrůstající popularita jazyka Java. Využili tedy dosavadních rozsáhlých zkušeností s virtuálními stroji a implementovali vlastní virtuální stroj pro jazyk Java (dále také jako JVM, Java Virtual Machine)[10]. O dva roky později převzala kontrolu nad společností Animorphic Systems firma Sun Microsystems, která se v roce 2009 stala součástí firmy Oracle.

[1] Palo Alto se nachází v Silicon Valley (Kalifornie, USA).

Klíčovovou informací je, že virtuální stroj vyvinutý společností Animorphic Systems a založený na stroji pro jazyk SELF, popřípadě SmallTalk, je právě JVM HotSpot[2][7], který je rozvíjen až do současnosti.

2.2 Koncept a popis virtuálního stroje

Virtuální stroj poskytuje abstrakci k rozhraní hardwaru. Tímto se program zbavuje zodpovědnosti za svůj běh v daném prostředí, tato zodpovědnost je totiž delegována na virtuální stroj. Vždy je základním úkolem virtuálního stroje namapovat své rozhraní na rozhraní nižší vrstvy. Právě tato vlastnost je obvykle společná všem typům virtuálních strojů. Zdroje reálného systému mohou být namapovány na rozhraní i více než jednoho vrituálního stroje s rozdílnými vlastnostmi.

Monitor VM, program zodpovědný za řízení virtuálního stroje, někdy též jako hypervizor, spravuje požadavky na hardwarové zdroje jednotlivých virtuálních strojů na jednom systému tak, aby bylo dosáhnuto naprosté izolace těchto strojů. Oddělení více virtuálních strojů na jednom systému musí být shodné s fyzicky oddělenými systémy.

Této výhody se mimo jiné často využívá pro ochranu soukromých dat v síti. Vytvořením dvou virtuálních strojů v jednom reálném systému, kdy jeden má přístup pouze do soukromé a druhý pouze do veřejné sítě, jsme schopni posílit ochranu dat v privátní síti před útoky zvenčí.

Encapsulation, zapouzdření, je vlastnost virtuálních strojů, která umožňuje monitoru VM zacházet s virtuálním strojem jako s jedním celkem (i když v něm běží více procesů). V současné době se rozšiřuje vlastnost, kdy rozhraní virtuálního stroje se spouští jako samostatná komponenta – služba nad virtuálním strojem[11]. Zapouzdření také umožňuje jednodušší přenášení virtuálních strojů mezi platformami.

Virtuální stroje pomáhají řešit problémy s testováním softwaru. Na jednom reálném systému lze otestovat produkt v mnoha různých prostředích. Výhody doplňuje i řešení problému, kdy všechny potřebné programy nemusí být dostupné pro zvolený systém, virtuálním strojem lze však virtualizovat jiné systémy a dosáhnout tak plné kompatibility.

Seznam nevýhod virtuálního stroje je krátký, přesto velmi významný. Jedná se o ztrátu přímé výkonnosti systému, neboť je zde mnohem větší režie, než se požadavek z virtuálního stroje dostane na reálný hardware a zpět. Snahou je však tyto negativní dopady efektivně minimalizovat kvalitním návrhem/implementací virtuálního stroje a jeho monitoru.

Implementace kompletního virtuálního stroje je náročný proces. Je nezbytné zajistit výhradní přístup ke zdrojům reálného systému. S tím souvisí zvýšení nároků na plánovací činnosti (především správa CPU a paměti). Systémové volání virtuálního stroje pozastaví jeho běh, kontrolu získá monitor VM, který deleguje požadavek na reálný systém. Po zpracování požadavku monitor VM přenese výsledky do virtuálního stroje – podle potřeby upraví jeho registry a opětovně ho spustí.

2.3 Monitor virtuálního stroje

Nepřímou součástí virtuálních strojů je jejich monitor (hypervizor). Obecně existují dva typy těchto monitorů, základním rozlišením je umístění virtualizační vrstvy[14].

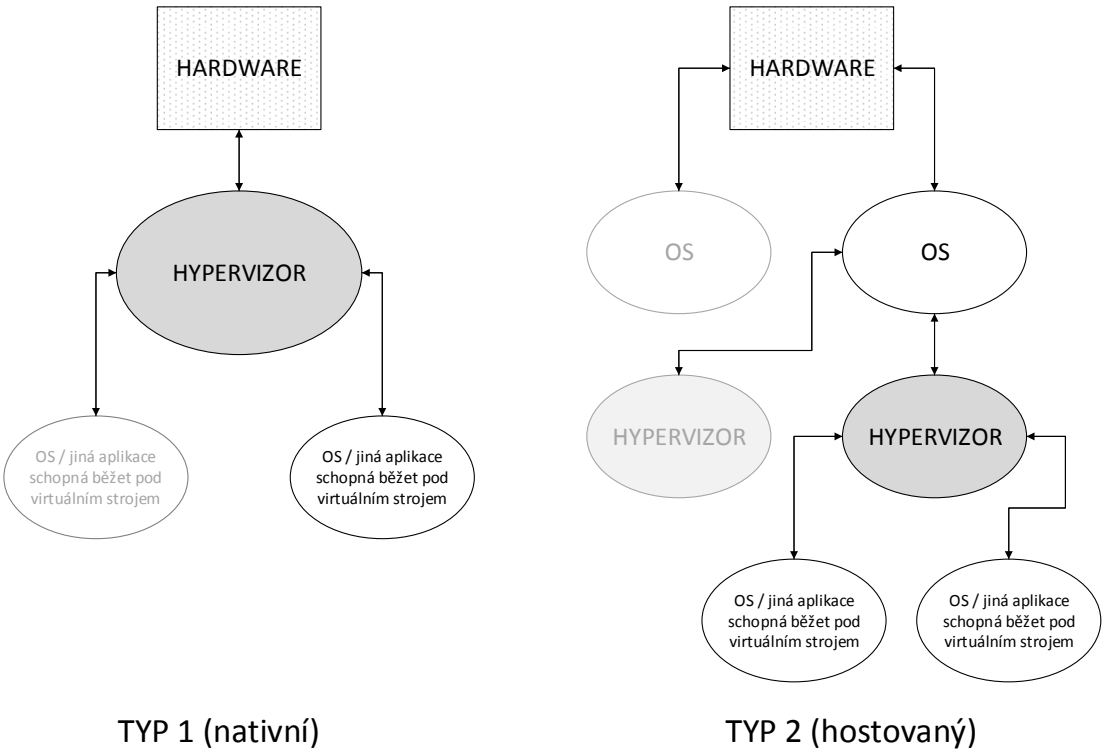
- Typ 1: nativní

[2]Název techniky *HotSpot* si firma Animorphic Systems dokonce nechala zapsat jako ochrannou známku.

- Typ 2: hostovaný

Monitor typu jedna je spuštěný přímo nad reálným hardwarem. Řídí běh jednotlivých virtuálních strojů především z hlediska hardwarových prostředků. Jedná se o původní monitor, jehož historie sahá až k firmě IBM do roku 1960, kdy vyvinuli pvní hypervizor CP/CMS.

Monitor typu 2 je spouštěn až pod operačním systémem, jeho úkolem je tedy předávat požadavky do jádra operačního systému.



Obrázek 2.1: Schéma typů hypervizorů

2.4 Rozdělení virtuálních strojů

2.4.1 Hardwarový virtuální stroj

Termínem hardwarový virtuální stroj, často jen virtuální stroj, je označován původní význam virtuálního stroje. Označuje systém spouštěný v jiném systému (ten slouží jako VM monitor). Nejčastější případ tohoto typu je virtualizace celého operačního systému v jiném systému.

Hardwarový virtuální stroj se podle hloubky virtualizace (neboli umístění virtualizační vrstvy) dělí na tři typy:

1. Emulace (virtuální stroj simuluje kompletní hardware (kterýkoliv – nemusí to být jen ten fyzicky přítomný), požadavky nejdou přímo na hardware, ale nejprve projdou emulací – procesem, při kterém jsou převedeny na požadavky na přítomný hardware)
2. Plná virtualizace (abstrahuje přítomný hardwarový systém)

3. Paravirtualizace (nesimuluje kompletní hardware, nabízí místo toho k němu určité API, vyžaduje spolupráci operačního systému, důvodem může být nemožnost virtualizovat daný prostředek nebo zjednodušit hardwarový systém)

4. Nativní virtualizace (virtuální stroj simuluje jen část hardwaru, zbytek obvykle využívá hardwarovou podporu pro virtualizaci, vyžaduje speciální hardware, který toto umožňuje)

2.4.2 Aplikační virtuální stroj

Počítačový program, který dovoluje spouštět aplikace určené právě pro tento druh programu, vyžadující obvykle konkrétní jazyk nebo kód. Aplikace napsané pro tento typ virtuálního stroje může běžet na kterékoliv platformě, kde se nachází virtuální stroj podporující daný typ aplikace. Aplikační virtuální stroj používá interpret nebo Just-in-time kompilaci (JIT), aby vykonal kód spouštěného programu. Typickým příkladem je právě Java virtuální stroj.

2.4.3 Další typy

Mezi další typy virtuálních strojů patří například virtuální soukromý server (virtuální prostředí). Vytváří virtuální prostředí pro běh programů na úrovni uživatele operačního systému (nevirtualizuje tedy jádro operačního systému a ovladače hardwaru, ale běží na vyšší úrovni).

Méně často se objevuje pojem virtuální stroj ve spojení s počítačovým clusterem. Jedná se však obvykle také o obdobu virtuálního stroje, neboť jeden stroj zapouzdřuje mnoho počitačů do jednoho velkého celku.

Virtuální stroje představují velmi používanou technologii již několik desetiletí a jejich vývoj stále není u konce. Současné trendy se v této oblasti ubírají několika směry. Jedním z nich jsou tzv. virtuální servery. Ty mohou být spouštěny i v cloudu a využít tak plně a efektivně výkon jednoho reálného stroje. Druhou oblastí, ve které probíhá intenzivní výzkum, je použití konceptu virtuálních strojů k efektivnímu využití vícejádrových systémů. V neposlední řadě také společnosti v oboru vývoje upouští od strategie mít aplikaci pro každý systém, ale preferují využití konceptu virtuálních strojů.

Odhaduje se tedy přesun virtuálních strojů z pozadí do popředí zájmu nejen odborné veřejnosti, kdy reálný systém nebude tvořit základ pro vývoj (omezujíce vývojáře zvolenou platformou), ale virtuální prostředí bude základním kamenem vývojáře (které mu poskytne svobodu). V ideálním případě bude jednotné virtuální prostředí zabudováno v každé platformě[16]. Tuto vlastnost však začala technologie Java rozvíjet již před více než dvěma desetiletími.

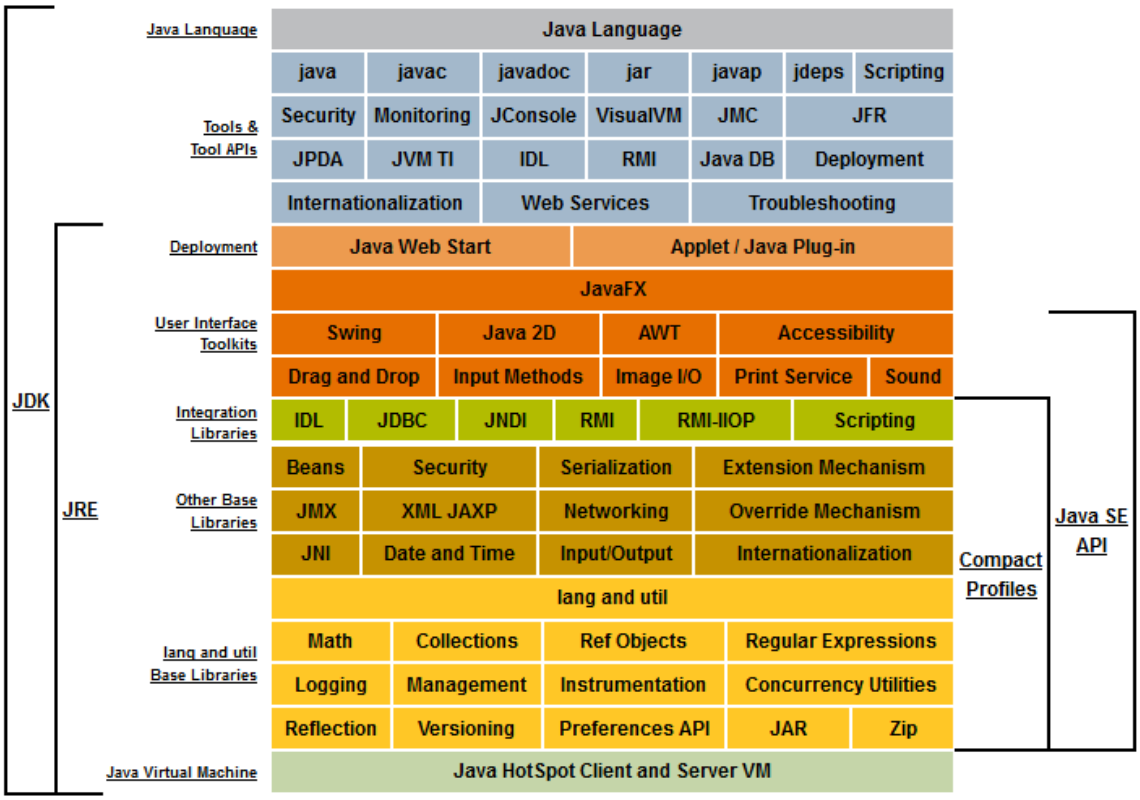
Kapitola 3

Platforma Java

Java bývá často označována jen jako programovací jazyk, ve skutečnosti je to však celá technologie, která se skládá z těchto hlavních částí:

1. Specifikace programovacího jazyka Java (jazyk Java)
2. Rozhraní pro programování aplikací (Java API)
3. Specifikace Java virtuálního stroje (JVM)

Java API společně s JVM tvoří platformu Java, pro kterou lze programovat v jazyce Java.



Obrázek 3.1: Konceptuální diagram platformy Java verze 8[3]

3.1 Historie platformy

Původní návrh platformy Java byl pouze interním projektem firmy Sun na počátku 90. let 20. století. Měla činit lepší alternativu k rozšířeným jazykům C/C++. První stabilní verze byla zveřejněna jako JDK 1.0.2 v lednu 1996, později označovaná jednoduše jako Java 1 (již během roku 1995 byly zveřejňovány nestabilní beta verze). Od té doby prochází platforma Java stále vývojem, v letech 2006–2007 byla Java postupně uveřejněna jako open-source software. Část knihoven, které jsou součástí běhového prostředí Javy, však bylo potřeba re-implementovat, neboť původní verze byly zatíženy autorskými právy třetích stran. V roce 2009 byla společnost Sun včetně projektu Java skoupena firmou Oracle, která je nyní zodpovědná za směrování a specifikace platformy Java. V současnosti (duben 2017) je nejaktuálnější verze Java SE 8 vydaná v březnu 2014, avšak pracuje se již na verzi 9, která je dustupná zatím v režimu předběžného přístupu, datum vydání stabilní verze zatím není známo.

Motivací pro vytvoření jazyka Java bylo umožnit programátorovi soustředit se primárně na implementovaný algoritmus a vedení k čistě objektově orientovanému kódu. Jazyk Java vypouští pojmy ukazatel a odkaz, místo toho jsou základní typy vždy předávány hodnotou a objekty odkazem. Programátor není ani zatěžován správou paměti. Odstranila také tzv. *diamond problem* z jazyka C++, tudíž Java obecně neumožňovala vícenásobnou dědičnost (respektive tato vlastnost měla určitá omezení; Java SE 8 však zavedla i klíčové slovo `default`, které umožňuje vícenásobně dědit chování určením výchozí metody v rozhraní třídy)[3].

Motivací pro vznik celé platfromy Java včetně virtuálního stroje bylo uspokojivě vyřešit problém přenositelnosti aplikací.

Java se rychla rozšířila a je vhodná pro širokou škálu projektů od drobných webových komponent (především v minulosti) přes mobilní/klasické desktopové aplikace až po rozsáhlejší serverová řešení.

3.2 Bajtkód

Přímým překladem a interpretací zdrojových souborů jazyka Java ve virtuálním stroji by se nedalo dosáhnout optimální rychlosti běhu Java aplikací. Z toho důvodu bylo rozhodnuto o překládání zdrojových kódů v jazyce Java do tzv. bajtkódu (mezikódu). Překladač Java (nejčastěji například nástroj `javac` – *java compiler*) přeloží zdrojové soubory z jazyka Java do mezikódu a provede všechny typické úkoly překladače, jako např. syntaktická a sémantická kontrola, základní optimalizace atp. Tento vygenerovaný mezikód je na všech platformách stejný a umí ho zpracovat každý Java virtuální stroj. Ten pak kód může pouze interpretovat (vykonávat). Bajtkódové soubory jsou označeny příponou .class. Každá třída zdrojového kódu Javy včetně anonymních tříd či rozhraní získá po překladu svůj bajtkódový soubor.

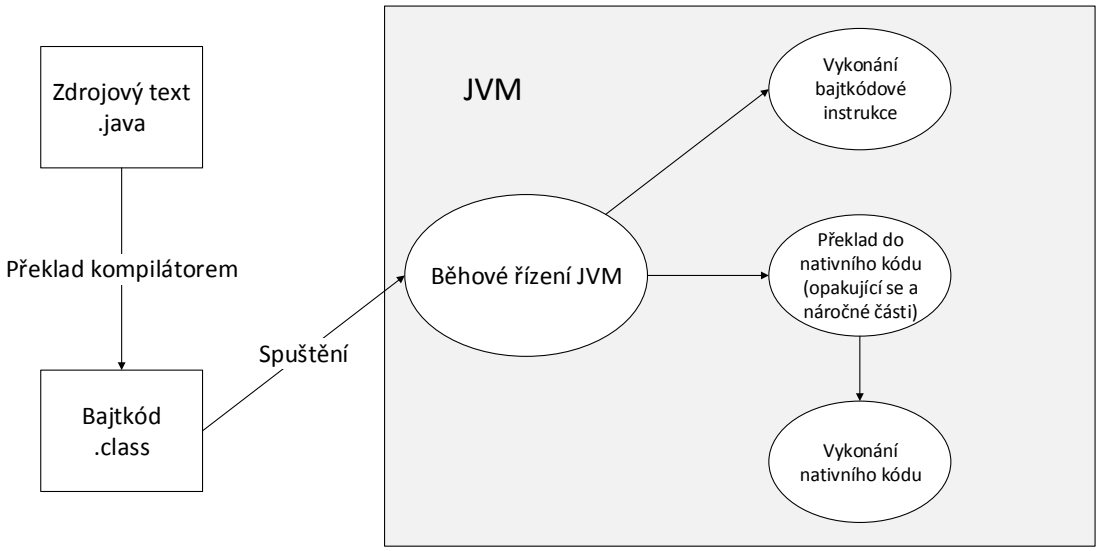
Bajtkód nemusí být využit jen pro spuštění napsaného programu, ale i pro důkladnou analýzu kódu, pokrytí kódu testy a dalšími nástroji.

3.2.1 Bajtkódový soubor

Bajtkódový soubor obsahuje instrukce přeložených Java zdrojových souborů. Jeden soubor reprezentuje jednu třídu, rozhraní nebo výčet. Pojem bajtkód v kontextu platformy Java

označuje intstrukce přeložených zdrojových souborů z jazyka Java pro Java virtuální stroj. Instrukce jsou zapsány ve formátu operačního kódu a 0 a více jeho operandů.

Počátek souboru musí obsahovat na prvním místě magickou konstantu 0xCAFEBABE v šestnáctkové soustavě, která jednoznačně identifikuje platný .class soubor. Na druhém místě by se měla objevit majoritní a minoritní verze souboru, která určuje, pro kterou verzi platformy Java je zdrojový kód přeložen. JVM vždy vyhodnotí podle této hodnoty, zda je vůbec schopen takový kód korektně spustit. Rozsah jednotlivých verzí, které musí daný stroj podporovat, je součástí specifikace každého JVM a je určen specifikací společnosti Oracle. Jedná se zde především o to, že různé verze platformy Java mohou obsahovat různá API, jehož se dovolává aplikace v průběhu svého běhu.



Obrázek 3.2: Zjednodušené schéma spouštění Java aplikace od zdrojového kódu po vykonání instrukce

Bajtkódový soubor také obsahuje *contstant pool*, který odpovídá tabulce symbolů. Její obraz se zkopíruje do paměti za běhu programu, aby se s ní dalo pracovat. Bajtkódová třída také obsahuje informace o dědičnosti a modifikátorech přístupu, tyto informace jsou shrnuty v jedné šestnáctibitové masce, která zatím není plně využita – některé bity jsou rezervovány pro budoucí použití. Soubor také obsahuje symbolické odkazy, a to především na svou třídu (`this`) a svou nadtřídu/předka (`super`). Dále se zde nachází odkazy do constant poolu na seznam implementovaných rozhraní, seznam třídních proměnných včetně jejich modifikátorů a na závěr implementace jednotlivých metod třídy.

Ilustrační kód bajtkódu, který je uveden níže, vznikl překladem tohoto jednoduchého fragmentu zdrojového kódu v jazyce Java:

```
public class demoClass {
    public int variable1 = 100;
}
```

Vygenerovaný bajtkód výše uvedené třídy, která neobsahuje žádnou metodu, pouze jeden atribut s výchozí hodnotou, nástrojem *javac*:

```
public demoClass();
```

```
  Signature: ()V
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=1, args_size=1
       0: aload_0
       1: invokespecial #1        // Method java/lang/Object."<init>":()V
       4: aload_0
       5: bipush        100
       7: putfield      #2       // Field variable1:I
      10: return
```

Constant pool bude obsahovat tyto informace:

```
Constant pool:
   #1 = Methodref          #4.#16         //  java/lang/Object."<init>":()V
   #2 = Fieldref           #3.#17         //  demoClass.variable1:I
   #3 = Class              #13            //  demoClass
   #4 = Class              #19            //  java/lang/Object
   #5 = Utf8               variable1
   #6 = Utf8
I~#7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               LocalVariableTable
  #12 = Utf8               this
  #13 = Utf8               demoClass
  #14 = Utf8               SourceFile
  #15 = Utf8               demoClass.java
  #16 = NameAndType        #7:#8          //  "<init>":()V
  #17 = NameAndType        #5:#6          //  variable1:I
  #18 = Utf8               LSimpleClass;
  #19 = Utf8               java/lang/Object
```

Pokud by vygenerovaný bajtkód metody vyžadoval příliš objemná data, tak tato data nebudou obsaženy přímo v něm, ale bude zde pouze odkaz do constant poolu. Každá položka v constant poolu je složena ze svého indexu v constant poolu, svého datového typu a své hodnoty, která může obsahovat buď přímo hodnotu, nebo odkaz na jiné místo v constant poolu. Odkazování mimo vlastní constant pool probíhá pomocí řetězců (například na indexu #19).

Bajtkód Java SE 8 může obsahovat až 207 různých instrukcí. Jedním z důvodů pro velké množství instrukcí je jednodušší detekce náročnějších částí aplikace během rozhodování o provedení interpretace/kompilace do nativního kódu (více popsáno v kapitole 3.3).

3.3 Běhové prostředí Javy

Ke spuštění přeloženého programu v jazyce Java jsou nezbytné dvě komponenty, které společně tvoří tzv. běhové prostředí (*runtime environment*):

- Java virtuální stroj (kapitola 3.3.1)

- Java API (kapitola 3.3.2)

3.3.1 Java virtuální stroj

Java virtuální stroj (Java Virtual Machine – JVM) je program, který zajišťuje spouštění a běh programů napsaných v jazyce Java. Je zodpovědný za načítání tříd, tvorbu objektů, správu haldy a další operace. Obecně lze říci, že vytváří kompletní virtuální prostředí, ve kterém vykonává instrukce zapsané v bajtkódových souborech.

Implementovat Java virtuální stroj lze v jakémkoliv programovacím jazyce, který je podporován na dané platformě, pro kterou vytváříme tento stroj. Nejčastěji se používá C/C++, popř. jiné nativně kompilované jazyky, kdy při jejich výběru by měl být kladen důraz na rychlost (především kvůli schopnosti interpretace bajtkódu). Druhým, o něco méně důležitým kritériem, je přenositelnost JVM na další platformy. Implementace takového programu je velmi náročná, proto se obvykle projevuje snaha využít jednu implementaci ve větším množství prostředí s co nejmenšími úpravami.

Společnost Oracle vydává pro každou verzi platformy Java přesné technické požadavky, které musí splňovat každý Java virtuální stroj. Tyto požadavky jsou vzhledem k faktu, že se Java stala open-source projektem, dostupné elektronicky zdarma online[1] a také jako tištěná kniha dostupná v oborových knihkupectvích. Tyto požadavky obsahují všechny nezbytné náležitosti, které musí JVM splňovat, aby korektně spustil jakýkoliv program napsaný pro danou verzi platformy. Technické požadavky obsahují popis instrukční sady bajtkódu, obecný popis konstrukce virtuálního stroje, přístup k programování v jazyce Java, spojování a načítání tříd, popis vláken a práce s pamětí a další aspekty spojené nejen s tvorbou virtuálního stroje, ale s celou platformou Java[13]. Specifikace také obsahuje popis abstraktního modelu vykonávání programu. Ten popisuje, že každá třída, jejíž metoda je volána, musí být nejprve zcela načtena, správně nalinkována a inicializována.

Největší důraz je při implementaci JVM kladen na správnou implementaci datových typů a dodržení syntaxe a sémantiky bajtkódových instrukcí dle zvolené verze platformy Java. V rukou vývojáře JVM je kromě samotné implementace především optimalizace výkonu JVM (časových a paměťových nároků).

3.3.2 Java API

Java API by se dalo shrnout jako soubor knihoven pro jazyk Java, které má vývojář v jazyce Java vždy na všech platformách, pro něž existuje virtuální stroj, k dispozici.

Dostupnost knihoven je jeden z problémů, který výrazně ztěžuje přenositelnost programů napsaných např. v C/C++, jelikož jejich implementace se na každé platformě mohou různit, nebo nejsou vůbec k dispozici. Zde právě přichází Java API, jehož knihovny jsou díky spouštění ve virtuálním stroji na všech platformách stejné, a jeho přítomnost zajišťuje běhové prostředí virtuálního stroje, knihovny tedy nejsou přímou součástí aplikace.

Java API je rozsáhlá nabídka mnoha knihoven, které se dělí do jednotlivých balíčků (orientačně několikati desítek). Obecně jsou vývojáři velmi využívané, ovšem jejich nadměrné nebo nevhodné použití může programy značně zpomalovat.

Vzhledem ke svému rozsahu se Java API dělí do čtyř základních balíků, tzv. platforem:

[1] Nejnovější technické požadavky (Java SE 8) Java virtuálního stroje jsou dostupné jako online pdf kniha na internetových stránkách společnosti Oracle https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf

- Java SE (Standard Edition): implementace vhodná především pro aplikace určené na desktopové počítače
- Java EE (Enterprise Edition): její základ tvoří Java SE, nad ní jsou však postaveny další knihovny umožňující serverové nasazení aplikací, obsahuje také nativně databázové API a transakční model
- Java ME (Micro Edition): určená pro zařízení s omezenými prostředky, obsahuje podmnožinu knihoven z verze SE
- Java Card: je vhodná pro malé vestavěné zařízení s vlastní pamětí (karty SIM, bankovní karty atp.)[2]

3.4 Přehled a rekapitulace

Platforma Java nabízí možnost vytvářet přenositelné aplikace cílené na zařízení širokého rozsahu od drobných čipových karet, přes mobilní telefony a desktopové zařízení po rozsáhlá serverová řešení. Jazyk Java nabízí přísnou objektovou syntaxi podobnou mnoha dalším jazykům, například velmi rozšířenému C++, který je jedním z jeho hlavních konkurentů. Vzhledem k současné obecné rozšířenosti této platformy lze spoustět aplikace v jazyce Java na všech základních i mnoha dalších platformách díky velkému množství implementací opensourcových i komerčních virtuálních strojů.

Za těmito výhodami však následuje seznam nevýhod, ve kterém dominují zvýšené nároky na výkon a paměť zařízení. Virtuální stroj tvoří přidanou softwarovou vrstvu mezi spouštěnou aplikací a operačním systémem, a tím zákonitě dochází k poklesu výkonu.

Následující kapitoly se zabývají porovnáním dvou virtuálních strojů z hlediska poskytovaného výpočetního výkonu a využití zdrojů. Prvním z nich je Java Virtual Machine HotSpot, který tvoří zástupce dlouho vyvíjeného a rozšířeného stroje společnosti Oracle. Druhým z nich je Java Virtual Machine CACAO, který zastupuje skupinu alternativních možností. Kompaktnější JVM CACAO je sice vyvíjen o něco déle, ovšem s přihlédnutím k faktu, že se jedná o akademický projekt, také podstatně pomaleji.

[2] Některé zdroje uvádějí platformu Java Card jen jako součást platformy Java ME.

Kapitola 4

JVM HotSpot

Java Virtual Machine HotSpot je vyspělou implementací Java virtuálního stroje od firmy Oracle (většinu životního cyklu však tvořen „pod taktovkou“ společnosti Sun Microsystems). První verze byla vydána v roce 1999 a od roku 2006 je stroj šířen pod svobodnou licencí GPL 2. HotSpot je stále vyvíjen a udržován pro operační systémy MS Windows, Mac OS X a širokou škálu Linuxových systémů. Podporované instrukční sady jsou IA-32, x86-64, ARMv6, ARMv7 a SPARC (pouze OS Solaris). Implementace pro každou platformu je velmi náročná vzhledem k povaze stroje – zdrojové kódy obsahují mnoho strojového (assemblerového) kódu, a proto přenášení mezi platformami není jednoduché. Existují však odlehčené verze i na méně obvyklé platformy, ty však v podstatě nepoužívají klíčovou HotSpot metodu. Mimo strojového kódu je většina zdrojových kódů JVM HotSpot psána v jazyku C++ a využívá většinu jeho hlavních vlastností a výhod jako dědičnost, šablony, zásobníkově alokované objekty a další.

Pojmenování virtuálního stroje HotSpot prozrazuje jeho hlavní sílu: *hot* jako horký a *spot* jako místo. JVM HotSpot tedy vyhledává *horká místa* (ve smyslu často spouštěných metod) v programu. Mimo to obsahuje Just-in-time kompilaci.

Distribuce JVM HotSpot obsahuje dva rozdílné režimy provozu:

- JVM HotSpot Client: verze optimalizovaná pro běh aplikací menšího rozsahu v prostředí stanic se slabším výkonem. Klíčovými vlastnostmi jsou nízká spotřeba paměti, krátká doba mezi žádostí o start programu a jeho skutečným spuštěním a relativně dobrý výkon. V tomto režimu nejsou plně uplatněny *HotSpot* výhody a jde především o vylepšený Just-in-time kompilátor.

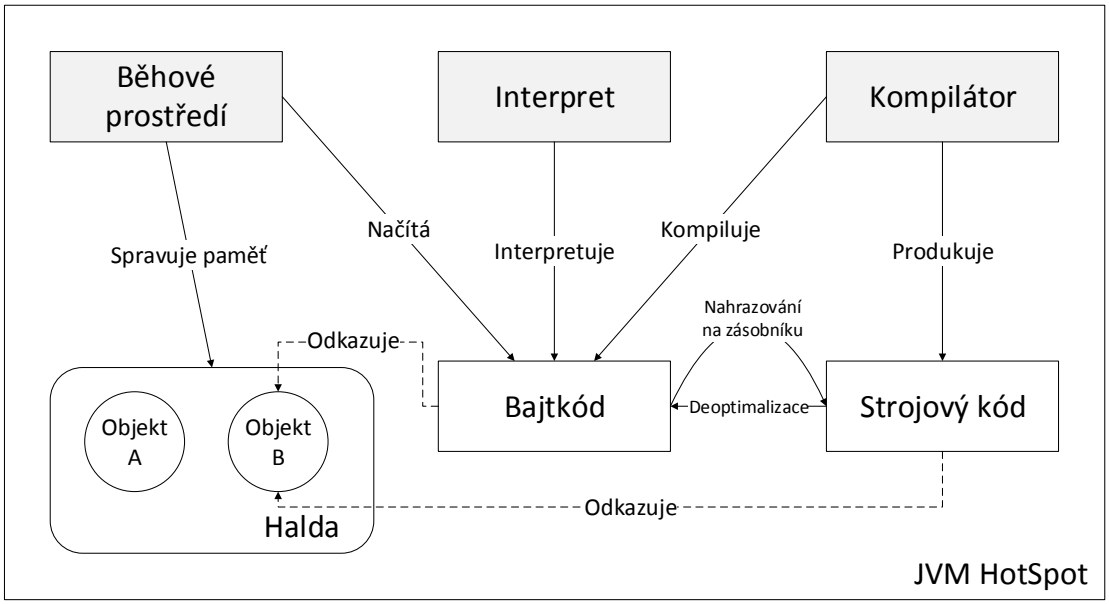
- JVM HotSpot Server: verze zacílená pro běh dlouhodobě spuštěných serverových aplikací. Klíčový aspekt zde jednoznačně tvoří rychlost vykonávaného programu. V tomto režimu má JVM čas provést analýzu bajtkódu před skutečným spuštěním programu. Za cenu delší doby spouštění můžeme dosáhnout vyrázně kvalitnějšího kódu. V dalším textu se tedy obvykle bude mluvit o verzi server, neboť jen ta využívá plný potenciál stroje JVM HotSpot.

Výběr daného režimu se provádí při spuštění uvedením přepínače `-client` nebo `-server`, výchozí hodnotou je typ server.

4.1 Běh a optimalizace

Mimo často volaných metod, které JVM HotSpot kompiluje do nativního kódu, může bajtkód obsahovat i mnoho dalších míst, které jej mohou zpomalovat. JVM HotSpot má však široký záběr optimalizačních technik.

JVM HotSpot se skládá ze tří hlavních modulů: běhového prostředí, interpretu a kompilátoru. Více na obrázku 4.1.



Obrázek 4.1: Schéma konstrukce Java virtuálního stroje HotSpot

4.2 Odstupňovaná kompilace

JVM HotSpot přišel ve verzi Java 7 a ve verzi 8 zdokonalil tzv. odstupňovanou kompilaci, *Tiered Compilation*. Kompilátor kompiluje metody, které dokáží odevzdávat virtuálnímu stroji profilovací informace o sobě samotných. Tímto typem optimalizace dokonce dokáže JVM HotSpot často spustit program v režimu server ještě rychleji než v režimu client. Cílový kód produkovaný server kompilátorem je virtuálnímu stroji ihned dostupný a nezřídka se stává, že jeho části jsou využity právě ještě v raných fázích inicializace. *Tiered Compilation* dává stroji více času k hloubkovějšímu profilování a analýze kódu (díky času získanému kompilací inicializačních metod), která vede k ještě lepším optimalizacím[2].

4.3 Úniková analýza

Virtuální stroj HotSpot v režimu server provádi tzv. únikovou analýzu (*escape analysis*). Dochází k rozboru nově vznikajících objektů za běhu programu za účelem rozhodnutí, zda je potřeba pro tyto objekty alokovat místo na haldě. Virtuální stroj zjistí, na kterých místech v programu může být ukazatel nově vzniklého objektu použit a jestli se může dostat mimo vlastní metodu, popř. vlákno.

V následujícím fragmentu kódu se v metodě `example` vytvoří dva objekty, kde jeden je argumentem druhého. Metoda `setFoo` uloží odkaz na přijatý objekt. Kompilátor zjistí, že objekt `Bar` sám o sobě nemůže opustit metodu `example`. Objekt `Foo` odkazovaný z objektu `Bar` také nemůže uniknout z tohoto prostoru. Kompilátor tak oba objekty bezpečně alokuje pouze na zásobníku.

```
class Main {
    public static void example() {
        Foo foo = new Foo();
        Bar bar = new Bar();
        bar.setFoo(foo);
    }
}

class Foo {}

class Bar {
    private Foo foo;
    public void setFoo(Foo foo) {
        this.foo = foo;
    }
}
```

Únikovou analýzou virtuální stroj zjistí únikový stav nově vzniklého objektu. Tento stav může nabývat jedné ze tří hodnot[2]:

- *Global escape*: objekt může uniknout ze své metody i vlákna, například objekt uložený ve statické třídní proměnné, nebo objekt, který je odkazován z jiného objektu, jež sám nabývá stavu *global escape*.
- *Argument escape*: objekt nebo odkaz na objekt, který je daný jako argument metodě, ale nedokáže z volané metody uniknout globálně. Ke zjištění tohoto stavu je potřeba analyzovat bajtkód volané metody.
- *No escape*: skalární nahraditelný objekt, alokace paměti může být z generovaného kódu zcela odstraněna.

4.4 Optimalizace cyklů

Vykonávání kódu, který se opakuje v dlouho běžících cyklech, má velmi negativní dopad na celkovou výkonnost. Interpretace těchto cyklů nezvyšuje tzv. *invocation count* metody, ačkoliv vykonáním cyklu může stroj ztratit více času než vykonáním zbytku metody. HotSpot řeší tento problém ještě dalším typem optimalizace, tzv. *on stack replacement*. Virtuální stroj si udržuje čítač cyklů pro každou metodu. Pokud hodnota tohoto čítače překročí určitou mez, vykonávání kódu cyklu pokračuje v kompilované verzi.

4.5 Deoptimalizace

JVM HotSpot používá relativně agresivní způsob optimalizace při kompilaci do strojového kódu za využití matematické pravděpodobnosti, kde se předpokládá, že určité stavy v pro-

gramu nenastanou, přestože by teoreticky mohly. Virtuální stroj však zavádí mechanismus, kterým ověřuje, zda optimalizace kódu je korektní. Pro ověření, zda určité stavy nenastanou, vkládá kompilátor do kódu kontrolní body, které se kontrolují za běhu vůči určité hodnotě. Pokud dojde ke zjištění, že hodnota je mimo optimalizační záběr, dojde k deoptimalizaci[5]. Zkompilovaný rámec nativního kódu je vrácen a dojde k jeho interpretaci bez překladu do strojového kódu.

4.6 Správá paměti

Jako každý Java virtuální stroj i HotSpot obsahuje garbage collector pro automatickou správu paměti. JVM HotSpot využívá generační přístup ke správě paměti. Objekty jsou na haldě rozděleny do oddílů podle svého stáří. V každé generaci probíhá správa paměti v různých časových intervalech. U nejmladší generace probíhá nejčastěji, neboť je obvyklé, že nejvíce objektů se stává odpadem krátce po svém vzniku.

Garbage collector zná přesné vztahy mezi adresami do paměti a objekty v běhovém prostředí. Pokud je potřeba provést úklid paměti, běh interpretu bajtkódu je pozastaven na aktuální pozici. V případě kompilovaného kódu je využit mechanismus bezpečných bodů (*safepoints*). Jedná se o speciální místa v kódu, kde je na zásobníku známo umístění interních ukazatelů na objekty[6]. Pokud je bezpečný bod v kódu dosažen a úklid paměti je v tu chvíli požadován, vykonávání kódu je také pozastaveno a úklid je proveden[1].

[1] Problematika správy paměti je velmi rozsáhlá a převyšuje cíl této práce. Velmi dobře je však popsána v práci *Memory Management in the Java HotSpot Virtual Machine*, která pojednává o HotSpot garbage collectoru, která je dostupná ve formátu pdf na oficiálních stránkách společnosti Oracle[6].

Kapitola 5

JVM CACAO

Java virtuální stroj CACAO byl navržen v roce 1996 na Technické univerzitě ve Vídni (Technische Universität Wien). Prvotním cílem bylo vytvořit stroj zaměřený na rychlost. CACAO využívá metodu Just-in-time kompilátoru – kompiluje tedy metodu do nativního kódu až v okamžiku volání.

CACAO je od roku 2004 distribuován pod licencí GPL (General Public Licence). Od té doby probíhá přepis z původní C implementace do C++, který je téměř dokončen. Cílová architektura byla původně pouze 64bitová RISC Alpha. Díky tomu šlo stroj poměrně snadno přenést na jiné 64bitové RISC architektury jako např. MIPS. Seznam současných (verze 1.6 z roku 2013) podporovaných verzí se rozrostl a čítá architektury i386, S390, Alpha, MIPS (32 i 64bit), POWERPC (32 i 64bit), x86-64 a ARM[1]. Tímto CACAO dostalo šanci být téměř na všech v současnosti rozšířených architekturách.

Největší výhodou stroje CACAO by se dala označit jeho malá velikost, která ho umožňuje nasadit i do vestavěných zařízení s omezenou pamětí. JVM CACAO využívá schopnosti hostovaného operačního systému ke správě vláken a vstupně-výstupních operací. Součástí JVM CACAO je ale později doimplementovaná knihovna, která umožňuje spouštění i mimo operační systém. V souvislosti s použitím ve vestavěných zařízeních bylo zjištěno, že se nelze spolehnout na rychlost Just-in-time kompilace v obvykle omezených možnostech výkonu tohoto typu zařízení. Vývojáři tedy iplementovali možnost využít *ahead of time* kompilaci. Během inicializace pak mohou být všechny třídy dopředně načteny bez nebezpečí kolize s vykonáním kriticky důležitých úloh. Tím dojde k upozadění dynamických vlastností stroje ve prospěch požadavků kladených na vestavěná zařízení (nenáročnost na hardwarové zdroje a nízký čas, za který musí být požadovaná operace provedena).

5.1 Překlad do nativního kódu

Vzhledem k podpoře RISC architektury[1], která obsahuje velmi omezené množství instrukcí, dochází během kompilace každé bajtkódové instrukce `load` a `store` k překladu do nativního kódu za využití instrukce `move`. Načítání a ukládání do paměti je poměrně častá činnost, generuje se zde tedy velmi mnoho instrukcí `move`. JVM CACAO je pro tento případ optimalizován – obsahuje interní tabulku, která obsahuje informace o registrech se stejnou hodnotou. Na konci každého bloku je pak instrukce `move` hromadně vygenerována pro všechny registry v tabulce.

[1] RISC označuje procesory s redukovanou instrukční sadou.

JVM CACAO provádí překlad do nativního kódu v několika krocích. Nejdříve dochází k detekci instrukcí skoků a jejich cílů, tímto se kód rozdělí do jednotlivých bloků. Ve druhé fázi jsou bloky procházeny a pro každý blok je vytvořen obraz v přechodném kódu. Do pseudoregistrů virtuálního stroje jsou vloženy lokální proměnné a další dočasné hodnoty jako například parametry funkcí. Pokud je do pseudoregistru umístěn parametr funkce, jeho životnost odráží dobu běhu metody, se kterou je spjat. V ostatních případech je životnost pseudoregistru nižší, protože obsahují nejčastěji operandy zásobníku virtuálního stroje. O tom, zda bude obsah pseudoregistru překlopen do skutečného procesorového registru, se rozhodne v další fázi, alokace registrů.

Modul alokátor registrů zajišťuje můstek a kopírování hodnot mezi pseudoregistry JVM a registry CPU. Vzhledem k faktu, že kód je generovaný za běhu programu, tak je využit jednoduchý sekvenční algoritmus, který mapuje registry k pseudoregistrům podle pořadí v kódu. Použití paměti místo registrů je minimalizováno. Pro architekturu CISC je plánována implementace jiného přístupu k alokaci registrů, jež bude využívat algoritmus typu *linear scan*.

Motivací výše uvedeného postupu je nahradit zásobník virtuálního stroje za pseudoregistry.

5.2 Načítání tříd

JVM CACAO má implementovány dva rozdílné algoritmy pro načítání tříd a rozhraní:

- Eager class loading: „hladové“ načítání tříd – jedná se o přednačtení tříd a rozhraní ještě před tím než jsou skutečně potřeba.
- Lazy class loading: „líné“ načítání tříd – načítá třídy a rozhraní až v okamžiku jejich skutečné potřeby.

5.2.1 Eager class loading

Eager class loading se aktivuje při spouštění uvedením přepínače `-eager`. K načtení tříd dochází co nejdříve ještě před skutečným vykonáním kódu v programu. Později v běhu programu je umožněn rychlejší přístup ke všem třídám použitým v programu, jelikož jsou již načtené a připraveny k použití.

5.2.2 Lazy class loading

Start programu je časově mnohem náročnější s aktivovaným eager class loadingem. Spuštění standardního HelloWorld programu s aktivovaným eager class loadingem provede 513 načtení tříd. Pokud se použije algoritmus lazy class loading, dojde pouze ke 121 načtením. Rozdíl je tedy čtyřnásobný. V souvislosti s tímto algoritmem, JVM CACAO provádí i lazy class linking, což šetří další čas.

Problém je, že implementace lazy class loading v JVM CACAO není zcela v souladu s oficiálními specifikacemi JVM. JVM s implementací lazy class loadingu by měla dle specifikací načítat a linkovat třídy v modulu běhového prostředí. CACAO ovšem díky několika dosud nevyřešeným problémům v implementaci provádí lazy class loading během parsování bajtkódu. Pokud tedy narazí na instrukci `JAVA_PUTSTATIC`, `JAVA_GETFIELD` nebo

`JAVA_INVOKE*`, tak požadavaná třída nebo rozhraní je okamžitě načtena a nalinkována během parsování aktuálně kompilované metody. Tímto jevem dochází k rozporům s ostatními JVM, které specifikace dodržují.

Ku příkladu lze uvést následující fragment kódu:

```
void testOk(boolean b) {
    if (b) {
        new B();
    }
    System.out.println('ok');
}
```

Pokud je výše uvedená metoda zavolána s parametrem `false` a třída B neexistuje, Java virtuální stroj by měl vypsat *ok* a skončit s návratovým kódem 0, tedy vykonat metodu bez potíží. Vzhledem k faktu, že JVM CACAO chce načíst třídu už během parsování a ne správně až v běhovém prostředí, dojde k pokusu načíst třídu B i v případě, že tento kód nebude vykonán. JVM CACAO skončí vyhozením vyjímky `java.lang.NoClassDefFoundError`, která není zachycena, text *ok* se nevypíše a běh programu je ukončen.

Vývojový tým JVM CACAO prozatím nenalezl uspokojivé řešení tohoto problému, neboť přesun načítání a linkování tříd z fáze kompilace do běhového prostředí není triviální. Správná implementace lazy class loadingu je jedna z klíčových vlastností, které mají být v budoucnu implementovány. Prioritou pro vývojáře je udělat vše pro to, aby JVM CACAO vyhovoval technickým specifikacím pro Java virtuální stroje[4].

5.3 Eliminace kopií

Za účelem eliminace nepotřebných kopií hodnot je načtení hodnot pozdrženo, dokud není dosažena instrukce, která danou hodnotu využije. Příkladem může být kód, který načítá z libovolného místa hodnoty x a y a na dalším řádku tyto hodnoty sečte. Hodnoty x a y ve skutečnosti nejsou načteny ve chvíli, kdy je požadováno bajtkódem jejich načtení, ale jsou načteny až ve chvíli, kdy jsou skutečně potřeba, aby virtuální stroj mohl získat jejich součet.

5.4 Zpracování výjimek

Nejčastěji vyhazovanou výjimkou v běhu programů Java je `java.lang.NullPointerException`. JVM CACAO má implementovanou hardwarovou kontrolu platnosti ukazatelů do paměti pro dosažení maximální rychlosti této operace. Pokud dojde k porušení ochrany paměti, je vyslán signál `SIGSEGV`, který CACAO zpracuje v interním modulu zpracování výjimek. Po tomto procesu se kontrolor musí sám znovu umístit na příslušné místo v hardwaru, aby mohly být zachyceny další výjimky.

Pro některé platformy je zpracován i hardwarový kontrolor chyb při zpracování čísel v plovoucí řádové čárce (`SIGFPE`), nejčastěji se jedná o dělení nulou.

5.5 Správá paměti

Implementace garbage collectoru není tak vyspělá jako v případě JVM HotSpot (popsán v kapitole 4.6). Garbage collector stroje CACAO využívá sledovací algoritmus a uplaťnuje

konkrétně přístup *mark and sweep*. Halda je rozdělena na osmibajtové bloky, každý blok obsahuje tři bity metainformací. S-bit je nastaven u bloků, kde začíná nový objekt nebo naopak nový blok volného místa (označení začátku volných bloků S-bitem je především z důvodu získání informace o konci předchozího objektu). M-bit značí, že objekt v bloku je odkazovaný. Třetí bit označuje možnost bloku odkazovat na jiný objekt.

Během první (značkovací) fáze, jsou všechny M-bity vynulovány. Následně se procházením všech odkazů zjišťuje, které objekty jsou dosažitelné – ty jsou po té označeny novým M-bitem. Druhá fáze (čištění) obnoví seznam volných bloků.

V současné době probíhají práce na nové vylepšené implementaci garbage collectoru.

Kapitola 6

Návrh testování

6.1 Prostředí

Ke spuštění a měření výkonnostních testů jsem vybral operační systém Fedora 25 32bit s konfigurací: procesor Intel Pentium P6100 (2 jádra, každé 2 GHz), operační paměť 3,87 GB. Záměrně jsem vybral zcela průměrný stroj. Jednotlivé testy budou také překládány zcela standardně bez různých volitelných přepínačů, kterými by se dala docílit ještě lepší optimalizace v určitých případech. Cílem je totiž porovnat oba virtuální stroje za standardních podmínek.

Zvolil jsem nejnovější dostupnou stabilní verzi obou porovnávaných strojů. V případě JVM CACAO jde o verzi 1.6.1 s podporou JDK verze 7. U stroje JVM HotSpot se jedná o verzi 1.8.0_131 s podporou JDK 8. JVM CACAO bylo zkompilováno za využití GNU Classpath. JVM HotSpot bude testován pouze ve výchozí verzi – server[1].

6.2 Spouštění

Celkově bude provedeno 8 testů, každý jednotlivý test je umístěn ve vlastní třídě (souboru). Každý test bude spuštěn 100krát. Zjištěné hodnoty potom budou náležitě zpracovány a v kapitele 7 prezentovány. Tyto akce budou provedeny dvakrát – pro každý Java virtuální stroj.

Před každou iterací testu je provedeno „zahřátí“ virtuálního stroje provedením určitého kódu, který JVM vždy přivede do (teoreticky) stejného stavu pro každé spuštění testu.

Všechny zdrojové kódy testů, naměřené hodnoty a skripty, kterými byly testy spouštěny, jsou k dispozici na přiloženém CD. Obsah CD je uveden v příloze. Důležité informace ke spuštění jsou uvedeny v souboru `readme.txt`.

6.3 Návrh výkonnostních testů

Konkatenace řetězců a uložení do seznamu

Podstatou testu `ConcatenateAndStoreInList` je provést konkatenaci řetězců operátorem plus a uložit je do generické kolekce seznam typu `ArrayList`. Tato operace bude provedena 5krát v cyklu, který bude spuštěn stotisíckrát. Vysoké číslo je záměrně zvoleno, aby se více

[1] Podle všech dostupných benchmarků má verze klient vždy horší výsledky než verze server (především kvůli absenci technologie HotSpot)

zjevily rozdíly ve výkonu, které mohou být v krátkých programech zanedbatelné. Tento test měří časovou náročnost.

Doba startu JVM

Test `ClassLoading` se zaměřuje na čas startu Java virtuálního stroje, testovaná třída potřebuje ke svému běhu 100 jiných vlastních tříd a další třídy, které jsou obsaženy v JRE. Dochází k prověření implementace classloaderu.

Volání anonymní funkce a řazení pole

Test `CallAnonymousFunction{Time|Memory}` se zaměřuje na schopnost vypořádat se s opakovaným voláním stejné funkce, která je ovšem deklarována pouze v anonymním rozhraní. Tato funkce je volána pro seřazení pole podle délky řetězců. Virtuální stroje zde mají možnost ukázat, jak se chovají v případě opakovaného volání stejného úseku kódu, který ovšem netvoří samostatnou metodu.

Tento test je implementován ve dvou souborech – v jednom případě se měří časová náročnost, ve druhém paměťová, testovaný kód je totožný.

Vytváření objektů a ukládání do pole

V testu `CreatingObjectsAndStoreInArray{Time|Memory}` se cyklicky vytvářejí objekty stejné třídy, nad kterýma je volána stejná metoda, jejíž výstup je ukládán do pole, jehož prvky jsou pravidelně přepisovány těmito výstupy. Objekty po přepsání již nejsou dále potřeba. JVM tedy mohou ukázat, jak se vypořádávají s vytvářením stejných objektů, voláním stejné metody a také se správou paměti (mazáním nedosažitelných objektů).

Test je implementován ve dvou souborech, kdy jeden měří časovou a druhý paměťovou náročnost.

Velký switch blok

JVM specifikace obsahují i specialní bajtkód pro blok switch. Test `SwitchBlock` prověřuje, který z testovaných strojů dokáže efektivněji zpracovat rozsáhlý switch blok o obsahu padesát možných případů. Kód každého case bloku přiřadí řetězec do proměnné a tato proměnná je vložena do třídní statické proměnné, které drží kolekci řetězců typu `ArrayList`. Typické využití rozsáhlého switch bloku může být například v implementaci konečného automatu, kterým může být řízeno vestavěné zařízení.

Vnitřní implementace switch bloku může být provedena jednoduše, ale neefektivně pomocí jednoduchých if/else bloků (série porovnání a skoků), ale také pomocí sofistikovanějších algoritmů, například převedení na tabulku skoků. V testu jsou záměrně volány případy pouze z druhé (vzdálenější od počátku) poloviny switch bloku, aby bylo porušeno rovnoměrné rozložení a znevýhodněna if/else implementace. Testuje se časová náročnost.

Tvorba nových objektů a jejich ukládání do seznamu

Podstatou testu `CreateObejctAndStoreInList` je změřit a porovnat paměťovou náročnost obou virtuálních strojů z hlediska uchovávání velkého mnžoství objektů v paměti. Test cyklicky „vyrábí“ nové objekty třídy `CO_Person` a umisťuje je do kolekce. Výstupem je jednoduché paměťové porovnání držení objektů v paměti a seznamu odkazů na ně.

Kapitola 7

Interpretace výsledků

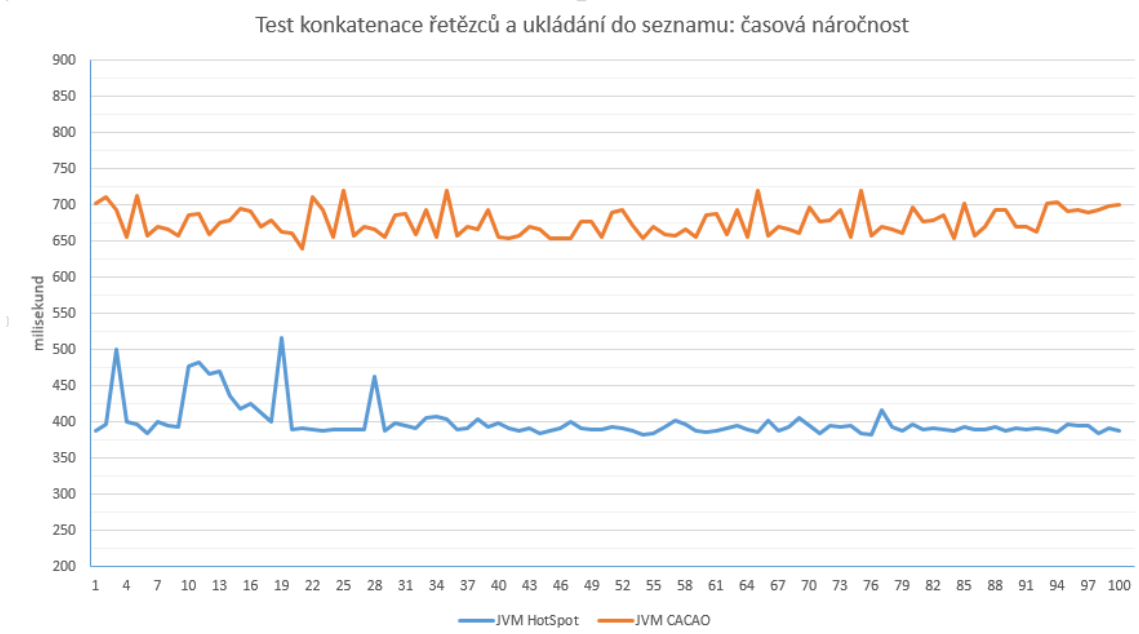
Ke statistickému porovnání byl zvolen aritmetický průměr hodnot vypočtený ze získaných dat pro každý test a stroj jako hlavní hodnota. Hodnoty mediánů se blíží aritmetickému průměru, což dokazuje vhodně zvolenou statistickou veličinu. Vyšší hodnoty směrodatných odchylek mohou způsobovat nahodilé hodnoty vzdálené od průměru, které však vzhledem k počtu opakování každého testu (100) jsou zanedbatelné. Kompletní výpočty lze najít na přiloženém CD v souboru `benchmark-results.xlsx`.

7.1 Konkatenace řetězců a uložení do seznamu

Test měřil základní výkonnost stroje prováděním běžné operace konkatenace a ukládání výstupu této operace do kolekce typu `ArrayList`. JVM HotSpot dosahoval výrazně lepších výsledků než JVM CACAO. V porovnání minimální naměřená hodnota JVM HotSpot a maximální hodnota JVM CACAO jde dokonce téměř o dvojnásobnou časovou náročnost v neprospěch CACAO. Průměrně byl JVM HotSpot o 40 % rychlejší. V grafu 7.1 je vidět, že JVM HotSpot po počátečních kolísavých výsledcích dával celkově výsledky s mnohem menší odchylkou (přestože celková směrodatná odchylka je vyšší, jak lze vidět v tabulce 7.1).

	JVM HotSpot	JVM CACAO
Průměr	399,93	676,77
Medián	392,00	670,00
Odchylka	24,63	19,08
Minimum	382	640
Maximum	517	721

Tabulka 7.1: Výsledky testu konkatenace řetězců a uložení do seznamu, hodnoty jsou v milisekundách



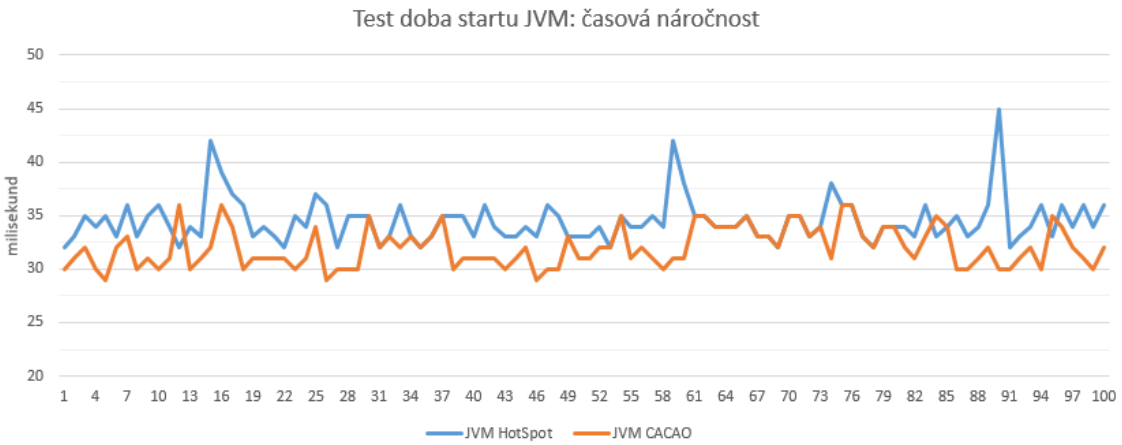
Obrázek 7.1: Graf výsledných hodnot všech běhů testu konkatenace řetězců a uložení do seznamu

7.2 Doba startu JVM

Velmi důležitým aspektem je doba startu virtuálního stroje – především v případě mobilních aplikací, kde cílem není program nechat běžet dlouhodobě, ale naopak je program potřeba právě ve chvíli spuštění. Oba stroje dosahovaly velmi podobných výsledků. Stroj JVM HotSpot více kolísal, výsledky dávají větší směrodatnou odchylku i větší rozdíl maximální a minimální doby startu. JVM HotSpot provádí po startu analýzu bajtkódu, což způsobuje jeho delší čas potřebný ke spuštění. JVM Cacao dosahoval pouze o 8 % lepších výsledků než JVM HotSpot, což vzhledem k faktu, že v ostatních testech dosahuje výrazně lepších výsledků JVM HotSpot, je téměř zanedbatelná hodnota. Úvodní analýza kódu za účelem pozdějších optimalizací je tedy velmi výhodná.

	JVM HotSpot	JVM CACAO
Průměr	34,50	32,00
Medián	34,00	32,00
Odchylka	2,10	1,84
Minimum	32	29
Maximum	45	36

Tabulka 7.2: Výsledky testu doba startu JVM, hodnoty jsou v milisekundách



Obrázek 7.2: Graf výsledných hodnot všech běhů testu doba startu JVM

7.3 Volání anonymní funkce a řazení pole

Časová náročnost

V porovnání časové náročnosti opakovaného volání anonymní funkce, která řadí pole řetězců, dosahuje dle očekávání lepších výsledků stroj HotSpot (relativně o 15 % proti stroji CACAO). JVM HotSpot metodu jednou zkompiluje do nativního kódu, který si uchová v paměti a ten pak pravidelně volá. Dalším testováním s jinými parametry lze zjistit, že pokud budeme zvyšovat počet iterací cyklu, ve kterém se tato funkce volá, tak se budou „nůžky“ paměťové náročnosti výrazně rozevírat, kdy časová náročnost u stroje CACAO se bude zvyšovat mnohem rychleji než u stroje HotSpot.

JVM HotSpot	JVM CACAO	
Průměr	115,67	134,89
Medián	114,00	136,00
Odchylka	5,74	10,38
Minimum	109	112
Maximum	160	163

Tabulka 7.3: Výsledky testu volání anonymní funkce a řazení pole, hodnoty jsou v milisekundách

Paměťová náročnost

Výsledky testu paměťové náročnosti obou strojů jsou shodné (liší se jen o 1 kB, což činí relativní rozdíl méně než 0,01 %, který je pod hranicí statistické významnosti). Využití paměti obou strojů bylo během všech iterací konstantní, kód neobsahuje žádný nedeterminismus.

Je třeba dodat, že podobnost výsledků je náhodná, v dalších testech již tento jev nenastává.

	JVM HotSpot	JVM CACAO
Průměr	13 935	13 934
Medián	13 935	13 934
Odchylka	0	0
Minimum	13 935	13 934
Maximum	13 935	13 934

Tabulka 7.4: Výsledky testu volání anonymní funkce a řazení pole, hodnoty jsou v kilobajtech

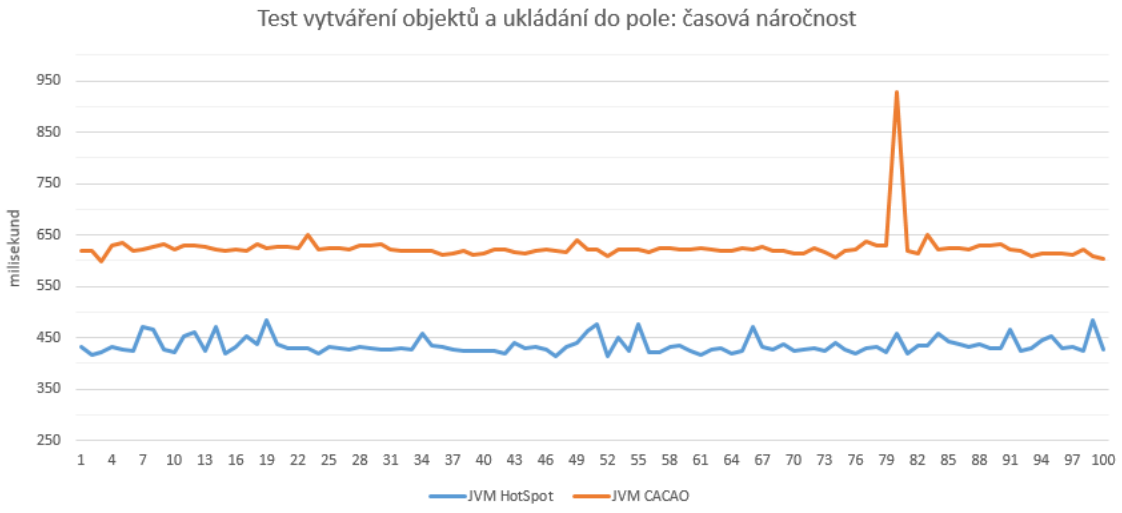
7.4 Vytváření objektů a ukládání do pole

Časová náročnost

Test opakovaně vytváří stejný objekt a ukládá ho do pole. Ve výsledcích lze vidět dobrou optimalizaci stroje HotSpot pro opakované operace. JVM HotSpot je v tomto případě o 30 % rychlejší než JVM CACAO. Z tabulky 7.5 je patrná téměř dvojnásobná směrodatná odchylka stroje JVM CACAO proti stroji JVM HotSpot – v grafu 7.3 je však vidět, že je to způsobeno pouze jedním vrcholem v grafu. Oba stroje poskytují relativně stabilní výsledky.

	JVM HotSpot	JVM CACAO
Průměr	434,77	624,87
Medián	429,00	621,00
Odchylka	15,95	31,48
Minimum	413	598
Maximum	485	928

Tabulka 7.5: Výsledky testu vytváření objektů a ukládání do pole, hodnoty jsou v milisekundách

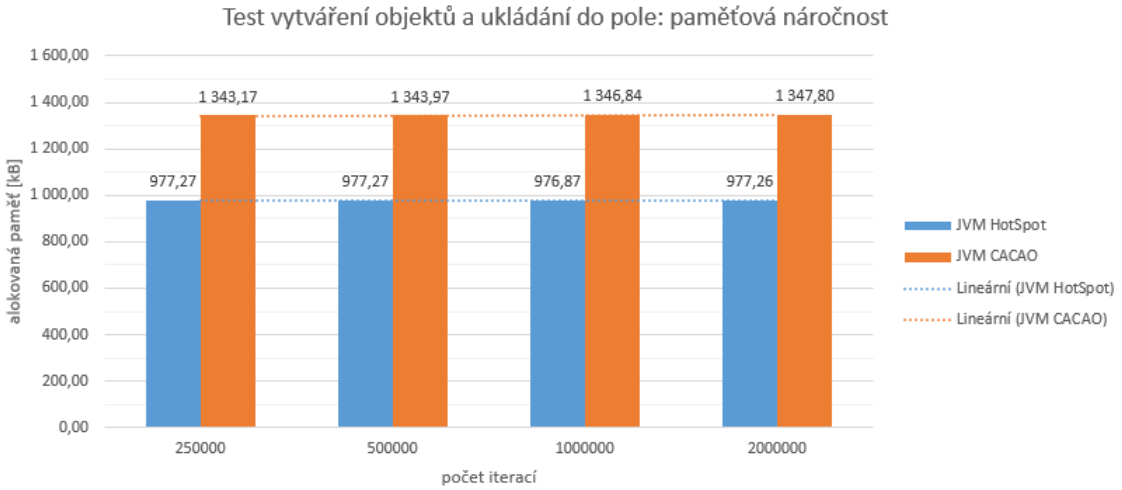


Obrázek 7.3: Graf výsledných hodnot všech běhů testu vytváření objektů a ukládání do pole (paměťová náročnost)

Paměťová náročnost

Benchmark vytváření objektů a ukládání do pole byl na každém stroji spuštěn 4krát pro různý počet iterací hlavního cyklu (250 tisíc, 500 tisíc, 1 milion a 2 miliony). V hlavním cyklu se opakovaně vytvářeli nové objekty, které se umisťovaly do pole (na prvních 1000 pozic), tedy s každou iterací od tisíce nahoru jeden objekt přestal být dosažitelný. Cílem bylo prověřit garbage collector obou strojů.

Výsledky tohoto benchmarku shrnuje graf 7.4. Graf zobrazuje průměrné hodnoty alokované paměti (odchylka mezi jednotlivými běhy u obou strojů byla pod hranicí statistické významnosti). Lze vidět, že paměťová náročnost je pro jednotlivé stroje (téměř) konstantní, z toho vyplývá, že oba garbage collectory jsou implementovány správně a alokovaná paměť nedosažitelných objektů je uvolňována. Celková alokovaná paměť (správně) závisí jen na velikosti použitých datových typů a nezávisí na délce běhu programu. Tato vlastnost je mimořádně důležitá pro dlouhodobě běžící serverové aplikace. Mimo tuto důležitou vlastnost, kterou oba stroje splňují, měl JVM HotSpot o 27 % menší paměťovou náročnost než stroj CACAO.



Obrázek 7.4: Graf výsledných hodnot všech běhů testu vytváření objektů a ukládání do pole (časová náročnost)

7.5 Velký switch blok

Oba stroje mají efektivně implementovaný switch blok pomocí tabulky skoků, přesto JVM HotSpot díky celkově lepším optimalizacím dosahuje o 20 % lepších výsledků.

	JVM HotSpot	JVM CACAO
Průměr	43,06	53,55
Medián	43,00	55,00
Odchylka	2,14	3,33
Minimum	39	40
Maximum	49	61

Tabulka 7.6: Výsledky testu velký switch blok, hodnoty jsou v milisekundách

7.6 Tvorba nových objektů a jejich ukládání do seznamu

Test prověřoval paměťovou náročnost uchovávání velkého množství objektů v paměti (milion objektů třídy `CO_Person` o dvou atributech (jméno typu `integer` a věk typu `string`)).

JVM HotSpot celkově alokoval o 3,5 % paměti více než JVM CACAO.

	JVM HotSpot	JVM CACAO
Průměr	205	198
Medián	205	198
Odchylka	0	0
Minimum	205	198
Maximum	205	198

Tabulka 7.7: Výsledky testu tvorba nových objektů a jejich ukládání do seznamu, hodnoty jsou v megabajtech

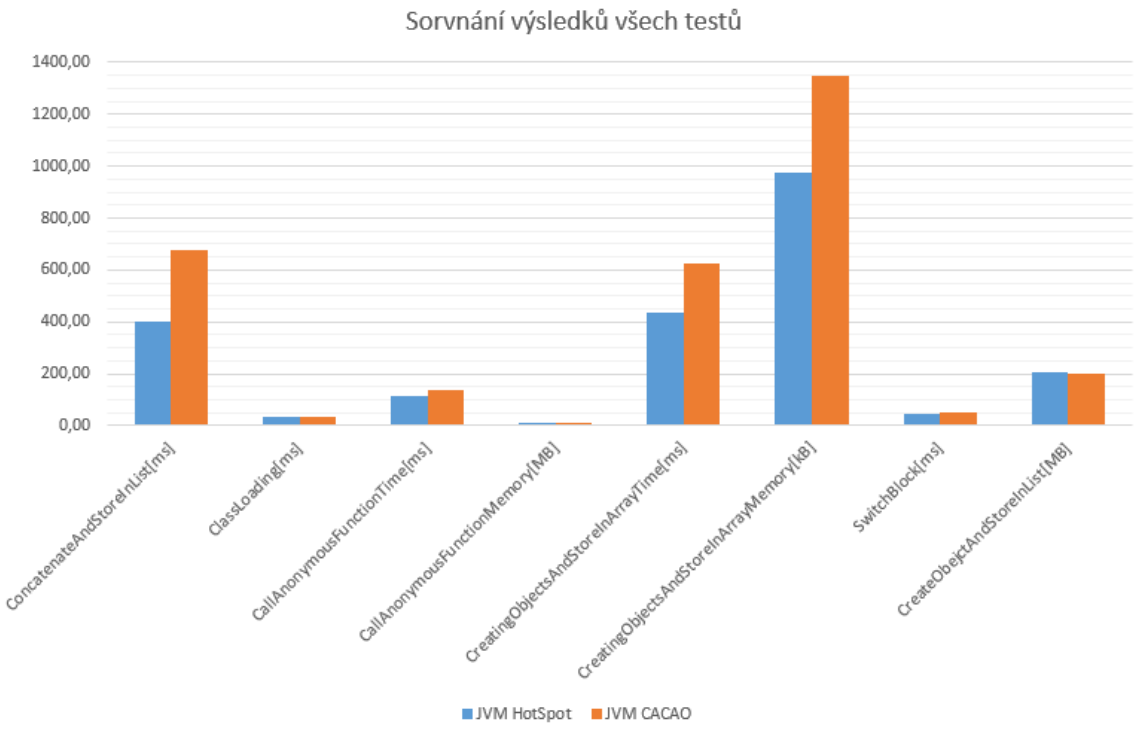
7.7 Shrnutí výsledků benchmarků

Tabulka 7.8 ukazuje, o kolik procent byl v každém testu vítězný stroj lepší. JVM HotSpot skončil výrazně lépe než JVM CACAO v 5 testech. JVM CACAO skončil jen nepatrně lépe než JVM HotSpot ve dvou testech. Jeden test poskytl totožné výsledky pro oba stroje.

JVM CACAO poskytl o něco lepší výsledky v oblasti paměťové náročnosti a době startu virtuálního stroje. Ne však vždy, v jednom testu paměťové nárčonsti výrazně lépe skončil JVM HotSpot, což bylo způsobeno pravděpodobně lepší implementací garbage collectoru, která dokáže dříve/častěji smazat nedostupné objekty. Paměť na serverech a desktopech však v dnešní době již většinou netvoří nedostupnou komoditu, a tak nasazením JVM HotSpot lze získat výrazné zrychlení běhu programů za drobné navýšení paměťové náročnosti v určitých případech.

	JVM HotSpot	JVM CACAO
Konkatenace řetězců a uložení do seznamu (čas)	o 40 %	
Doba startu JVM (čas)		o 8 %
Volání anonymní funkce a řazení pole (čas)	o 15 %	
Volání anonymní funkce a řazení pole (paměť)	vyrovnaný	vyrovnaný
Vytváření objektů a ukládání do pole (čas)	o 30 %	
Vytváření objektů a ukládání do pole (paměť)	o 27 %	
Velký switch blok (čas)	o 20 %	
Tvorba nových objektů a ukládání do seznamu (paměť)		o 3,5 %

Tabulka 7.8: Shrnutí výsledků jednotlivých benchmarků: relativní srovnání v procentech (o kolik procent lepšího výsledku daný stroj dosáhl proti méně výkonnému stroji)



Obrázek 7.5: Graf výsledných hodnot všech testů, nižší hodnota znamená lepší výkon

Kapitola 8

Závěr

Tato práce nastínila problematiku virtuálních strojů a popsala platformu Java. Platforma Java obecně přináší výhodné vlastnosti, které u standardně kompilovaných jazyků jako C/C++ nebyly uspokojivě vyřešeny. Díky pokročilým optimalizačním technikám dokáže být běh programu v Javě v určitých případech dokonce rychlejší než běh stejného programu v jazyce C++[15].

Byla popsána implementace vybraných prvků dvou virtuálních strojů, JVM HotSpot a JVM CACAO. Byl vytvořen soubor osmi testů, které prakticky porovnávaly oba stroje ve vybraných oblastech paměťové a časové náročnosti.

Stroj JVM CACAO byl vytvořen s cílem nabídnout rychlejší alternativu k tehdejším virtuálním strojům (navíc s menší paměťovou náročností). Jak popisuje kapitola 7.7, v současné době JVM CACAO zůstává výkonnostně daleko za JVM HotSpot. JVM HotSpot dosahoval obecně velmi dobrých výsledků, řádově o desítky procent. Výhodou JVM CACAO zůstává jeho menší velikost na disku, která umožňuje instalaci i na zařízení s velmi omezenou pamětí a o něco širší seznam podporovaných platforem[1]. Nevýhodou mimo pomalejší interpretaci kódu je nedostupnost pro nejnovější verzi platformy Java.

Vývoj stroje JVM CACAO stále pokračuje a v plánu jsou poměrně zásadní věci (přepsání garbage collectoru, upravení class loadingu v souladu se specifikacemi a další). Dle mého názoru JVM CACAO nemůže soupeřit s JVM HotSpot na zařízeních s dostatkem systémových prostředků, ale může ho spolehlivě a efektivně nahradit tam, kde JVM HotSpot není nebo nemůže být dostupný. Existují však i další open-source implementace Java virtuálních strojů jako Caffe, Zulu, Jam VM, Sable VM a další. JVM CACAO by tedy zasloužilo porovnat s těmito virtuálními stroji.

[1] Jako osobní poznámku si dovolím dodat, že JVM CACAO bych osobně nezvolil mimo jiné už jen kvůli problémům se sprovozněním. Program, u kterého potřebuji čtvrtý stroj a šestý operační systém, abych ho zprovoznil u mě nebudí důvěru. Navíc manuál je dlouhodobě nedostupný a návody na sestavení JVM CACAO velmi skromné. Ačkoliv se projekt stále rozvíjí a je „velmi živý“, poslední commit je z konce dubna 2017, tak pokud bych do projektu přispíval, zaměřil bych se na umožnění hladké instalace/kompilace a sepsal, jaké prerekvizity jsou ke kompilaci a spuštění JVM CACAO potřeba. Oficialní manuál totiž doslova říká, že jsou potřeba některé závislosti, které jednoduše zjistíme, až nám kompilace vyhodí chybu. A já tak po dvacáté spouštím kompilaci, abych se dozvěděl, co dál mi chybí...

Literatura

[1] *CACAO Wiki*. CACAOVM - Verein zur Förderung der freien virtuellen Maschine CACAO, [Online; navštíveno 6. 4. 2017].
URL http://c1.complang.tuwien.ac.at/cacaowiki/

[2] *Java HotSpot Virtual Machine Performance Enhancements*. ORACLE, [Online; navštíveno 15. 4. 2017].
URL https://docs.oracle.com/javase/8/docs/technotes/guides/vm/performance-enhancements-7.html

[3] *Java Platform Standard Edition 8 Documentation*. ORACLE, [Online; navštíveno 22. 2. 2017].
URL http://docs.oracle.com/javase/8/docs

[4] *Official home of the CACAO Java Virtual Machine*. CACAOVM - Verein zur Förderung der freien virtuellen Maschine CACAO, [Online; navštíveno 6. 4. 2017].
URL http://www.cacaojvm.org

[5] *The Java HotSpot Performance Engine Architecture*. ORACLE, [Online; navštíveno 29. 3. 2017].
URL http://www.oracle.com/technetwork/java/whitepaper-135217.html

[6] *Memory Management in the Java HotSpot Virtual Machine*. Sun Microsystems, 2006, [Online; navštíveno 2. 4. 2017].
URL http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf

[7] Campbell, B.; Iyer, S.; Akbal-Delibas, B.: *Introduction to Compiler Construction in a Java World*. 1. vydání, Chapman and Hall/CRC, 2012, ISBN 978-1-43-986088-5.

[8] Cass, S.: *The 2016 Top Programming Languages*. IEEE Spectrum, Červenec 2016, [Online; navštíveno 2. 4. 2017].
URL http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages

[9] Daly, J.: *A Brief History of Computer Programming Languages [Infographic]*. EdTech: Focus on Higher Education, Duben 2013, [Online; navštíveno 6. 4. 2017].
URL http://www.edtechmagazine.com/higher/article/2013/04/brief-history-computer-programming-languages-infographic

[10] Gosling, J.; Joy, B.; Steele, G. L.; aj.: *The Java Language Specification*. 2. vydání, Addison Wesley, 2000, ISBN 978-0-20-131008-5.

[11] Hwang, K.; Dongarra, J.; Fox, G. C.: *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things*. 1. vydání, Morgan Kaufmann, 2013, ISBN 978-0-12-800204-9.

[12] Krasner, G.: *Smalltalk-80 : bits of history, words of advice*. Xerox Palo Alto Research Center, 1983, ISBN 0-201-11669-3.

[13] Lindholm, T.; Yellin, F.; Bracha, G.; aj.: *The Java Virtual Machine Specification, Java SE 8 edition*. Oracle America, Inc., 2015, [Online; navštíveno 22. 2. 2017].
URL https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf

[14] Popek, G. J.; Goldberg, R. P.: Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM*, ročník 17, č. 7, Červenec 1974: s. 412–421, ISSN 0001-0782.
URL http://doi.acm.org/10.1145/361011.361073

[15] Purdy, C.: *When Is Java Faster Than C++?* Forbes, [Online; navštíveno 6. 5. 2017].
URL https://www.forbes.com/sites/quora/2015/05/26/when-is-java-faster-than-c

[16] Rosenblum, M.; Garfinkel, T.: *Virtual Machine Monitors: Current Technology and Future Trends*. *IEEE Computer Magazine*, ročník 38, č. 5, Květen 2005: s. 39–47.

Přílohy

Příloha A

Obsah CD

- Dokumentace
  - bp-nikolaj-malik-2017.pdf (technická dokumentace – bakalářská práce v pdf, včetně zadání)
  - src (zdrojové soubory dokumentace, obsahuje makefile pro překlad)
- Benchmarky
  - benchmark-results.xlsx (soubor s výpočty výsledků)
  - results-all
    - cacao (složka s výsledky testování JVM CACAO)
    - hotspot (složka s výsledky testování JVM HotSpot)
  - test-suite (složka se sadou benchmarků)
    - readme.txt (instrukce pro spuštění sady benchmarků)
    - run.sh (skript pro překlad a spuštění všech benchmarků)
    - src (zdrojové soubory s benchmarky)
    - target (připravená složka pro přeložené benchmarky)
    - results (připravená složka pro výsledky benchmarků)