VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

VIRTUÁLNÍ STROJ JAVY JAMVM NA ARCHITEKTUŘE ARM

JAMVM VIRTUAL MACHINE ON THE ARM ARCHITECTURE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **FILIP POBOŘIL**
AUTHOR

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

BRNO 2017

Zadání bakalářské práce/18466/2016/xpobor02

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Pobořil Filip**
Obor: Informační technologie
Téma: **Virtuální stroj Javy JamVM na architektuře ARM**
JamVM Virtual Machine on the ARM Architecture
Kategorie: Operační systémy

Pokyny:

1. Prostudujte vlastnosti alternativního virtuálního stroje Javy s názvem JamVM. Zaměřte se na způsob interpretace bajtkódu v JamVM
2. Porovnejte výhody a zápory JVM JamVM v porovnání s JVM HotSpot.
3. Přeložte a nainstalujte JDK/JRE s podporou JamVM na Raspberry PI 2 či na podobném zařízení s mikroprocesorem ARMv7 a minimálně 0,5 GB RAM.
4. Vytvořte demonstrační webovou aplikaci využívající servlet kontejner (Jetty či Tomcat), která bude sledovat stav vybraných GPIO či A/D převodníků, ukládat poslední stavy a na žádost klienta je posílat ve vybraném formátu (HTML, JSON)
5. Srovnejte systémové nároky a doby odezvy aplikace spuštěné nejprve v JVM JamVM a poté v HotSpot JVM.
6. Zhodnoťte dosažené výsledky a navrhněte možnosti dalšího vývoje projektu.

Literatura:

- Furber, S. ARM: System-on-chip architecture, 2nd edition, Addison-Wesley, 2000.
- Bill Venners. Inside the Java Virtual Machine, 2000. ISBN 0-07-135093-4
- JamVM - kompaktní virtuální stroj: http://jamvm.sourceforge.net. Dostupné říjen 2016.

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

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Tato práce popisuje a srovnává dva virtuální stroje Javy na architektuře ARM, jmenovitě JamVM a HotSpot na Raspberry Pi. V první části práce obecně popisuje Javu a princip virtuálního stroje. Další část se podrobněji zaměřuje na oba virtuální stroje a jsou v ní popsány jejich vlastnosti a způsob interpretace. V poslední části je popsána webová aplikace, která byla použita ke srovnávání virtuálních strojů, a vyhodnocení výsledků.

Abstract

This thesis describes and compares two Java virtual machines on the ARM architecture, namely JamVM and HotSpot on Raspberry Pi. First part of this thesis describes Java and virtual machine principle. Next part focuses on both virtual machines, describes their features and methods of interpretation. Last part describes web application, which was used to compare both virtual machines, and evaluation of results.

Klíčová slova

Java, virtuální stroj, HotSpot, JamVM, Jetty, Raspberry Pi, srovnání

Keywords

Java, virtual machine, HotSpot, JamVM, Jetty, Raspberry Pi, comparison

Citace

POBOŘIL, Filip. *Virtuální stroj Javy JamVM na architektuře ARM*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Kočí Radek.

Virtuální stroj Javy JamVM na architektuře ARM

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Radka Kočího Ph.D. Další informace mi poskytl Ing. Pavel Tišnovský Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Filip Pobořil
16. května 2017

Poděkování

Děkuji panu Ing. Radku Kočímu a panu Ing. Pavlu Tišnovskému za poskytnutí rad a za podporu při zpracování této bakalářské práce.

Obsah

Kapitola 1

Úvod

Java je objektově orientovaný programovací jazyk a platforma sloužící ke spouštění a vývoji programů v jazyce Java. Zdrojové kódy v jazyce Java je před spuštěním nejprve zkompilovat do takzvaného Java bajtkódu, který je poté možné interpretovat. K interpretaci bajtkódu se využívá Java virtuální stroj (JVM), díky čemuž je možné programy spouštět na jakékoliv platformě, kde je dostupný virtuální stroj. Virtuální stroj se skládá z několika logicky oddělených částí. Nejdůležitější částí je interpret, který se stará o provádění bajtkódu programu a určuje tedy výsledný výkon programů, které jsou jím vykonávány. Další důležitou částí je správce paměti, který se stará o vytváření a odstraňování objektů z paměti. Virtuální stroj, včetně jeho částí a bajtkódu, který provádí, je v této práci podrobněji popsán ve 2. kapitole.

V další části práce, kapitola 3, jsou popsány dva virtuální stroje, které jsou dostupné pro architekturu ARM. Prvním je JamVM vyvíjený Robertem Lougherem, který cílí na embedded zařízení s méně obvyklými procesorovými architekturami a omezeným množstvím hardwarových prostředků. Druhým virtuálním strojem je HotSpot od firmy Oracle, který slouží zároveň jako referenční implementace, ale také se jedná o velmi výkonný virtuální stroj. U obou virtuálních strojů jsou popsány jejich specifické vlastnosti a cíle. Práce se především zaměřuje na způsob interpretace a optimalizace, které jsou v nich implementovány.

Ve 4. kapitole, jsou oba virtuální stroje srovnány na Raspberry Pi 2. Pro srovnání byla místo běžných benchmarků zvolena aplikace, která by odpovídala reálnému využití Raspberry Pi a to jako teploměru s jednoduchým webovým rozhraním a exportem naměřených hodnot. Na aplikaci je srovnána doba odezvy a spotřeba paměti při spuštění pod oběma virtuálními stroji. Odezva aplikace je pak také měřena a srovnávána při více současných požadavcích, od jednoho až po 8, čímž se testuje jak virtuální stroj pracuje s více vlákny na vícejádrovém procesoru. Poslední 5 kapitola nakonec obsahuje shrnutí a vyhodnocení srovnání obou virtuálních strojů na testovací aplikaci.

Kapitola 2

Java

Platformu Java a programovací jazyk Java začala vyvíjet firma Sun Microsystems v roce 1990 jako alternativu k programovacím jazykům C a C++. Jazyk Java byl vyvíjen jako jednoduchý, objektově orientovaný se syntaxí podobnou jazykům C a C++. Při vývoji bylo především dbáno na to, aby jazyk byl robustní, bezpečný, nezávislý na architektuře a díky tomu i snadno přenositelný [23].

Výsledný jazyk je plně objektový s výjimkou osmi primitivních datových typů. Jazyk vůbec neobsahuje ukazatele nebo explicitní reference (jako C++), bezznaménkové číselné datové typy, příkaz `goto`, nebo jiné nízkoúrovňové konstrukce. Tyto konstrukce byly už při návrhu jazyka vynechány a to především důvodu častých chyb v programech způsobených jejich používáním. Místo používání ukazatelů platí, že při volání metod se primitivní datové typy předávají hodnotou a objekty odkazem. Jazyk navíc neumožňuje programátorovi spravovat paměť přímo a místo toho poskytuje automatickou správu paměti. Díky této správě nedochází ve výsledných programech k únikům paměti nebo k chybnému přístupu na již uvolněnou paměť. Automatická správa paměti kromě eliminace chyb způsobených špatnou správou paměti také programátorovi poskytuje určitý komfort při vývoji programů [22].

Zmíněné vlastnosti vedly k tomu, že se Java stala jedním z nepoužívanějších jazyků. Díky své přenositelnosti se používá pro programy, které mají pracovat na různých platformách bez nutnosti znovu program kompilovat nebo jakkoliv upravovat. Později byla platforma Java, díky svému úspěchu, rozdělena i na specifické edice, které jsou zaměřeny pro použití ve webových aplikacích nebo aplikacích pro embedded a mobilní zařízení. [22].

2.1 Bajtkód

Přenositelnost programů v Javě je zajištěna tak, že se místo překladu do strojového kódu určité architektury využívá interpretace. Problémem úplné interpretace je však nutnost při každém spuštění programu provádět úplnou analýzu zdrojového kódu. Výsledkem je sice plně přenositelný program, ale kvůli analýzám při každém spuštění je tento způsob interpretace poměrně pomalý. Z toho důvodu se vývojáři Javy rozhodli použít překlad do mezikódu – takzvaného Java bajtkódu. Díky tomuto přístupu analýzy zdrojových kódů stačí provést pouze jednou při kompilaci a výsledkem je *class* soubor s Java bajtkódem. Výsledný bajtkód je poté možné přímo interpretovat. Překladač zdrojových kódu do bajtkódu není součástí standardního běhového prostředí Javy, ale pouze vývojového prostředí [23].

Java bajtkód je uložen ve formě binárního souboru s příponou *class*, který lze spustit pomocí virtuálního stroje (podrobněji popsán v sekci 2.2). Každý *class* soubor má pevně

danou strukturu, která je přesně popsána ve specifikaci virtuálního stroje Javy. *Class* soubor smí obsahovat pouze jednu třídu, rozhraní nebo výčet. To znamená, že pokud kompilovaný zdrojový soubor obsahoval více tříd, třída obsahovala vnořené třídy nebo anonymní třídy, tak se při zkompilování vygeneruje více *class* souborů. *Class* soubory se stejným způsobem generují i pro rozhraní a výčty, z pohledu *class* souboru se jedná pouze o třídy s dodatečným příznakem [7].

Seznam položek obsažených v *class* souboru: [7]

Magická konstanta: 4 bajtová hodnota (hexadecimálně): `CA FE BA BE`

Verze class souboru: minoritní a majoritní verze formátu class souboru, slouží k určení zda je virtuální stroj schopen interpretovat daný class soubor

Constant Pool: tabulka symbolů a konstant (řetězce, čísla apod. zapsané přímo ve zdrojovém kódu)

Příznaky a viditelnost třídy: `interface`, `enum`, `static`, `public` aj.

this: odkaz na aktuální třídu

parent: odkaz na rodičovskou třídu

Rozhraní: počet a seznam rozhraní, které daná třída implementuje

Proměnné: počet všech proměnných třídy (i soukromé) a jejich seznam

Metody: počet všech metod třídy (včetně soukromých) a jejich seznam

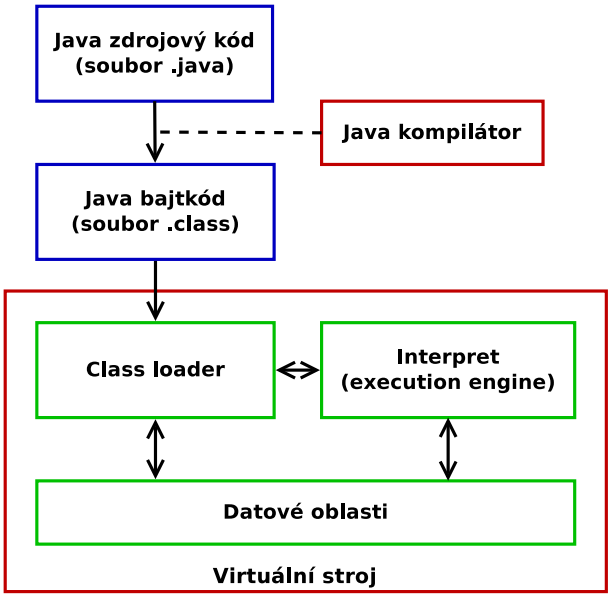
Atributy: dodatečné informace, například název zdrojového zdrojového java souboru

Class soubor také samozřejmě obsahuje i samotný kód metod třídy ve formě bajtkódu.

2.2 Virtuální stroj

Virtuální stroj Javy (JVM, Java Virtual Machine) je program, který slouží k vykonávání bajtkódu Javy. Díky překladu zdrojového kódu Javy do bajtkódu a jeho vykováváním pomocí virtuálního stroje je možné programy snadno přenášet na různé platformy, bez nutnosti jakýchkoliv úprav nebo opětovné kompilace zdrojového kódu. Virtuální stroj poskytuje podobné rozhraní jako fyzický stroj; registry, paměť a další a tím vytváří vrstvu mezi samotným programem a fyzickým strojem, na kterém je spuštěn. Programy v Javě pak díky tomu vůbec nemusí řešit jaké prostředky jim architektura fyzického stroje a operační systém poskytuje nebo jak s nimi pracovat. Tento přístup má pouze jedno omezení, a tím je existence virtuálního stroje na platformě, kde se má program spouštět [23].

Virtuální stroj je možné implementovat v libovolném jazyce, avšak kvůli požadavku na výkon jsou obvykle voleny jazyky C nebo C++. Kritické části, kde je obzvlášť kladen důraz na rychlost provádění, mohou být implementovány v jazyce symbolických instrukcí, neboli assembleru. Ke zvýšení rychlosti interpretace se také často používají různé optimalizace bajtkódu za běhu programu. Například JIT (Just-In-Time) kompilace, která je založena na dynamickém překladu často používaných částí bajtkódu do nativního strojového kódu a tím pak dochází k výraznému zvýšení rychlosti. Dalším požadavkem na virtuální stroj je snadná přenositelnost na jiné platformy. To může být také další důvod k použití C/C++, protože



Obrázek 2.1: Struktura virtuálního stroje

překladače C/C++ jsou dostupné na velkém množství různých platforem. Požadavek na přenositelnost může být částečně v rozporu s prvním požadavkem na rychlost, protože používání strojového kódu ke zvýšení rychlosti komplikuje přenos na jinou platformu. Části naprogramované v assembleru je pak nutné naprogramovat a udržovat pro každou platformu zvlášť.

2.2.1 Struktura JVM

Každý virtuální stroj se skládá ze tří základních, logicky oddělených, částí; *class loaderu*, datových oblastí a interpretu. Class loader se stará o načítání souborů s bajtkódem z disku nebo jiného média do běhových datových oblastí, které jsou umístěny v paměti počítače, a nakonec interpret provádí instrukce bajtkódu.

Class loader

Class loader, nebo-li zavaděč tříd, je část virtuálního stroje starající se o vyhledávání a načítání souborů bajtkódu za běhu programu. Zavaděče tříd jsou třídy organizované do hierarchické struktury se vztahem rodič-potomek. Kořenem této struktury je *bootstrap class loader*, který je vytvořen při spouštění virtuálního stroje a stará se o zavádění základních tříd Java API. Další v hierarchii je *extension class loader*, ten načítá různé třídy rozšiřující základní Java API. O načítání tříd aplikace, z adresářů definovaných uživatelem jako *CLASSPATH*, se stará *system class loader*. Zmíněné tři *class loader*y jsou standardní součástí virtuálního stroje a jsou nezbytné pro jeho správné fungování. [15]. Nejníže v hierarchii se nachází uživatelem definované zavaděče. Jedná se o běžné třídy, které mají jako rodiče třídu `ClassLoader`. Nejčastěji se používají k určení bezpečnostní domény, ale je možné je použít i k načítání tříd ze sítě nebo tříd vytvořených za běhu [11].

Zavaděče při přijetí požadavku na zavedení třídy nejprve prohledají vlastní cache a zjistí jestli stejný požadavek už zpracovávaly a třída je již načtena v paměti. Pokud třída ještě nebyla načtena, tak dotazovaný zavaděč neprovádí ihned načtení třídy, ale požadavky nej-

prve deleguje na svého rodiče. V rodiči probíhá stejný proces, pokud je třída načtena, tak se zavádění ukončí jako úspěšné, pokud ne, tak se deleguje na rodiče. Delegace probíhá u všech zavaděčů až ke kořenovému zavaděči, *bootstrap class loader*u, který žádného rodiče nemá. Pokud při delegaci žádný ze zavaděčů neměl třídu načtenou v paměti, tak zavaděč, který požadavek obdržel, načte třídu ze souborového systému nebo jiného média [15].

Datové oblasti

Datová oblast je část paměti, kterou virtuální stroj získal od operačního systému ve kterém je spuštěn. Datová oblast je rozdělena na několik částí podle použití: PC registr, zásobník, halda, oblast metod a běhový *constant pool* [15].

PC registr je vytvořen pro každé vlákno, které je v programu vytvořeno. Vytváří se okamžitě po spuštění vlákna a slouží k uložení adresy aktuálně prováděné instrukce bajtkódu. Zásobník je stejně jako PC registr v každém vláknu od doby jeho spuštění a je použit k uložení rámců metod. Na tento zásobník je při každém zavolání metody přidán rámec metody, který obsahuje lokální proměnné metody, zásobník operací pro uložení parametrů a návratových hodnot. Rámec navíc obsahuje odkaz na constant pool. Oblast metod je sdílena mezi všemi vlákny programu, obsahuje bajtkód metod všech tříd a statické proměnné tříd. Běhový *constant pool* má podobnou strukturu jako *constant pool* v class souborech. Obsahuje například odkaz na umístění metod a proměnných třídy v paměti. Kromě odkazů do paměti obsahuje také konstantní hodnoty z definic tříd, například `"Ahoj Světe"` z příkazu `System.out.println("Ahoj Světe")` ve zdrojovém kódu programu [7].

Ve virtuálním stroji je při spuštění vytvořena také halda, která slouží pro ukládání dat aplikace jako jsou objekty a pole. Halda je automaticky spravována *garbage collectorem*, který se stará o uvolňování nepoužívaných a nedostupných objektů. Nedostupné objekty jsou takové, na které není v programu žádná reference, což znamená, že neexistuje žádná proměnná ke které by byly přiřazeny. Explicitně nelze uvolňovat konkrétní objekty z haldy, lze pouze spustit kolekci nad celou haldou voláním `System.gc()`, což se ale provádí i automaticky pokud na haldě zbývá málo volného místa pro data. Specifikace virtuálního stroje neuvádí přesnou metodu správy haldy, tato část je ponechána na uvážení autora a jeho požadavcích na výsledný virtuální stroj [7]. Vzhledem k tomu, že operace vytváření a rušení objektů je častá, je způsob správy této části datové oblasti částečně zodpovědný za výkon virtuálního stroje. Správa paměti může ovlivnit celkovou rychlost virtuálního stroje a případně i způsobovat dočasné zastavení provádění programu [15].

Interpret

Interpret, nebo také *execution engine*, je ta část virtuálního stroje, která se stará o provádění samotného bajtkódu metod tříd. Tato část se chová podobně jako procesor, který provádí jednotlivé instrukce uložené v paměti. Instrukce bajtkódu se skládají z jednobajtového kódu operace (*OpCode*) a dodatečných operandů.

Podobu této jednotky nebo možné optimalizace bajtkódu za běhu interpretu specifikace neuvádí a konkrétní implementaci nechává čistě jen na uvážení a požadavcích autora. Specifikace pro interpret uvádí pouze seznam všech instrukcí a jejich sémantiku, kterou je nutné dodržet [7].

V této jednotce se může provádět jak jednoduchá interpretace, tak i složitější interpretace s optimalizací bajtkódu za běhu programu. Jednoduchou interpretací může být například použití konstrukce *switch* s návěstím *case* pro každou instrukci. V případě složitější interpretace s optimalizací bajtkódu může jít o Just-In-Time kompilátor, který za běhu

programu analyzuje bajtkód. Často používané části programu, nebo ty části kde je program časově nejnáročnější, nahradí nativním strojovým kódem, což jej zrychlí. Instrukce v interpretu mohou být z důvodu rychlosti implementovány v jazyce symbolických adres, tím se ale komplikuje přenos na jinou architekturu, protože je instrukce nutné implementovat a udržovat pro každou architekturu odděleně [10].

Kapitola 3

Srovnání JamVM a HotSpot VM

Tato kapitola teoreticky srovnává dva různé virtuální stroje Javy – HotSpot [12] od firmy Oracle a alternativní virtuální stroj JamVM [8]. U obou virtuálních strojů je nejprve popsán prvotní záměr, který stál za zahájením vývoje a pak také technologie a algoritmy použité při samotnému vývoji. Především jsou popsány metody interpretace a optimalizace, které virtuální stroje využívají k dosažení vyšších výkonů vykonávaných aplikací. Nakonec je rozebrána jejich metoda správy paměti, která může do značné míry ovlivňovat výkon virtuálního stroje a také jeho celkovou spotřebu fyzické paměti.

Oba virtuální stroje jsou určeny ke stejnému účelu, a tím je vykonávat programy v Javě. Jejich zaměření jsou ale odlišné. HotSpot je od začátku vyvíjen jako výkonný virtuální stroj, ale zároveň vyžaduje vyšší výkon počítače, na kterém je spuštěn. Vývojáři HotSpotu se také snaží o co nejlepší využití vícejádrových procesorů a efektivní správu velkého množství paměti RAM. JamVM naopak cílí na počítače s omezeným výpočetním výkonem a na embedded zařízení, kde jde především o efektivitu využití fyzických prostředků zařízení než o vysoký výkon. Značně rozdílný je také způsob vývoje a velikost týmu, který daný virtuální stroj vyvíjí. HotSpot je již od začátku vyvíjen velkou softwarovou společností, která disponuje zkušenými vývojáři a zároveň je určitou zárukou budoucího vývoje. JamVM je naproti tomu vyvíjen převážně jedním člověkem jako open source projekt s poměrně malou komunitou. Poslední stabilní verzí JamVM je 2.0.0, která byla vydána v červenci 2014 [8], od té doby okolo projektu není ze strany autora nebo komunity téměř žádná aktivita.

3.1 JamVM

JamVM je virtuální stroj Javy s otevřeným zdrojovým kódem vyvíjený Robertem Lougherem a malou komunitou okolo projektu. Ve srovnání s virtuálním strojem HotSpot je vyvíjen tak, aby byl co nejmenší a současně ale vyhovoval specifikaci virtuálního stroje verze 2 (blue book). Cílem JamVM jsou embedded zařízení, která často nedisponují velkým množstvím paměti RAM. Vyvíjen je tedy jako velmi malý, paměťově nenáročný, dobře přenositelný, ale i dostatečně výkonný virtuální stroj pro běžné použití. Podporované platformy jsou x86, x86_64, ARM, MIPS, PowerPC, PowerPC64, SPARC a HPPA. Nízká spotřeba paměti a vysoký výkon je do určité míry zajištěn použitím jazyka C a především mnoha optimalizacemi, ať už ve způsobu využití paměti a její správě, tak i v interpretaci bajtkódu. Snadná přenositelnost je zajištěna použitím pouze malého množství platformě závislého assembleru [8].

(a) *Switched* interpret

```
while (1) {
  switch (*pc++) {
    case ICONST_1:
      *sp++ = 1; break;
    case ICONST_2:
      *sp++ = 2; break;
    case IADD:
      --sp; sp[-1] += *sp; break;
    /* ... */
  }
}
```

(b) *Direct-threading* interpret

```
/* přechod na první instrukci */
goto **(pc++);

ICONST_1:
  *sp++ = 1; goto **(pc++);
ICONST_2:
  *sp++ = 2; goto **(pc++);
IADD:
  --sp; sp[-1] += *sp; goto **(pc++);

/* ... */
```

Obrázek 3.1: *Switched* a *threaded* interpret [6]

3.1.1 Interpret

Interpret v JamVM je implementován dvěma různými způsoby; *switched* interpret a *inline-threaded* interpret.

Switched interpret je základním a nejjednodušším typem interpretu, je postaven na základě konstrukce *switch*. Virtuální stroj nejprve načte bajtkód programu a uloží si jej v paměti jako pole. Poté interpret ve smyčce s konstrukcí *switch* prochází pole obsahující jednotlivé instrukce, které se postupně provádějí. V konstrukci *switch* jsou definovány větve *case* s implementací pro každou možnou instrukci bajtkódu. Tento typ interpretace je poměrně pomalý kvůli použití konstrukce *switch* a cyklu. Typicky jsou po zkompilování tohoto typu interpretu vygenerovány navíc tři strojové instrukce pro každou instrukci bajtkódu. Jedna instrukce pro skok na konstrukci cyklu, druhá pro porovnání instrukce, zda patří mezi *switch-case* hodnoty a třetí je skok na návěští *case*. Tyto tři strojové instrukce se musí provést pro každou instrukci bajtkódu, což je důvodem nízkého výkonu toho typu interpretu. V JamVM je tento typ interpretu implementován pouze z důvodu vyšší přenositelnosti, protože konstrukce *switch* je součástí ANSI standardu jazyka C [24, 6].

Direct-threading je typ interpretace, který ke své činnosti používá skoky na kód implementace instrukce. Skoky jsou v jazyce C realizovány příkazem *goto*. Před samotnou interpretací se z instrukcí bajtkódu nejprve vytvoří seznam skoků *Direct Threading Table*, který obsahuje kód instrukce (*OpCode*) převedený na adresu návěští (*label*) pro příkaz *goto*. Interpret při spuštění načte první instrukci a pomocí příkazu *goto* přejde na její implementaci. Přechod na další instrukci je umístěn na konci implementace každé instrukce. Jde se o stejný příkaz jako u přechodu na implementaci první instrukce. Tento typ interpretu, při srovnání se *switched* interpretem, redukuje počet instrukcí potřebných k přechodu na další instrukci bajtkód na jednu strojovou instrukci. *Threading* interpret je díky tomu rychlejší než *switching* interpret, ale vyžaduje nestandardní, GNU C, rozšíření překladače *Labels as Values* [24].

To, jestli JamVM bude využívat *switched* interpret nebo se bude používat metoda *direct-threading* se určuje při kompilaci na cílové platformě. Pokud překladač jazyka C na dané platformě podporuje nestandardní rozšíření *Labels as Values*, tak se použije rychlejší metoda interpretace *direct-threading*, pokud ne, tak se zvolí *switched* interpret. *Direct-threading* lze i manuálně vypnout dodatečnými parametry kompilace.

JamVM při interpretaci metodou *direct-threading* navíc využívá optimalizaci *inline-threading*. Stejně jako u *direct-threading* se používají skoky příkazem *goto*, ale eliminuje

(a) Implementace instrukcí

```
ICONST_1_start: *sp++ = 1;
ICONST_1_end: goto **(pc++);
INEG_start: sp[-1] = -sp[-1];
INEG_end: goto **(pc++);
DISPATCH_start: goto **(pc++);
DISPATCH_end: ;
```

(c) Super instrukce

```
ICONST_1_body: *sp++ = 1;
INEG_body: sp[-1] = -sp[-1];
DISPATCH_body: goto **(pc++);
```

(b) Implementace vytvoření super instrukce

```
size_t iconst_size = &&ICONST_1_end - &&ICONST_1_start;
size_t ineg_size = &&INEG_end - &&INEG_start;
size_t dispatch_size = &&DISPATCH_end - &&DISPATCH_start;

/* spojení instrukcí do jedné */
void *dynamic = malloc(iconst_size + ineg_size + dispatch_size);
memcpy(dynamic, &&ICONST_1_start, iconst_size);
memcpy(dynamic, &&INEG_start, ineg_size);
memcpy(dynamic, &&DISPATCH_start, dispatch_size);

/* provedení nově vygenerované instrukce */
goto **dynamic;
```

Obrázek 3.2: *Inline-threading*; blok instrukcí převedený na jednu super instrukci [6]

se režie potřebná k přechodu na další instrukci v blocích kódu. Základem techniky je vyhledání základních bloků bajtkódu, které obsahují větší množství jednoduchých instrukcí. Nalezené bloky instrukcí jsou pak nahrazeny jedinou instrukcí, takzvanou super instrukcí. Super instrukce jsou generovány dynamicky za běhu programu a jejich implementace je uložena v paměti. Super instrukce obsahují stejnou implementaci instrukcí jako běžné instrukce, ale tím, že byly sloučeny do jedné, není nutné provádět skoky mezi jednotlivými instrukcemi a tím je daná část programu zrychlena. Největší vliv na zrychlení je u jednoduchých instrukcí, u kterých by trval déle přesun na další instrukci, než provedení samotné instrukce. Ne všechny instrukce je ale možné převést do super instrukce. Instrukce, které obsahují volání funkcí jazyka C, volají skryté funkce (vygenerované překladačem) nebo i jednoduché instrukce podmíněných výrazů mohou zabránit v převedení bloku kódu na super instrukci [6].

Pro zrychlení provádění instrukcí je v JamVM navíc implementováno *stack-caching* – cacheování zásobníku. *Stack-caching* je založen na předpokladu, že přístup k hodnotám umístěným v registru procesoru je výrazně rychlejší, než přístup k hodnotám v paměti RAM. Umístěním vrcholu zásobníku do registrů procesoru je tak možné zrychlit provádění instrukcí, zbytek zásobníku je umístěn v paměti RAM [4]. Implementace jednotlivých instrukcí, při použití *stack-caching*, pak musí být provedena několikrát podle počtu registrů používaných pro cache. V JamVM se využívají dva registry pro cache, takže každá instrukce je implementována třemi způsoby a to pro případy kdy není žádná hodnota uložena v cache, s jednou hodnotou v cache a dvěma hodnotami v cache. Pokud není některý operand instrukce v cache, tak se načte ze zásobníku umístěného v paměti. Výsledky instrukcí se vždy ukládají do cache [20].

Díky implementovaným optimalizacím *inline-threading* a *stack-caching* je v interpretu JamVM dosaženo podobného výkonu jako v případě, kdy by byl k optimalizaci použit jednoduchý Just-In-Time kompilátor [8].

3.1.2 Správa paměti

V JamVM se používají dva algoritmy pro *garbage collector* a oba jsou implementovány jako *stop-the-world*, což znamená, že při spuštění úklidu paměti je pozastaven běh spuštěné aplikace. Samotná správa paměti je rozdělena na dvě fáze s různými metodami úklidu paměti, které se nepravidelně střídají – *mark and sweep* a *mark and compact* z důvodu minimalizace fragmentace haldy [9].

Mark and sweep je základní algoritmus používaný pro úklid paměti. Pracuje ve dvou fázích. V první fázi *mark* algoritmus rekurzivně vyhledává a označuje objekty, které jsou přímo nebo nepřímo dostupné v programu. Přímo dostupné objekty jsou ty, které jsou uloženy v lokálních a statických proměnných. Nepřímo dostupné objekty jsou objekty uložené v třídních proměnných některého dostupného objektu. Ve druhé fázi *sweep* se prochází všechny objekty na haldě a vyhledávají se ty, které nemají značku z předchozí fáze. Objekty bez značky, které nejsou z programu dostupné, se v této fázi uvolňují z paměti. Nevýhodou tohoto algoritmu je to, že způsobuje fragmentaci hlady [17].

Druhý implementovaný algoritmus *mark and compact* je modifikací algoritmu *mark and sweep*. V první fázi *mark* pracuje stejně jako algoritmus *mark and sweep* – označuje dostupné objekty. Ve druhé fázi *compact* dochází k přesunu objektů, které byly označeny v předchozí fázi, na jedno souvislé místo na haldě [16]. Tento algoritmus se využívá k defragmentaci haldy v případě, kdy fragmentace způsobená algoritmem *mark and sweep* dosáhla určité meze a začne být nezanedbatelným problémem. Fragmentace haldy by mohla způsobit výrazné zpomalení alokace nových objektů nebo způsobit řídké zaplnění haldy [9]. Při přesunu objektů je nutné změnit všechny reference na nové adresy objektů, k tomu JamVM využívá modifikovaný Jonkerův algoritmus [9].

Pro alokaci nových objektů se v JamVM využívá strategie *next-fit* [9]. Tato metoda pracuje tak, že pro nově alokovaný objekt vyhledá v paměti dostatečně velký volný prostor, kde bude možné objekt vytvořit. Vyhledávání takového místa začíná za posledním alokovaným objektem. Pokud se při hledání volného místa dojde až na konec paměti, tak vyhledávání začne od začátku paměti. V případě že se při vyhledávání dojde na stejné místo, kde hledání započalo, vytváření nového objektu skončí neúspěchem. Tato metoda je rychlejší než metoda *first-fit*, která hledá volné místo od začátku paměti nebo metoda *best-fit*, která hledá velikostně nejvhodnější volný prostor [2]. Nevýhodou této metody je fragmentace haldy [9].

Správa paměti se v JamVM provádí jednotně nad celou haldou a díky tomu virtuální stroj využije paměť lépe než HotSpot, ve kterém část haldy zůstává nevyužita kvůli kopírovacímu kolektoru.

3.1.3 Reprezentace objektu

Kvůli úspoře paměti jsou v JamVM zjednodušeny i hlavičky objektů. Virtuální stroje využívají hlavičky objektů pro uložení dodatečných informací o objektu, takzvaných metadat. Hlavička objektu v JamVM má délku dvě slova, obsahuje pouze zámek objektu a ukazatel na svou třídu [8].

Hlavička objektu může za určitých podmínek obsahovat i hash objektu. V JamVM je hash objektu založen na jeho adrese v paměti. Hash založený na adrese paměti má tu nevýhodu, že pokud dojde k přesunu objektu v paměti, hash se změní. Díky použití kolektoru

mark and compact, který objekty v paměti přesunuje, k této situaci může dojít. V JamVM je tento problém řešen tak, že je v hlavičce objektu uložen navíc stav hashe. Výchozím stavem, pro nové objekty, je *unhashed* – hash se nevyužívá, kolektor tedy může objekt bez problémů přesunout. Pokud byl v programu použit hash objektu, hodnota příznaku se změní na hodnotu *hashcode-taken*, to značí že se hash používá, ale kolektor ještě objekt nepřesunul, takže je stále možné používat aktuální adresu paměti. Když kolektor *mark and compact* přesunuje objekty, musí se kontrolovat hodnota tohoto příznaku. Pokud je příznak při přesunu objektu nastaven na hodnotu *hashcode-taken*, tak se jeho původní adresa musí uložit do dodatečné hlavičky a příznak se změní na *has-hashcode*. Objekt s příznakem *has-hashcode* pak jako hash používá obsah vytvořené hlavičky, který se už nemění a objekt je možné v paměti přesunout [9].

3.2 HotSpot

HotSpot je virtuální stroj vyvíjený firmou Oracle Corporation. Primárně je tento virtuální stroj zaměřen na použití na běžných počítačích a serverech, zároveň slouží jako referenční implementace. Je rozdělen na dvě různé implementace interpretu, klient a server, podle použití a nároků spouštěné aplikace. Jeho předností je jeho vysoký výkon díky použití Just-In-Time kompilátoru a adaptivní optimalizaci. Nevýhodou však je nepřenositelnost na jiné platformy kvůli použití velkého množství platformně závislého kódu. HotSpot je dostupný pouze architekturách x86, x86_64 a ARM. HotSpot má také poměrně vysoké nároky na množství paměti, což může být problém na zařízeních s malým množstvím dostupné paměti. Svůj název HotSpot získal díky použitému způsobu optimalizace, při kterém se vyhledávají a optimalizují takzvaná horká místa (*hot spots*), která jsou v programu často prováděna a mohou jeho běh zpomalovat [21].

Pro použití na více různých platformách vznikla speciální implementace HotSpotu nazvaná Zero. Zero HotSpot je stejně jako původní HotSpot implementován v C++ ale bez použití platformně závislého assembleru, díky tomu je možné jej přenést na jakoukoliv platformu, která disponuje překladačem C++. Použitím čistého C++, bez optimalizace v podobě použití assembleru, a vynecháním JIT došlo k výrazné ztrátě výkonu ve srovnání s původním HotSpotem [14]. V linuxových distribucích pro architekturu ARM bývá Zero HotSpot často použit jako výchozí virtuální stroj [21].

3.2.1 Interpret

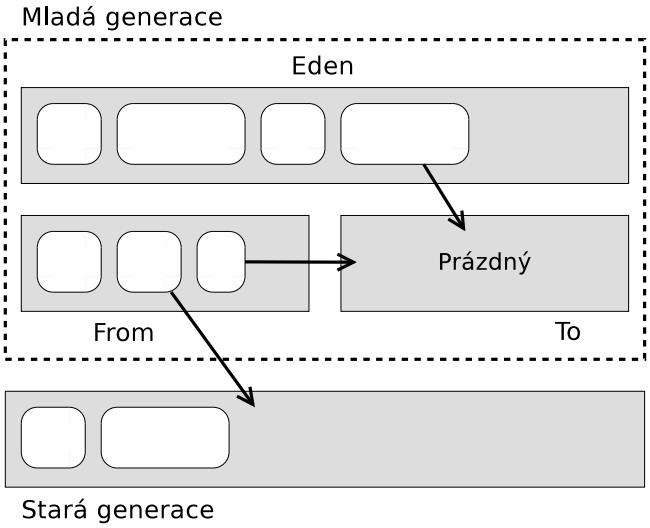
HotSpot používá šablonový interpret, který je založen na používání šablon kódu symbolických adres pro jednotlivé instrukce. Při spuštění se sestaví šablona přímo pro architekturu, na které je virtuální stroj spuštěn. Šablona je v paměti uložena jako instance `TemplateTable`, ve které je pro každou instrukci bajtkódu uložena jeho implementace. Interpretace se pak provádí pomocí skoků na předem vypočítané adresy paměti, kde je umístěn kód prováděné instrukce. Po provedení kódu instrukce se načte další instrukce a skočí se na její kód a tak dále. Tento typ interprety tedy funguje podobně jako *threading* interpret v JamVM [13].

Součástí interpretu v HotSpotu je také Just-In-Time kompilátor, který za běhu převádí bajtkód Javy do nativního strojového kódu. Implementovány jsou dva druhy; klient a server, které se liší ve způsobu a úrovni optimalizací. Klient je zaměřen na rychlejší provedení programu, tedy pro programy, které poběží relativně krátkou dobu. Server se spíše zaměřuje na rychlost vykonávaní programu, který v ideálním případě bude spuštěn nekonečně dlouhou

dobu, pomocí většího počtu optimalizací a jejich agresivnějšího používání [18]. Klient je tedy vhodnější pro programy, které běží kratší dobu protože četné optimalizace by měly za následek celkově delší dobu provádění. Server je naopak vhodnější programy, které poběží ideálně nekonečně dlouho a počáteční zpomalení prováděním optimalizací bude nakonec vynahrazeno vyšším výkonem programu.

3.2.2 Správa paměti

Správa paměti v HotSpotu je řešena rozdělením paměti na tři částí, takzvané generace. Objekty jsou pak rozděleny do různých generací podle délky jejich života. Pro uvolňování objektů v různých generacích se používají různé metody [19].



Obrázek 3.3: Struktura rozdělení paměti v HotSpotu, mladá a stará generace.

První generace se nazývá mladá generace. Tato generace je navíc rozdělena na další tři části; eden, první a druhý přeživší prostor. Nové objekty jsou alokovány na části eden. Při spuštění úklidu paměti se projde část eden a objekty, které jsou aktivní se přesunou do jedné ze dvou přeživších oblastí. Objekty v přeživších oblastech jsou pak spravovány kopírováním z jedné oblasti do druhé. Při úklidu paměti se tedy prochází pouze jeden přeživší prostor, *From*, a aktivní objekty se přesouvají do druhého, *To*. Do přeživšího prostoru *To* se při úklidu kopírují objekty také z části eden. Po skončení cyklu kolektoru je vždy jeden přeživší prostor prázdný a role přeživších prostorů *From* a *To* se vymění [19].

Další generace se nazývá stará generace. V této generaci se nacházejí objekty, které zůstaly aktivní i po určitém počtu cyklů kolektoru v mladé generaci. Objekty se v této oblasti mohou alokovat i přímo (nemusí být zkopírovány z mladé generace), ale pouze v případě, že v mladé generaci již není pro nové objekty místo. Kolektor je z důvodu existence vysokého počtu aktivních objektů optimalizován pro prostorovou efektivitu a rychlost je až na druhém místě. Ve staré generaci se využívá metoda *mark-sweep-compact*. Tato metoda nejprve vyhledá nedosažitelné objekty, poté je uvolní a nakonec přesune aktivní objekty na začátek oblasti, tím se získá na konci oblasti nepřerušovaný volný prostor [19].

Poslední generace je trvalá generace. Tato generace je určena pro objekty potřebné k běhu virtuálního stroje a nebudou tedy z paměti odstraňovány, jedná se objekty popisující třídy, metody, proměnné a také samotné metody s jejich bajtkódem [19].

Tento generační způsob správy paměti je velmi efektivní, protože není potřeba v jednom cyklu prohledávat celou paměť, ale stačí prohledat pouze její část. Na efektivitě přidává také to, že jsou objekty rozděleny podle délky jejich života a díku tomu lze zvolit optimální metodu úklidu těchto objektů. Nevýhodou tohoto přístupu je však to, že v mladé generaci se vždy nachází část nevyužité paměti – jeden přeživší prostor. Při malém množství dostupné paměti se pak může stát to, že se budou objekty kvůli nedostatku místa alokovat i na staré generaci, která není určena pro vytváření nových objektů a její správce paměti s mladými objekty nepočítá [19].

Ve srovnání s generační správou paměti v HotSpotu, JamVM spravuje paměť jako jeden celek. Objekty se tak mohou vytvářet kdekoliv, kde je pro ně místo, především ale není nutné udržovat jeden přeživší prostor, který zůstává v době běhu programu nevyužit. Nevyužitá část paměti může být problém na zařízeních s malým množstvím paměti [9].

Pro alokování nových objektů se používá technika *bump-the-pointer*, která pracuje podobně jako *next-fit* v JamVM. Zaznamenává se adresa posledního alokovaného objektu a nový objekt je alokován za posledním objektem [19].

V HotSpotu je vedle popsaného sériového kolektoru implementován také kolektor paralelní. Paralelní kolektor se používá na počítačích s velkým množstvím paměti a více procesory. Důvodem k jeho vývoji bylo lepší využití více procesorů než tomu je u sériové varianty, kdy je úklid paměti spuštěn pouze v jednom vlákně na jednom procesoru [19].

Kapitola 4

Testovací aplikace

Testování probíhalo na Raspberry Pi 2 Model B s 900 MHz čtyř jádrovým procesorem ARM Cortex-A7 a 1 GB paměti RAM. Jako systém byl použit Debian 8 (stable větev). JamVM byl nainstalován standardním způsobem z repozitáře Debianu stable a HotSpot byl stažen z oficiálních stránek Oracle. Oba virtuální stroje byly určeny pro Javu verze 7.

4.1 Návrh a implementace aplikace

Aplikace je navržena tak, aby odpovídala reálnému využití Raspberry Pi. Nejedná se tedy pouze o sadu benchmarků, ale o reálně využitelnou webovou aplikaci, která využívá servlet kontejner Jetty [3]. Sestavenou aplikaci jako archiv war by bylo možné nasadit i do jiného webového serveru s podporou Java servletů. Aplikace periodicky ukládá hodnoty z externího zařízení přes GPIO piny (General Purpose Input/Output, programovatelné vstupní/výstupní piny). Externí zařízení, použité jako zdroj dat, je analogově digitální převodník MCP3301, který je možné použít například k měření teploty nebo osvětlení, pro účely testování na měřené veličině nezáleží. Na požadavek uživatele aplikace reaguje zasláním naměřených hodnot jako HTML stránku pro zobrazení v prohlížeči a nebo exportem ve formátu JSON. HTML stránka je tvořena nečíslovaným seznamem, ve kterém jsou vypsány všechny uložené hodnoty s časem jejich zaznamenání. JSON (JavaScript Object Notation, formát pro výměnu dat) export obsahuje stejné informace, pouze v jiném formátu.

Po spuštění aplikace se inicializuje servlet kontejner Jetty a zahájí se sběr hodnot z vybraných GPIO pinů. Sběr hodnot probíhá v odděleném vlákně, nezávisle na požadavcích uživatelů. Komunikace s převodníkem probíhá přes SPI rozhraní a přenáší se 13 bitů dat (určeno typem převodníku) – naměřená hodnota napětí na vstupu převodníku. Hodnoty se ukládají do seznamu, který udržuje určitý omezený počet posledních hodnot s časem jejich uložení. Seznam je uložen pouze v paměti, takže po ukončení aplikace jsou data ztraceny. K ovládání GPIO pinů a vytvoření SPI rozhraní na straně Raspberry Pi se používá knihovna Pi4J. Pro generování HTML stránky se používá technologie *JavaServer Pages* (JSP). JSON export je generován knihovnou JSON.simple. Doba mezi jednotlivými čteními hodnot z převodníku i délka seznamu uložených hodnot lze nastavit při nasazování na server. V testovací aplikaci byla doba nastavena na 15 sekund a ukládalo se 60 posledních hodnot. Pro účely testování se při spuštění aplikace seznam předvyplnil náhodnými hodnotami, aby nebylo nutné čekat až se naměří hodnoty a naplní se jimi seznam.

4.2 Metoda testování

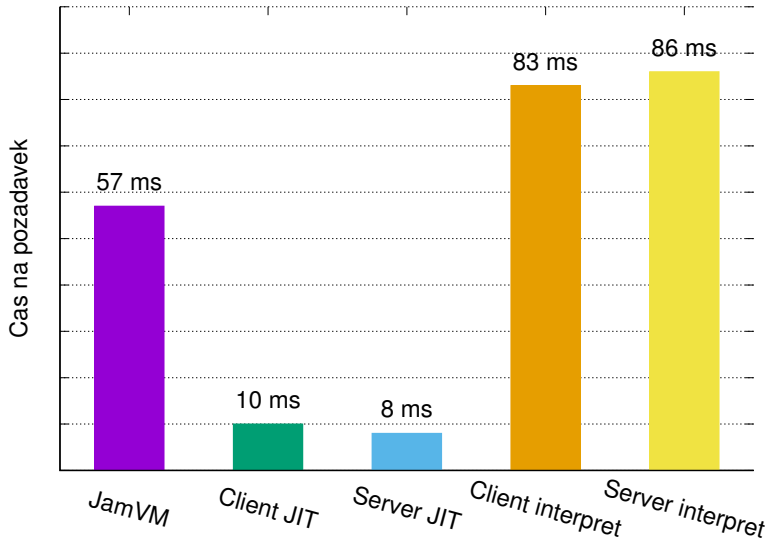
Při testování virtuálních strojů se měřila odezva aplikace, což je čas mezi odesláním požadavku a přijetím odpovědi, a spotřeba paměti. Pro měření času odezvy byl použit nástroj ApacheBench [1], kterým lze testovat libovolné webové servery. Tento nástroj pracuje tak, že odešle na webový server zadaný počet požadavků a vypočítá průměr, medián, minimum a maximum doby odezvy. Tento nástroj také umožňuje odesílat více požadavků současně a otestovat tak škálovatelnost webového serveru nebo aplikace. ApacheBench byl spouštěn z druhého počítače, aby jeho běh neovlivnil výsledky měření. Spotřeba paměti byla zaznamenávána v době běhu a po skončení ApacheBench ze souboru `/proc/$PID/status`.

Jednotlivé virtuální stroje byly srovnávány vícekrát s různým nastavením počtu současných požadavků od jednoho až po 8. Celkový počet požadavků při všech testech byl 10 000. Po každém 1 000. požadavku se zaznamenala spotřeba paměti procesu virtuálního stroje. Při každé změně počtu současných požadavků se aplikace ukončila a před spuštěním dalšího testu znovu spustila. Samotné virtuální stroje byly spouštěny bez jakýchkoliv parametrů nastavení VM (`-X` parametry), použily se tedy výchozí hodnoty nastavené vývojáři. Pouze v případě testování samotného interpretu HotSpotu bez JIT bylo nezbytné použit parametr `-Xint`.

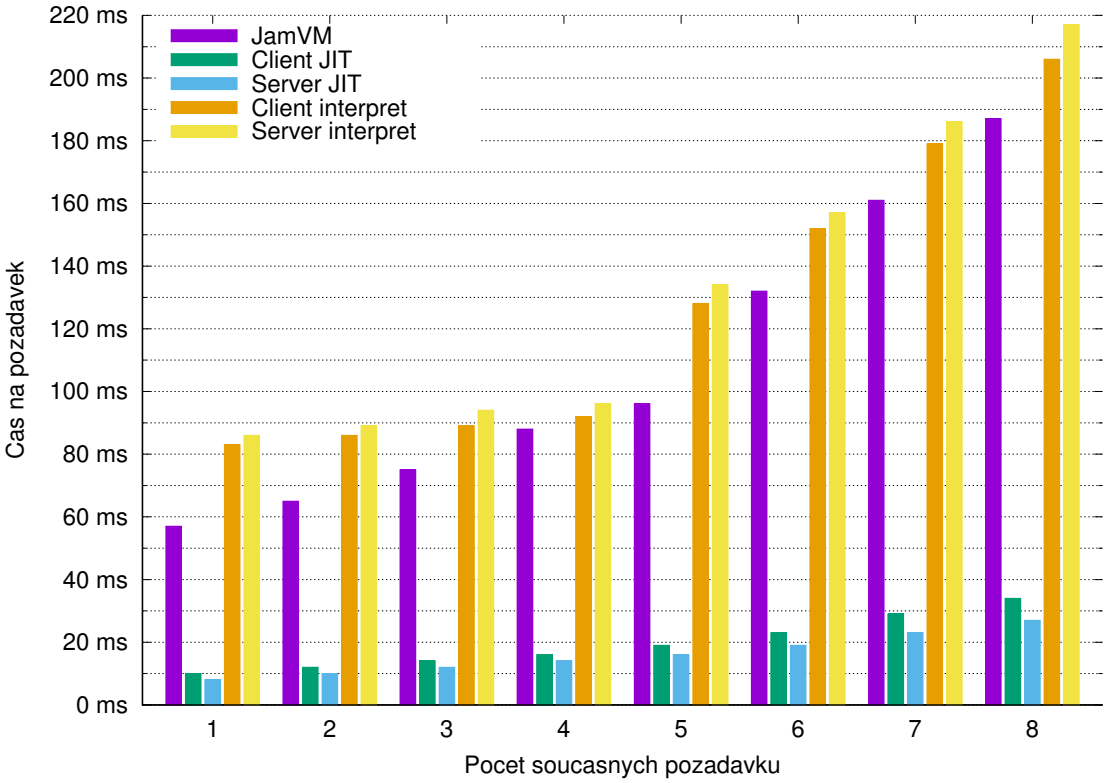
4.3 Srovnání

V prvním testu se měřil výkon aplikace na stránce s výpisem naměřených hodnot ve formátu HTML, pro zobrazení uživatelem. HTML stránka se generovala pomocí JSP, které zjednodušuje vytváření HTML obsahu v Javě. Z pohledu programátora je JSP jen obyčejný XML soubor, interně ale pracuje stejně jako *servlet*. Pokud se přijme požadavek na zobrazení stránky generované pomocí JSP, tak Servlet kontejner použije odpovídající servlet, pokud existuje. Pokud servlet pro JSP neexistuje, tak se vygeneruje z odpovídajícího JSP souboru. Jedná se o klasickou Java třídu, která se ale vytvoří, zkompiluje a uloží do class souboru až za běhu aplikace a po ukončení aplikace se smaže [5]. První požadavek na stránku JSP je tedy, relativně k dalším požadavkům, výrazně pomalejší. Z toho důvodu se při testování prvních 5 požadavků do vypočítávaných statistik nezapočítávalo, provedly se ještě před spuštěním ApacheBench.

Z grafu naměřených hodnot (obrázek 4.1) je zřejmé, že HotSpot s JIT je nejrychlejší, což se dalo předpokládat. Překlad do nativního kódu, který JIT v HotSpotu provádí, vede k vysoké rychlosti aplikací, které jsou pod ním spuštěny. Z JIT variant server a klient je rychlejší server, i když jen v řádech jednotek milisekund. Samotný interpret HotSpotu je ve srovnání s HotSpotem s JIT výrazně pomalejší, jak je vidět na grafu, je až 10× pomalejší. Zajímavé je srovnání samotných interpretů bez JIT, kde JamVM byl asi o 40% rychlejší než interpret HotSpotu. Tento výsledek svědčí o kvalitnějším návrhu interpretu a optimalizacích v JamVM.



Obrázek 4.1: Medián odezvy při jednom současném požadavku



Obrázek 4.2: Srovnání rychlosti odezvy při více současných požadavcích

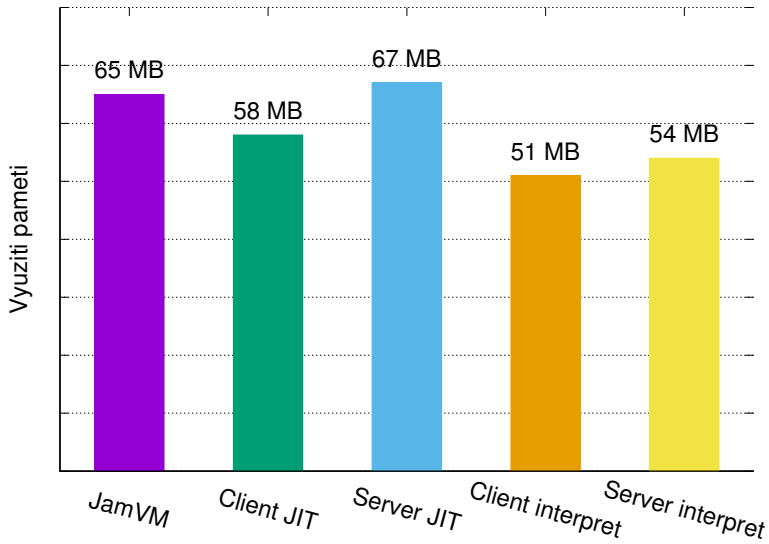
Při více současných požadavcích se doba odezvy jednotlivých požadavků u všech virtuálních strojů zvýšila (viz obrázek 4.2). Toto zpomalení je způsobeno tím, jak virtuální stroj pracuje s vlákny a také operačním systémem, který přiděluje procesorový čas ne jen vir-

současných požadavků	JamVM	Client JIT	Server JIT	Client interpret	Server interpret
1	57	10	8	83	86
2	65	12	10	86	89
3	75	14	12	89	94
4	88	16	14	92	96
5	96	19	16	128	134
6	132	23	19	152	157
7	161	29	23	179	186
8	187	34	27	206	217

Tabulka 4.1: Rychlosti odezvy (v milisekundách) při více současných požadavcích

tuálnímu stroji, ale také jiným programům. Pokud by aplikace nebo virtuální stroj nebyly schopné pracovat s vlákny, tento nárůst by byl vyšší než 100%.

Při každém zvýšení počtu současných požadavků je zpomalení pro JamVM a HotSpot s JIT přibližně 20%. U interpretu HotSpotu je toto zpomalení pouze několik jednotek milisekund, což je méně než 5% zpomalení. Do 4 současných požadavků sice rychlost zpracování jednotlivých požadavků klesá, ale za určitou časovou jednotku je požadavků zpracováno více, protože se díky vícejádrovému procesoru zpracovávají současně. Procesor Raspberry Pi má jen 4 jádra, takže při 5 a více současných požadavcích už nelze vlákna zpracovávající požadavky rozložit na jádra procesoru a zpracovávat je současně. Z grafu (obrázek 4.2) lze toto překročení počtu jader rozeznat výraznějším zvýšením doby odezvy. Nejvýraznější zpomalení je vidět u interpretu HotSpotu, u kterého také došlo ke snížení počtu zpracovaných požadavku za sekundu (viz tabulky v příloze A.1).



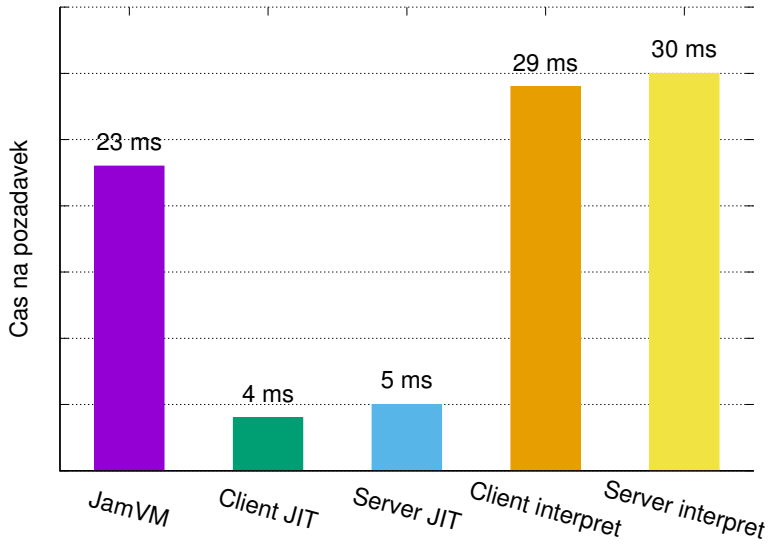
Obrázek 4.3: Srovnání spotřeby paměti

Z grafu se srovnáním spotřeby paměti (obrázek 4.3) lze vyčíst maximální množství paměti, kterou aplikace a virtuální stroj v průběhu testování spotřeboval. Nejvíce paměti spotřeboval HotSpot se server JIT, to se dalo předpokládat, protože provádí četné optimalizace a generuje nativní kód což vyžaduje poměrně velké množství paměti. Client JIT

provádí optimalizace méně než server, což má v tomto případě za následek, že je pomalejší, ale díky tomu má nižší spotřebu paměti. JamVM má spotřebu téměř stejnou jako server JIT, rozdíl mezi nimi je pouze 2 MB. Ve srovnání s interprety HotSpotu má JamVM výrazně vyšší spotřebu paměti, client interpret spotřeboval o 14 MB méně paměti, což je 20% rozdíl proti JamVM. Spotřeba paměti se v závislosti na počtu současných požadavků prakticky neměnila, rozdíly byly menší než 1 MB.

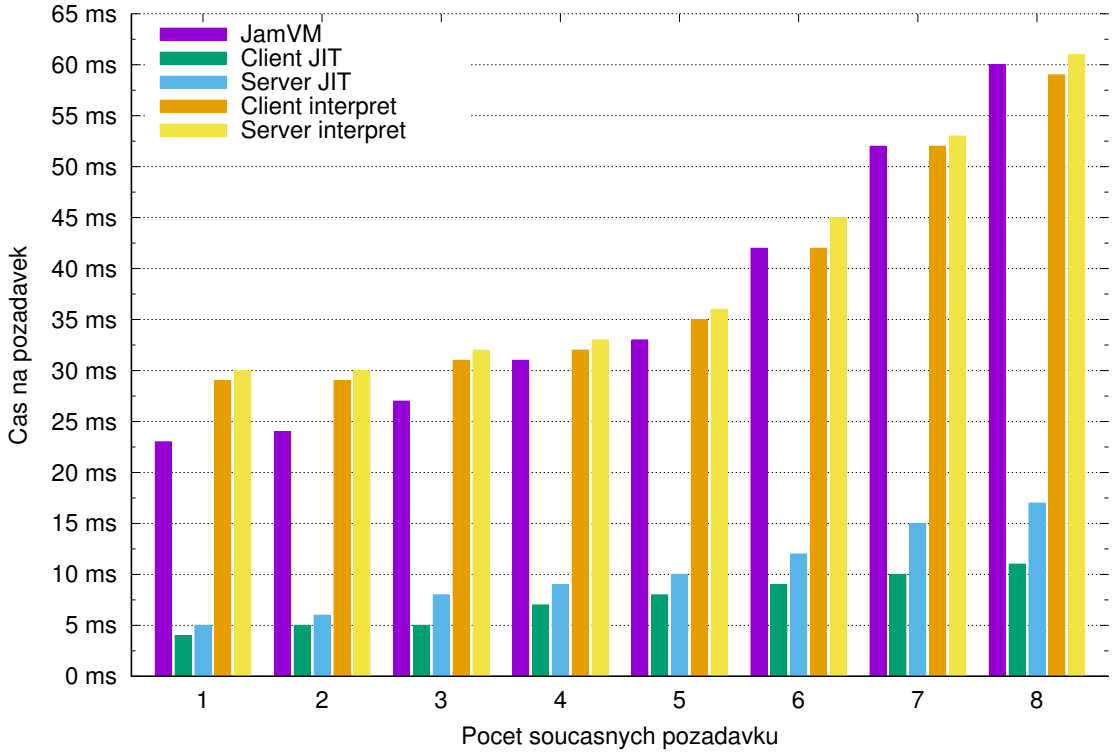
JSON export

JSON export je ve srovnání s HTML výpisem jednodušší a tedy i méně náročný na výkon. Servlet při požadavku pouze iteruje přes seznam naměřených hodnot a převádí je na objekty knihovny JSON.simple, které následně vkládá do speciální kolekce ze stejné knihovny. JSON se poté vytvoří jednoduše převedením objektu kolekce na řetězec metodou `toString` a odešle se jako odpověď. Po dokončení požadavku odesláním dat tyto objekty zanikají a mohou být *garbage collectorem* odstraněny z paměti. Tento JSON export tedy testuje hlavně schopnost virtuálního stroje rychle vytvářet větší množství objektů, které se používají pouze po krátkou dobu při zpracovávání požadavku. S objekty také souvisí *garbage collector*, který může ovlivňovat rychlost odezvy, především u JamVM protože je implementován jako *stop-the-world* (popsáno v sekci 3.1.2).



Obrázek 4.4: Medián odezvy při jednom současném požadavku

Z grafu rychlosti odezvy (obrázek 4.4) je vidět, že poměry mezi rychlostmi virtuálních strojů jsou podobné jako u předchozího testu. HotSpot s JIT je nejrychlejší, ale v tomto případě byla varianta JIT client překvapivě rychlejší než server. Pravděpodobně je to způsobeno tím, že client JIT je určený pro programy, které jsou obecně jednodušší a běží kratší dobu. Server JIT je naproti tomu určen pro náročnější aplikace, které poběží velmi dlouhou dobu a zpomalení způsobené agresivnější optimalizací bude zanedbatelné. V tomto testu se tedy u HotSpotu s JIT projevuje toto zpomalení na provádění optimalizací. Mezi interprety HotSpotu a JamVM je opět nezanedbatelný rozdíl v rychlosti ve prospěch JamVM a interpret HotSpotu je tedy nejpomalejší.



Obrázek 4.5: Srovnání rychlosti odezvy při více současných požadavcích

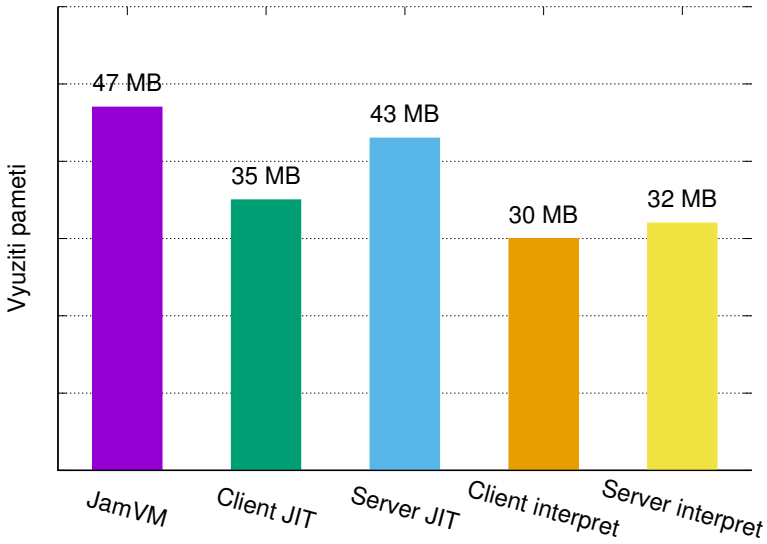
současných pož.	JamVM	Client JIT	Server JIT	Client int.	Server int.
1	23	4	5	29	30
2	24	5	6	29	30
3	27	5	8	31	32
4	31	7	9	32	33
5	33	8	10	35	36
6	42	9	12	42	45
7	52	10	15	52	53
8	60	11	17	59	61

Tabulka 4.2: Rychlosti odezvy (v milisekundách) při více současných požadavcích

Na grafu se srovnáním odezvy při více současných požadavcích (obrázek 4.2) se stejně jako v případě HTML výpisu postupně zvyšovala rychlost odezvy. Do 4 současných požadavků je toto zpomalování dostatečně malé, aby se za určitou časovou jednotku zpracovalo více požadavků. Z grafu je vidět, že více současných požadavků zvládaly nejlépe interprety, tedy JamVM a HotSpot interpret, doba odezvy u nich vzrůstala asi o 10%. U HotSpot JIT je toto zpomalení vyšší, asi 20%.

U 5 a více současných požadavků, virtuální stroje využily všechna dostupná procesorová jádra a některé požadavky tedy čekaly na zpracování, což je v grafu vidět nárůstem odezvy. Totéž se stalo i u předchozího testu. V tomto případě ale JamVM dosahovalo horších výsledků a odezva byla přibližně stejná jako u interpretu HotSpotu. Toto výraznější zpomalení může být způsobeno *garbage colletorem*, který je v JamVM implementován jako *stop-the-world*, což znamená, že v době úklidu paměti se nezpracovávaly žádné příchozí požadavky.

Při více současných požadavcích je v paměti více objektů, které se aktivně využívají, takže mohlo dojít k větší fragmentaci haldy a zpomalit tak její úklid.



Obrázek 4.6: Srovnání spotřeby paměti

V tomto testu měl nejvyšší spotřebu paměti JamVM, to je poměrně nečekané, protože JamVM by měl být určen pro zařízení s menším množstvím paměti. Raspberry Pi má ale 1 GB RAM a virtuální stroj s výchozím nastavením má limit velikosti haldy ¼ dostupné paměti, to je 250 MB. JamVM tedy i přes vyšší spotřebu paměti nedosáhlo stanového limitu. Při vyšší spotřebě paměti aplikací by se JamVM chovalo lépe, z pohledu využití paměti, což bylo vidět u předchozího testu. U HotSpotu s JIT měl client, díky menším množství prováděných optimalizací, výrazně nižší spotřebu paměti než server. HotSpot interpret měl spotřebu paměti nejnižší, protože neprovádí takové optimalizace jako JIT, ale také byl nejpomalejší. Při přidávání počtu současných požadavků se spotřeba paměti téměř neměnila, rozdíly byly menší než 1 MB.

4.4 Shrnutí

Z provedených měření na testovací aplikaci jasně vyplývá, že HotSpot s JIT je nejrychlejší, odezva aplikace byla výrazně nižší než u JamVM, které bylo až 7× pomalejší. Ve srovnání čistých interpretů, se ukázal kvalitnější návrh JamVM a JamVM tak byl nezanedbatelně rychlejší než interpret HotSpotu. V paměťové náročnosti byl HotSpot JIT a JamVM přibližně na stejné úrovni, v prvním, na paměť náročnějším, testu mezi nimi byl rozdíl pouze několika jednotek MB. Nejnižší spotřebu paměti měl interpret HotSpot, ale samozřejmě za cenu nízkého výkonu. Ve všech virtuálních strojích byla aplikace také dobře škálovaná a plně se využil vícejádrový procesor.

Kapitola 5

Závěr

V této práci byly rozebrány dva virtuální stroje, HotSpot a JamVM, dostupné na procesorové architektuře ARM, konkrétně na Raspberry Pi. V první části práce jsem shrnul obecné vlastnosti Javy a virtuálních strojů, které se používají ke spouštění programů v Javě. Dále jsou podrobněji rozebrány vlastnosti virtuálních strojů JamVM a HotSpotu. Popsány jsou jejich vlastnosti, které vývojáři od výsledných virtuálních strojů požadovali, správci paměti ale především byl popsán způsob interpretace bajtkódu a optimalizační techniky, které využívají. V další části byly tyto virtuální stroje otestovány na aplikaci, která využívá servlet kontejner a zobrazuje hodnoty z analogově digitálního převodníku.

Výsledky měření výkonu aplikace prokázaly, že virtuální stroj HotSpot s JIT (*Just-In-Time kompilátor*) dosahuje výrazně vyšších výkonů než JamVM. Vysoký výkon prováděných aplikací je cílem vývojářů HotSpotu už od samého počátku vývoje, takže tento výsledek potvrzuje dosažení jejich cíle. JamVM byl naproti tomu vyvíjen jako jednoduchý a ve srovnání s jinými virtuálními stroji malý, ale zároveň tak, aby vyhovoval specifikaci. Navržen byl tak, aby byl jeho výkon dostatečný pro běžné použití, ale především bylo při jeho návrhu důležité, aby byl snadno přenositelný na platformy, kde HotSpot nebyl dříve dostupný, pouze jako *Zero-assembler* port, který je velmi pomalý. Při srovnáním výkonu s HotSpotem s JIT je JamVM výrazně pomalejší, ale ve srovnání s čistým interpretem v HotSpotu byl nezanedbatelně rychlejší, což je v kombinaci s dobrou přenositelností velmi dobrý výsledek. Při srovnávání paměťové náročnosti dosahoval JamVM přibližně stejných výsledků jako HotSpot s JIT. Raspberry Pi, na kterém byly prováděny testy, disponuje poměrně velkým množstvím paměti, na platformě s omezenější pamětí by JamVM měl být díky svému způsobu správy paměti vhodnější než HotSpot. JamVM tedy splnil požadavky vývojáře na dostatečný výkon a snadnou přenositelnost. Použití JamVM je tak vhodné na platformách, kde není dostupný HotSpot nebo je dostupné pouze omezené množství paměti.

Práci by bylo možné rozšířit o testování dalších virtuálních strojů dostupných na platformě ARM, například CACAO nebo Kaffe, a porovnání jejich výkonu s JamVM a HotSpotem na stejné aplikaci.

Literatura

[1] Apache Software Foundation: ab – Apache HTTP server benchmarking tool. [Online; navštíveno 11. 05. 2017].
URL https://httpd.apache.org/docs/2.4/programs/ab.html

[2] Bays, C.: *A Comparison of Next-fit, First-fit, and Best-fit. Commun. ACM*, ročník 20, č. 3, Březen 1977: s. 191–192, ISSN 0001-0782, doi:10.1145/359436.359453.
URL http://doi.acm.org/10.1145/359436.359453

[3] Eclipse Foundation: Jetty - Servlet Engine and Http Server. [Online; navštíveno 11. 05. 2017].
URL http://www.eclipse.org/jetty/

[4] Ertl, M. A.: *Stack Caching for Interpreters. SIGPLAN Not.*, ročník 30, č. 6, Červen 1995: s. 315–327, ISSN 0362-1340, doi:10.1145/223428.207165.
URL http://doi.acm.org/10.1145/223428.207165

[5] Fields, D. K.; Kolb, M. A.: How JSP Works: Servlets and JavaServer Pages. 2001, [Online; navštíveno 21. 04. 2017].
URL https://people.apache.org/~jim/NewArchitect/webrevu/2001/02_02/developers/index03.html

[6] Gagon, E.; Hendren, L.: *Effective Inline-Threaded Interpretation of Java Bytecode...* 2003, [Online; navštíveno 19. 02. 2017].
URL http://www.sable.mcgill.ca/publications/papers/2003-2/sable-paper-2003-2.ps.gz

[7] Lindholm, T.; Yellin, F.; Bracha, G.; aj.: *The Java Virtual Machine Specification.* 2013, [Online; navštíveno 30. 01. 2017].
URL http://docs.oracle.com/javase/specs/jvms/se7/html/index.html

[8] Lougher, R.: *JamVM.* 2014, [Online; navštíveno 16. 02. 2017].
URL http://jamvm.sourceforge.net/

[9] Lougher, R.; Wielaard, M.: *Emailová komunikace mezi R. Lougherem a M. Wielaardem.* 2006, [Online; navštíveno 20. 02. 2017].
URL https://gcc.gnu.org/ml/java/2006-08/msg00021.html

[10] Myers, A.: *Interpreters, compilers, and the Java Virtual Machine.* [Online; navštíveno 15. 02. 2017].
URL
http://www.cs.cornell.edu/courses/cs2112/2012sp/lectures/lec27-12sp.pdf

[11] Oracle Corporation: *ClassLoader*. [Online; navštíveno 14. 02. 2017].
URL http://docs.oracle.com/javase/7/docs/api/java/lang/ClassLoader.html

[12] Oracle Corporation: *The HotSpot Group*. [Online; navštíveno 16. 02. 2017].
URL http://openjdk.java.net/groups/hotspot/

[13] Oracle Corporation: *HotSpot Runtime Overview*. 2016, [Online; navštíveno 20. 02. 2017].
URL http://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html

[14] Oracle Corporation: *Zero-Assembler Project*. 2016, [Online; navštíveno 16. 02. 2017].
URL http://openjdk.java.net/projects/zero/

[15] Park, S. H.: *Understanding JVM internals*. 2012, [Online; navštíveno 13. 02. 2017].
URL http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals

[16] Preiss, B. R.: *Mark-and-Compact Garbage Collection*. 1998, [Online; navštíveno 22. 02. 2017].
URL https://www.brpreiss.com/books/opus5/html/page428.html

[17] Preiss, B. R.: *Mark-and-Sweep Garbage Collection*. 1998, [Online; navštíveno 22. 02. 2017].
URL https://www.brpreiss.com/books/opus5/html/page424.html

[18] Simonis, V.: *OpenJDK JVM Internals*. 2012, [Online; navštíveno 19. 02. 2017].
URL http://www.progdoc.de/papers/Jax2012/jax2012.html

[19] Sun Microsystems: *Memory Management in the Java HotSpot Virtual Machine*. 2006, [Online; navštíveno 19. 02. 2017].
URL http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf

[20] Thomm, I.: *Implementation and Evaluation of Fast Untyped Memory in a Java Virtual Machine*. 2006.
URL https://www.researchgate.net/publication/265178470

[21] Wikipedia: *HotSpot — Wikipedia, The Free Encyclopedia*. 2017, [Online; navštíveno 16. 02. 2017].
URL https://en.wikipedia.org/w/index.php?title=HotSpot&oldid=765232042

[22] Wikipedia: *Java (programming language) — Wikipedia, The Free Encyclopedia*. 2017, [Online; navštíveno 29. 01. 2017].
URL https://en.wikipedia.org/w/index.php?title=Java_(programming_language)&oldid=762300256

[23] Wikipedia: *Java (software platform) — Wikipedia, The Free Encyclopedia*. 2017, [Online; navštíveno 29. 01. 2017].
URL https://en.wikipedia.org/w/index.php?title=Java_(software_platform)&oldid=763308624

[24] Zaleski, M.: *Dispatch Techniques*. 2008, [Online; navštíveno 19. 02. 2017].
URL http://www.cs.toronto.edu/~matz/dissertation/matzDissertation-latex2html/node6.html

Přílohy

Příloha A

Naměřené hodnoty

A.1 HTML export

součass. pož.	1	2	3	4	5	6	7	8
průměr	63.5	79.7	97.8	117.2	139.8	167.1	195.1	222.1
sm. odchylka	30	42.8	52.6	61.6	74.1	81.6	83.8	86.4
minimum	11	12	14	15	14	23	36	59
maximum	519	541	599	693	744	822	721	855
medián	57	65	75	88	96	132	161	187
1. kvartil	57	64	73	81	85	101	127	153
3. kvartil	57	67	87	112	167	236	271	300
požadavků/s	15.7	25.1	30.7	34.1	35.8	35.9	35.9	36

Tabulka A.1: JamVM

součass. pož	1	2	3	4	5	6	7	8
průměr	11.1	14.2	17.2	20.3	23.5	28.6	32	36.5
sm. odchylka	5.4	21.2	10.8	18.4	15	32.5	18	18.9
minimum	6	7	10	11	11	9	12	12
maximum	457	1404	480	525	496	976	480	484
medián	10	12	14	16	19	23	29	34
1. kvartil	10	11	12	13	15	17	21	25
3. kvartil	11	13	20	24	30	35	39	44
požadavků/s	89.7	141.3	174.7	197	212.9	209.5	219	219.4

Tabulka A.2: Client JIT

současn. pož	1	2	3	4	5	6	7	8
průměr	9.3	11.8	15	17.8	20.9	25.1	29.2	32.3
sm. odchylka	5.8	10.8	26.5	14.5	17	20	25.9	23.5
minimum	7	7	7	7	7	8	9	9
maximum	460	474	1403	478	489	492	573	483
medián	8	10	12	14	16	19	23	27
1. kvartil	8	9	10	11	13	15	17	20
3. kvartil	9	11	16	20	25	30	34	38
požadavků/s	107.7	170	199.4	224.6	239	238.7	239.5	247.9

Tabulka A.3: Server JIT

současn. pož	1	2	3	4	5	6	7	8
průměr	83.7	87.3	94.6	96.4	131.1	155.3	182.6	207.2
sm. odchylka	5.7	12.8	14	24.9	28.4	38	53.5	41.3
minimum	13	14	14	13	16	25	24	48
maximum	482	592	516	1075	544	625	1219	678
medián	83	86	89	92	128	152	179	206
1. kvartil	83	85	88	90	110	128	153	181
3. kvartil	84	87	98	100	149	178	207	233
požadavků/s	11.9	22.9	31.7	41.5	38.1	38.6	38.3	38.6

Tabulka A.4: Client interpret

současn. pož	1	2	3	4	5	6	7	8
průměr	86.4	90.5	99	100.1	136.4	160.2	187.6	219.7
sm. odchylka	9.8	11.3	16.1	13.4	29.6	37.8	41.6	64
minimum	13	13	15	14	23	20	29	58
maximum	859	585	559	558	590	600	647	1684
medián	86	89	94	96	134	157	186	217
1. kvartil	85	88	92	94	115	133	159	190
3. kvartil	86	90	102	104	155	184	214	244
požadavků/s	11.6	22.1	30.3	40	36.7	37.4	37.3	36.4

Tabulka A.5: Server interpret

A.2 JSON export

současn. pož	1	2	3	4	5	6	7	8
průměr	23.4	26.3	31.1	35.6	42	49.6	57.9	66.5
sm. odchylka	6	8.9	11.7	13.9	18.1	20.3	21.4	22
minimum	9	8	12	23	13	17	28	27
maximum	99	99	116	120	138	156	170	169
medián	23	24	27	31	33	42	52	60
1. kvartil	22	24	26	28	31	36	43	52
3. kvartil	23	25	31	36	50	58	66	74
požadavků/s	42.8	76.1	96.4	112.4	119.2	121	120.8	120.3

Tabulka A.6: JamVM

současn. pož	1	2	3	4	5	6	7	8
průměr	4.7	5.8	7.1	8.8	10	11.4	13	16.3
sm. odchylka	2	3.1	4.2	10.7	6.2	7.2	7.9	39.1
minimum	4	4	4	4	4	4	4	4
maximum	26	37	57	476	87	81	65	989
medián	4	5	5	7	8	9	10	11
1. kvartil	4	4	5	5	6	6	7	8
3. kvartil	4	6	7	9	11	14	17	20
požadavků/s	214.7	343.1	420.6	455.8	501.8	526.3	540.5	489.6

Tabulka A.7: Client JIT

současn. pož	1	2	3	4	5	6	7	8
průměr	5.9	7.6	9.1	11.2	13.1	16	17.6	21.7
sm. odchylka	1.8	15.2	4.7	6.5	8.3	32.3	10.8	54
minimum	4	4	4	4	5	5	5	5
maximum	39	937	78	92	126	1286	113	1897
medián	5	6	8	9	10	12	15	17
1. kvartil	5	6	6	7	8	9	10	12
3. kvartil	6	8	10	13	15	18	22	25
požadavků/s	170.1	263.3	329.7	358.2	382.9	375.9	397.4	368.5

Tabulka A.8: Server JIT

současn. pož	1	2	3	4	5	6	7	8
průměr	29.6	30.2	33.1	34.7	39.3	46.3	54.7	61.5
sm. odchylka	6	4.2	6.3	20.2	11.5	13.6	18.9	14.1
minimum	8	7	15	10	15	12	17	27
maximum	446	78	97	1010	111	126	567	146
medián	29	29	31	32	35	42	52	59
1. kvartil	28	28	29	30	31	36	44	52
3. kvartil	29	30	34	36	44	54	62	69
požadavků/s	33.8	66.3	90.6	115.2	127.2	129.5	128.1	130.1

Tabulka A.9: Client interpret

současn. pož	1	2	3	4	5	6	7	8
průměr	30.3	31.3	34.7	35.5	41	49.6	56.2	64.2
sm. odchylka	8.9	3.9	6.5	6.6	11.8	35.5	14.8	14.5
minimum	8	12	11	9	10	24	13	28
maximum	655	70	78	103	122	1439	158	159
medián	30	30	32	33	36	45	53	61
1. kvartil	29	30	31	31	33	38	45	54
3. kvartil	30	31	36	37	46	58	65	72
požadavků/s	33	63.8	86.5	112.6	121.8	121.1	124.5	124.6

Tabulka A.10: Server interpret