VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

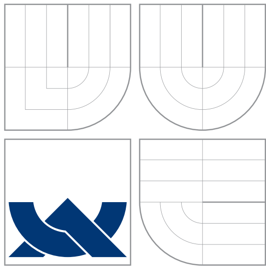
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

JAMVM: ALTERNATIVNÍ VIRTUÁLNÍ STROJ JAVY

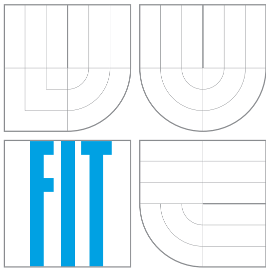
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — ONDŘEJ KALMAN
AUTHOR

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

JAMVM: ALTERNATIVNÍ VIRTUÁLNÍ STROJ JAVY

JAMVM: AN ALTERNATIVE JAVA VIRTUAL MACHINE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ONDŘEJ KALMAN

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. VENDULA HRUBÁ

BRNO 2014

Abstrakt
Cílem této práce je srovnání dvou virtuálních strojů Javy a to virtuálního stroje HotSpot a virtuálního stroje JamVM. Úvodní část práce čtenáře seznamuje s platformou Java, jsou zde také shrnuty obecné vlastnosti a principy funkčnosti všech virtuálních strojů Javy. Na tento teoretický úvod poté navazuje kapitola, ve které jsou srovnány klíčové vlastnosti virtuálních strojů HotSpot a JamVM. Na základě těchto vlastností je poté navržena a popsána sada výkonnostních testů, které byly v rámci této práce implementovány a provedeny. Poslední kapitoly této práce jsou věnovány prezentaci výsledků provedených výkonnostních testů a jejich vyhodnocení. Výsledky testů prokázaly, že virtuální stroj HotSpot výkonnostně převyšuje konkurenční JamVM, který naopak více šetří systémové prostředky a rychleji startuje.

Abstract
This thesis deals with comparison of two virtual machines, namely HotSpot and JamVM. The first chapters of this thesis contain introduction to Java platform and summarize general properties and principals of Java virtual machine. The next chapters follow on this introduction and compare different properties of HotSpot and JamVM virtual machines. Based on these differences was designed and described set of benchmark tests which was also implemented and performed as practical part of this thesis. Last chapters deals with results of performed benchmark tests. Results showed that Hotspot virtual machine has bigger computing power than JamVM, however JamVM is more efficient with system resources and starts faster.

Klíčová slova
Java, virtuální stroj, interpret, optimalizace, bajt kód, výkonnostní testy

Keywords
Java, virtual machine, interpreter, optimization, byte code, benchmark tests

Citace
Ondřej Kalman: JamVM: Alternativní virtuální stroj Javy, bakalářská práce, Brno, FIT VUT v Brně, 2014

JamVM: Alternativní virtuální stroj Javy

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením paní Ing. Venduly Hrubé.

. . . .
Ondřej Kalman
14. května 2014

Poděkování

Děkuji paní Ing. Vendule Hrubé a panu Ing. Pavlu Tišnovskému Ph.D. za poskytnutí cenných rad a za podporu při zpracování této bakalářské práce.

© Ondřej Kalman, 2014.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Teoretický úvod do platformy Java **4**
2.1 Historie a návrh 4
2.2 Bajtkód 4
2.2.1 Binární soubory tříd .class 5
2.3 Běhové a vývojové prostředí Javy 7
2.3.1 Virtuální stroj 8
2.3.2 Java API 8
2.3.3 Specifikace virtuálního stroje 9
2.4 Shrnutí 9

3 Srovnání virtuálních strojů HotSpot a JamVM **10**
3.1 Použité technologie 10
3.2 Optimalizace výkonu 11
3.3 Návrh výkonnostních testů 12
3.3.1 Cyklení a práce s polem 12
3.3.2 Práce s objekty a constant poolem 12
3.3.3 Invokování metod 13
3.3.4 Rychlost rekurze 13
3.3.5 Práce s řetězci 13
3.3.6 Vyhodnocování podmínek 13
3.3.7 Rychlost startu virtuálního stroje 14
3.3.8 Výpočet v plovoucí desetinné čárce 14
3.3.9 Velikost virtuálního stroje v operační paměti 14
3.3.10 Práce s haldou 14
3.4 Shrnutí 15

4 Výkonnostní testy **16**
4.1 Řazení pole algoritmem bubble-sort 16
4.2 Řazení pole algorimem quick-sort 19
4.3 Výpočet čísla π 21
4.4 Zřetězení stringů 22
4.5 Test podmínek 24
4.6 Rychlost rekurze 25
4.7 Invokace metod 26
4.8 Start virtuálního stroje 28
4.9 Velikost virtuálního stroje 29

4.10 Práce s haldou . . . 29
4.11 Shrnutí . . . 31

5 Závěr **32**

A Naměřené hodnoty **34**
A.1 Výsledky testů pro algoritmy bubble-sort . . . 34
A.2 Výsledky testů pro algoritmus quick-sort . . . 50
A.3 Výsledky testů výpočtu π . . . 55
A.4 Výsledky testů zřetězení stringů . . . 57
A.5 Výsledky testů rychlosti provádení podmínek . . . 64
A.6 Výsledky testů rychlosti rekurze . . . 64
A.7 Výsledky testů invokací metod . . . 66
A.8 Výsledky testu pro měření rychlosti strartu virtuálního stroje . . . 66
A.9 Výsledky měření velikosti virtuálního stroje . . . 67
A.10 Výsledky monitorování heapu . . . 67

Kapitola 1

Úvod

Java je objektově orientovaným programovacím jazykem, který svou syntaxí vychází z C a C++. Počátek projektu Javy se datuje do roku 1990 a byla vytvořena pro to, aby řešila problém s přenositelností kódu mezi různými operačními systémy a hardwarovými platformami, který nebyl v C a C++ nikdy úplně vyřešen. Problém se povedlo vyřešit vyvinutím konceptu virtuálního stroje. Ten na sebe bere úlohu vykonávání programu a vytváří tak mezivrstvu mezi programem a operačním systémem. Výsledkem tohoto řešení je, že Javovský program můžeme spouštět na jakémkoli operačním systému a hardwarové platformě, pro kterou je nějaký virtuální stroj dostupný. Nevýhodou virtuálního stroje je pomalejší vykonávání programu a vyšší nároky na systémové prostředky (jak na procesor tak na operační paměť) oproti standardně kompilovaným programům.

Ve své práci se zaměřuji především na to, jak se u dvou různých implementací virtuálního stroje podařilo naložit s problémem ztráty výkonu a systémové náročnosti. A to jak formou srovnání použitých technik a technologií, tak přímým porovnáním výkonu obou virtuálních strojů.

V následující kapitole 2 se budu zabývat platformou Javy jako takové, abych tím nastínil problematiku běhu programu ve virtuálním stroji a poodhalil specifické vlastnosti, které to obnáší. Kapitola 3 se zabývá srovnáním dvou virtuálních strojů, konkrétně virtuálním strojem HotSpot a virtuálním strojem JamVM. Porovnává a shrnuje použité technologie, vývojové přístupy a z toho pro ně plynoucí vlastnosti. Na základě těchto vlastností jsem v podkapitole 3.3 připravil návrh výkonnostních testů. V kapitole 4 jsou zpracovány a vyhodnoceny výsledky navržených výkonnostních testů. Závěrečná kapitola 5 obsahuje souhrnné srovnání a zhodnocení výsledků a také návrh možného rozšíření této práce. Poslední část práce tvoří příloha, která obsahuje konkrétní naměřené hodnoty, ze kterých jsou vytvořeny grafy uvedené v kapitole 4.

Kapitola 2

Teoretický úvod do platformy Java

Tato kapitola se zabývá použitými technikami a technologiemi platformy Java. Vysvětluje principy, ze kterých plynou její specifické vlastnosti a to jak ty kladné, tak i záporné. Podkapitola 2.1 se zabývá vznikem Javy a její historií, následující podkapitola 2.2 potom obsahuje popis bajtkódu, který tvoří mezistupeň mezi zdrojovým kódem Javy a nativním binárním kódem a se kterým virtuální stroje pracují. Následující podkapitola 2.3 čtenáře seznamuje s konstrukcí platformy Java.

2.1 Historie a návrh

Platforma Java vznikla jako interní projekt společnosti Sun v prosinci 1990 jakožto alternativa k jazykům `C` a `C++`. Při návrhu bylo dbáno na čistotu objektově orientovaného kódu, který se v Javě používá a na zjednodušení úkonů programátora, které s řešením algoritmických problémů příliš nesouvisí. Výsledkem je jazyk, který prakticky nezná slova jako ukazatel či odkaz a místo toho používá základní pravidlo, že základní datové typy jsou předávány hodnotou a objektové datové typy odkazem. Výsledkem je jazyk, který programátora nezatěžuje správou operační paměti a při dodržování základních programátorských pravidel nepřipouští úniky paměti (tzv. memory leak). Díky těmto vlastnostem se Java stala populární v širokém spektru vyvíjených softwarových produktů ať už se jedná o jednoduché webové applety, pokročilé desktopové produkty a editory, až po implementaci řady serverových služeb.V roce 2006 a 2007 byla postupně Java uveřejněna jako open source. Některé knihovny, které byly pro platformu klíčové, však bylo třeba implementovat znovu, protože původní implementace nebylo možné jako open source uveřejnit. Příčinou byla autorská práva firem třetích stran, které se na implementaci implementaci těchto knihoven podílely. V roce 2009 byla společnost Sun odkoupena Oraclem spolu s projektem Javy. Oracle se tedy stal správcem platformy Java a má pod kontrolou směrování jejího vývoje a její specifikace.

2.2 Bajtkód

Aby byly programy psané v Javě přenositelné mezi platformami musí být jejich běh odstíněn od operačního systému na kterém jsou spouštěny. Při vývoji, si ale vývojáři dobře uvědomovali, že čistá interpretace zdrojového kódu je příliš pomalá a pro větší projekty nepoužitelná. Použili tedy techniku kompilování do mezikódu (bajtkódu). Kompilátor Javy ze zdrojových kódů vytvoří .class soubory obsahující program převedený do instrukcí srozumitelných pro virtuální stroj Javy. Virtuálnímu stroji je věnována podkapitola 2.3.1.

Výhodou tohoto postupu je, že se syntaktická a sémantická analýza provádí jen jednou a to při překladu.

Automaticky generované .class soubory jsou vytvářeny pro každou třídu nacházející se v kódu, tedy i pro vnořené a anonymní třídy nebo výčet. Práce s obsahem těchto souborů je základním pilířem virtuálního stroje a z hlediska výkonu je důležité jeho zpracování a analýza.

2.2.1 Binární soubory tříd .class

Jedná se o soubor, který představuje jednu Javovskou třídu, rozhraní nebo výčet. Kromě samotných instrukcí bajtkódu metod(pokud se jedná o třídu) obsahuje informace důležité pro virtuální stroj, těmi to informacemi jsou:

- *Magická konstanta* je to šestnáctková konstanta, která má hodnotu *0xCAFEBABE*. Tato konstanta identifikuje .class soubor, pokud by ji soubor na svém začátku neobsahoval nebude s ním zacházeno jako s platným .class souborem.

- *Minoritní a majoritní verze souboru* jsou čísla, podle nichž je virtuální stroj schopen poznat, jestli danou verzi .class souboru podporuje nebo ne. Rozsah podporovaných verzí pro jednotlivé verze platformy Java určuje společnost Oracle a musí jej respektovat všechny virtuální stroje.

Dále je zde předdefinován tzv. constant pool což je obdoba tabulky symbolů. Podle něj se poté za běhu programu vytváří do paměti jeho obraz se kterým se pracuje. Každá třída musí mít také definovány modifikátory vlastností dědičnosti, ty určují jestli a jak se může z třídy dědit a také modifikátory přístupových práv, které se používají při kontrole zapouzdření. Všechny tyto údaje jsou definovány pomocí jediné šestnácti bitové masky. Rozsah masky zatím není plně využitý a obsahuje volné bity pro možné budoucí použití. Dalšími položkami .class souboru jsou symbolické odkazy do constant poolu a to konkrétně odkaz na aktuální třídu (v syntaxi programovacího jazyka je to atribut *this*) a odkaz na nadtřídu neboli předka (v syntaxi programovacího jazyka je to atribut *super*). Po odkazech následuje počet a seznam implementovaných rozhraní s odkazy do constant poolu, počet a výčet datových položek třídy, jejich modifikátorů a nakonec počet a případná implementace jednotlivých metod.

Ilustrační příklad toho jak je soubor se zdrojovým kódem Javy překompilován do binárního .class souboru je uveden níže.

Zdrojový kód ilustračního příkladu:

```
package empty;

public class Main {
  public static void main(String[] params){
  }
}
```

Obsah binárního class souboru třídy s jedinou a prázdnou metodou main, převedený do textové podoby pomocí nástroje *javap*:

```
public class empty.Main
  SourceFile: "Main.java"
  minor version: 0
  major version: 51
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Class              #2            //  empty/Main
   #2 = Utf8               empty/Main
   #3 = Class              #4            //  java/lang/Object
   #4 = Utf8               java/lang/Object
   #5 = Utf8               <init>
   #6 = Utf8               ()V
   #7 = Utf8               Code
   #8 = Methodref          #3.#9         //  java/lang/Object."<init>":()V
   #9 = NameAndType        #5:#6         //  "<init>":()V
  #10 = Utf8               LineNumberTable
  #11 = Utf8               LocalVariableTable
  #12 = Utf8               this
  #13 = Utf8               Lempty/Main;
  #14 = Utf8               main
  #15 = Utf8               ([Ljava/lang/String;)V
  #16 = Utf8               params
  #17 = Utf8               [Ljava/lang/String;
  #18 = Utf8               SourceFile
  #19 = Utf8               Main.java
{
  public empty.Main();
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #8                 // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 3: 0
      LocalVariableTable:
```

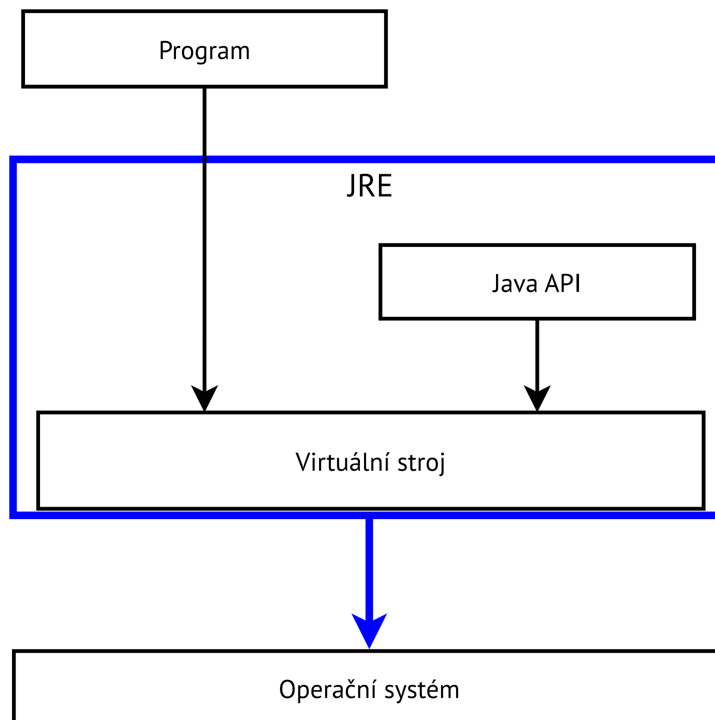
```
        Start  Length  Slot  Name   Signature
               0       5     0  this   Lempty/Main;

  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=0, locals=1, args_size=1
         0: return
      LineNumberTable:
        line 10: 0
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
               0       1     0 params   [Ljava/lang/String;
}
```

V obsahu class souboru si je dobré všimnout sekce constant poolu, kde můžeme vidět z čeho se jeho jednotlivé položky skládají. Každá položka je složena z indexu v constant poolu, svého datového typu a své hodnoty, která může odkazovat na jiné místo v constan poolu. Zajímavostí je, že odkazování mimo samotný constant pool probíhá symbolicky pomocí řetězců. Dále je dobré si uvědomit, že binární soubor obsahuje dvě metody *empty.Main()* a *main(java.lang.String[])* zatím co ve zdrojovém kódu je pouze jedna. Metoda *empty.Main()* je totiž vygenerována automaticky a jedná se o implicitně generovaný konstruktor, zatím co metoda *main(java.lang.String[])* je hlavní metodou, implementovanou ve zdrojovém kódu třídy.

2.3 Běhové a vývojové prostředí Javy

K řádnému spuštění programu psaného v Javě jsou potřeba dvě základní komponenty, virtuální stroj popsaný v podkapitole 2.3.1 a Java API (Application Programming Interface) uvedené v podkapitole 2.3.2, komponentám jako celku se říká „běhové prostředí" (Java Runtime Enviroment). Java API je načteno automaticky virtuálním strojem po jeho spuštění, takže programy, které jej využívají mohou s jeho přítomností v době jejich běhu počítat [6]. Na obrázku 2.1 je znázorněno jak běhové prostředí vytváří vrstvu mezi programem a operačním systémem a jak zapouzdřuje své hlavní komponenty. Otevřenost platformy Java také umožňuje vývojářům vytvořit si vlastní virtuální stroj, který nejlépe odpovídá jejich požadavkům a potřebám. Pro zajištění kompatibility bajtkódu mezi jednotlivými stroji je potřeba dodržet formální specifikaci, která je blíže představena v podkapitole 2.3.3.



Obrázek 2.1: Schéma běhového prostředí

2.3.1 Virtuální stroj

Virtuální stroj Javy, anglicky označovaný Java Virtual Machine (JVM nebo jen VM), je program starající se nejen o samotné vykonávání instrukcí bajtkódu, ale také o správné načítání tříd, správu haldy se kterou pracuje a jiných operací nezbytných k vytvoření kompletního virtuálního prostředí. Implementace virtuálního stroje je obvykle realizována pro každou platformu samostatně v některém z nativně kompilovaných jazyků jako jsou `C` nebo `C++`. Přesto, že je možné virtuální stroj implementovat v libovolném programovacím jazyce, jedním z kritických požadavků na virtuální stroj je rychlost interpretace bajtkódu, popsaného v kapitole 2.2. Z tohoto důvodu volíme při implementaci programovací jazyky, které rychlé vykonávání kódu samotného virtuálního stroje mohou zaručit.

Druhým požadavkem je většinou multiplatformnost virtuálního stroje. Přestože se obvykle kompilace pro jinou platformu neobejde bez zvětších či menších úprav, je jednodušší mít program napsaný v jazyce pro který existují kompilátory na co největší množství platforem, aby se co nejvíce již napsaného kódu dalo znovu použít.

2.3.2 Java API

Java API je předem předpřipravený soubor tříd, který má programátor ihned k dispozici pro své využití. Způsob implementace je pro výkon celé platformy podstatný, třídy Java API jsou programátory hojně využívány a při nevhodné implementaci by mohly běh programu zbrzďovat. Třídy jsou rozděleny tematicky do několika (desítek) balíčků. Z tohoto hlediska je Java API podobné například standardním knihovnám v `C`. Vzhledem k tomu, že je soubor tříd Java API velice rozsáhlý, byl ještě dále rozdělen na tzv. platformy, které definují jaké typy programů se v daném běhovém prostředí dají spouštět.

V základu se rozlišují tyto tři platformy:

- „Java Platform, Standard Edition" (Java SE) Tato platforma se využívá především pro psaní desktopových aplikací a programů [3].
- „Java Platform, Enterprise Edition" (Java EE) Platforma zaměřená především na serverové nasazení rozšiřuje Javu SE například o transakční model a databázové API [1].
- „Java Platform, Micro Edition" (Java ME) Platforma vytvořena pro menší mobilní zařízení jakými jsou například PDA, na rozdíl od Enterprise edice neobsahuje všechny třídy obsažené v původní Standardní edici [2].

Na rozdíl od standardních knihoven používaných v C, není kód tříd Java API součástí zkompilovaného programu. Je počítáno s tím, že všechny požadované třídy Java API budou k dispozici virtuálnímu stroji v rámci běhového prostředí. [4]

2.3.3 Specifikace virtuálního stroje

Důležitým krokem k otevření platformy Java bylo vydání přesné technické specifikace, kterou musí každý virtuální stroj splňovat pro to, aby byl schopen správně interpretovat korektně vygenerovaný bajtkód. Specifikaci lze nalézt jak na internetových stránkách společnosti Oracle[1], tak i tištěnou jako knihu [7] (obsahově jsou obě verze totožné). Dokument se zabývá různými aspekty a funkcionalitami virtuálního stroje a jeho vývoje, ale nezabývá se optimalizací výkonu virtuálního stroje, tato část implementace je zcela v rukou vývojářů. Dokument se také zabývá konceptem programování v Javě jako takovým, popisem virtuálního stroje, jeho spuštění, toho jak má načítat, linkovat a inicializovat třídy, dále se zabývá obsahem binárních .class souborů a instrukční sadou, která je v nich použita. Popisuje také návrh Javovských vláken a jejich spolupráci s pamětí. V rámci specifikace má programátor dánu spoustu volnosti v návrhu a implementaci svého řešení, nicméně některé požadavky musí být přesně dodrženy. Jedná se především o správnou implementaci datových typů a sémantiky instrukcí.

2.4 Shrnutí

Spouštění a běh programů na platformě Java dává vývojářům jistotu kompatibility zkompilovaných programů napříč hardwarovými platformami i operačními systémy. Za tento luxus se však jako vždy platí. V případě Javy je to vyššími nároky na systémové prostředky, které virtuální stroj potřebuje a poklesem výpočetního výkonu, který virtuální stroj jakožto softwarová vrstva zapříčiňuje. Následující kapitoly se tedy zaměří na to, jak si proti sobě stojí dva konkurenční virtuální stroje a to jak z hlediska nároků na systémové prostředky, tak z hlediska výpočetního výkonu.

[1] http://docs.oracle.com/javase/specs/jvms/se5.0/html/VMSpecTOC.doc.html

Kapitola 3

Srovnání virtuálních strojů HotSpot a JamVM

V této kapitole se čtenář blíže seznámí se dvěma konkurenčními virtuálními stroji, jimiž se tato práce zabývá a to: virtuálním strojem HotSpot a virtuálním strojem JamVM. V podkapitole 3.1 bude seznámen s použitými technologiemi, které byly při vývoji obou virtuálních strojů použity a s vlastnostmi, které z jejich použití plynou. V následující podkapitole 3.2 bude dále čtenář seznámen s optimalizačními technikami, které virtuální stroje používají pro zvýšení výpočetního výkonu. Posledním důležitým bodem této kapitoly pak je návrh výkonnostních testů, popsaný v podkapitole 3.3, které byly pro porovnání výkonu a vlastností obou virtuálních strojů použity.

Oba virtuální stroje plní stejnou úlohu a každý program, který na nich spustíme vykonají stejně, přesto jsou v mnoha ohledech dost odlišné. Začít by se dalo rovnou u vývoje. Přesto, že zdrojové kódy HotSpotu byly uvolněny jako open source, je od počátku jeho vývoj zaštítěn silnou softwarovou společností. Dříve to byl Sun, dnes je jí Oracle, viz. historie Javy v podkapitole 2.1. To s sebou přináší jisté výhody, jakými jsou například: nepřetržitá práce zkušených vývojářů, či přístup ke know-how a výzkumům, které se mimo společnost nedostanou. Na druhou stranu jsou na HotSpot jako na referenční virtuální stroj kladeny vyšší požadavky z hlediska spolehlivosti, výkonu či bezpečnosti. Naproti tomu čistě komunitní projekt JamVM, který byl z počátku napsán jedním člověkem a následně uveřejněn jako open source, si klade za cíl poskytovat minimalistickou variantu virtuálního stroje, který plně odpovídá aktuální druhé verzi technické specifikace, která byla uvedena v podkapitole 2.3.3 a zároveň poskytuje dostatečný výkon k běhu napsaných programů. Jeho vývoj mezi open source komunitou může stále probíhat, avšak poslední stabilní verze, kterou se tato práce zabývá, byla vydána v lednu 2010 [5].

3.1 Použité technologie

Z hlediska použitých technologií, zejména vývojových prostupů, použitého jazyka a způsobu interpretace kódu, je mezi oběma virtuálními stroji podstatný rozdíl. Prvním i když z hlediska samotného výkonu virtuálního stroje nejméně podstatným je zvolený programovací jazyk. HotSpot je psán v objektově orientovaném `C++`, naproti tomu JamVM byl od začátku vyvíjen v `C`. Obě volby mají svá pro a proti. `C++` se jakožto objektově orientovaný jazyk zdá pro objektově orientovanou Javu logičtější volbou, ale vzhledem ke konstrukci virtuálního stroje potažmo interpretu to nemusí být zásadní výhoda. Hlavní výhodu představuje

lepší a jednodušší možnost dekompozice celého projektu, který tak může být rozsáhlejší a dlouhodobě lépe udržovatelný. Jako nevýhoda by se dala zmínit vyšší paměťová náročnost zkompilovaných programů. Volba jazyka `C` u JamVM vychází z úmyslu napsat co nejmenší a přesto výkoný interpret Javy. To znamená, že požadavky na maximální možnou dekompozici a udržovatelnost projektu v týmu s větším počtem lidí nejsou tolik kritické. Jedním z hlavních cílů při vývoji JamVM byla nízká paměťová náročnost virtuálního stroje a proto tuto vlastnost je `C` dobrou volbou.

Z hlediska konstrukce interpretu se oba virtuální stroje zásadně liší. Primárním cílem při vývoji HotSpotu bylo maximalizovat jeho výkon, což znamená maximalizovat rychlost vykonávání bajtkódu Javy. Vývojáři se rozhodli zvolit cestu takzvaně šablonově založeného interpretu. Šablony představují bloky automaticky generovaného assemblerovského kódu pro jednotlivé instrukce bajtkódu. Interpret je generován automaticky podle tabulky šablon, která poskytuje funkce pro získání šablon k jednotlivým instrukcím bajtkódu. Nevýhodou tohoto přístupu je horší přenositelnost virtuálního stroje. Automatický generátor musí být pro každou platformu přepsán protože obsahuje značné množství assemblerovského kódu, což značně ztěžuje vývoj, navíc ladění automaticky generovaného interpretu je obtížnější. Další nevýhodou je pomalejší start virtuálního stroje, protože jeho vygenerování stojí cenné milisekundy, na což je zaměřen jeden z výkonnostních testů, uvedený v podkapitole 4.8. Vývoj JamVM měl za cíl kromě paměťově nenáročného interpretu vytvořit virtuální stroj i pro platformy na kterých HotSpot neběžel (třeba proto, že platforma byla příliš malá na to aby se jeho portace vyplatila). Je konstruován jako klasický "přepínačový"interpret který pomocí klasické Cčkovské funkce *switch* pro jednotlivé instrukce bajtkódu volí odpovídající kód v jazyce C. Nevýhodou tohoto postupu je, že instrukce na konci přepínače jsou vykonávány pomaleji, nežli ty nahoře a také to, že automaticky generovaný nativní kód je o něco pomalejší. Na druhou stranu to dává jednodušší možnost portace JamVM na různé další platformy s relativně malým úsilím. Nejzásadnějším omezením pro JamVM při portaci je striktní vyžadování implementace POSIX vláken na úrovni operačního systému. Oficiálně jsou virtuálním strojem JamVM podporovány platformy: x86(_64), ARM, SPARC, PowerPC (PowerPC64) a MIPS. Zatímco HotSpotem jsou podporovány pouze platformy x86(_64), ARM, SPARC (pouze s OS Solaris).

3.2 Optimalizace výkonu

Interpretování kódu na úrovní softwaru bylo vždy pomalejší než-li běh programu přeloženého do nativního kódu, z tohoto důvodu bylo nutné na této vlastnosti zapracovat, aby měla platforma Java šanci uspět. Společnost Sun si toho byla vědoma a při návrhu virtuálního stroje HotSpot se na výkon kladl veliký důraz. Od toho se odvíjí samotné jméno virtuálního stroje, kdy se za běhu programu monitorují jeho takzvaná "horká místa", což jsou úseky programu, ve kterých virtuální stroj tráví nejvíce času jejich vykonáváním. Této technice sledování běhu programu se také někdy říká profilování. Vytypované úseky programu pak prochází poměrně zásadní optimalizací, kterou je překlad do nativního kódu dané platformy. Tomuto postupu se říká kompilace za běhu programu, anglicky just-in-time compiling, zkráceně JIT. Kompilace je výpočetně náročná operace a může mít za následek dočasnou ztrátu výkonu virtuálního stroje. Je tedy zapotřebí, aby byla prováděna jen tam kde to bude mít z dlouhodobého hlediska pozitivním vliv na výkon. Optimalizace by tedy měla celkově zkrátit dobu provádění daného kusu programu o více času, než kolik se ztratí jeho kompilací. K samotné kompilaci se přidávají i další optimalizace, jako třeba optimalizace cyklů či podmínek, kdy virtuální stroj analyzuje bajtkód a přebytečné kusy

kódu záměrně vynechává a vůbec je neprovádí.

Virtuální stroj HotSpot má v 32-bitové verzi k dispozici dva JIT kompilátory, které se nazývají Client a Server, navzájem se od sebe liší mírou prováděných optimalizací a tedy i délkou samotné kompilace. Server provádí hlubší analýzu kódu a kompilace trvá déle, zatímco kompilátor Client je při kompilaci kódu rychlejší nicméně u dlouhodobě běžících programů podává nižší výkon. V nejnovějších 64-bitových distribucích JRE od Oraclu je k dispozici HotSpot pouze se Server kompilátorem.

Další důležitou optimalizací v rámci běhu programu je takzvané vkládání metod (anglicky *inlining*). Invokace metody byla z hlediska výkonu považována za úzké hrdlo virtuálního stroje, proto se za běhu programu kód metody vkládá na místo, kde je metoda volaná. Tím se odstraní přebytečná režie obslužných rutin, která je při klasickém invokování metody přítomna. Vzhledem k tomu, že v Javě se standardně programuje vůči rozhraní, může se kód volané metody za běhu měnit. Je proto potřeba aby byl virtuální stroj schopen provedené optimalizační techniky vrátit zpět (proces deoptimalizace) a umožnit tak vykonat metodu s kódem aktuálně použité třídy. Techniku vkládání metod a deoptimalizaci kódu podporuje na platformě x86 standardně i interpret JamVM.

3.3 Návrh výkonnostních testů

Sada výkonnostních testů je navržena s ohledem na použité technologie obou virtuálních strojů. Testy mají prověřit ty vlastnosti strojů, na které byl při jejich implementaci kladen důraz a také to, jak si poradí s nejběžnějšími programovými konstrukcemi používanými napříč všemi programovacími jazyky. Jejich úkolem je rovněž prověřit, jak a jestli se při běhu projeví optimalizační techniky, které vývojáři u svých virtuálních strojů implementovali. Každý z testovacích programů je pro zvýšení přesnosti měření a eliminaci chyby, která by mohla vzniknout, spouštěn pětkrát. Pokud je variační koeficient naměřeného datového vzorku pod 10%, je průměrná hodnota, která je vynášena do grafu brána jako reprezentativní hodnota datového vzorku. V našem případě je datový vzorek získán z pěti spuštění daného testu. Podle nastavení testů při spuštění jsou buďto všechny naměřené hodnoty zaznamenávány, nebo je měřena celková doba běhu testu či jiná pro test specifická veličina. V následujících podkapitolách jsou uvedeny testované vlastnosti a pro ně navrhnuty možnosti testování.

3.3.1 Cyklení a práce s polem

Kombinace cyklů a polí patří k častým programovým konstrukcím. Zejména u dlouhých polí se často stává, že jeho procházení zabere velkou část z celkového času, po který program běží. Pro otestování výkonnosti virtuálních strojů po této stránce byl vybrán test bubblesort, který bude nad dostatečně velkým polem provádět řazení. V tomto testu by se také mohly projevit možné optimalizace ze strany virtuálních strojů.

3.3.2 Práce s objekty a constant poolem

Vzhledem k tomu, že je Java objektově orientovaný programovací jazyk, je zapotřebí rychlá práce s objekty, jako je jejich vytváření a kombinace. Práce s objekty je proto důležitým úkolem pro virtuální stroj. Při vytváření objektů a odkazování se na ně používá virtuální stroj constant pooly, které jsou obdobou tabulky symbolů z jiných jazyků. Rychlý přístup do constant poolu a nalezení příslušných dat a hodnot je kritickou operací, jejíž rychlost

může zásadně ovlivnit výkonnost programu. Pro otestování toho, jak si virtuální stroje výkonnostně stojí při práci s těmito vnitřními datovými strukturami byl zvolen test při kterém se řadí velké množství dat pomocí quick-sortu. Data jsou vkládána do objektů, z těch jsou poté čtena a přemisťována do jiných nově vzniklých objektů. Práce virtuálního stroje s constant poolem je tak maximalizována.

3.3.3 Invokování metod

Práce s objekty není pouze o jejich vytváření a kombinaci, ale také o využívání metod kterými disponují. Invokace (volaní) metody daného objektu patří k základním operacím které s objektem jako takovým programátor provádí. Rychlost invokace metody byla v raných verzích Javy úzkým hrdlem výkonu. Test má tedy za úkol změřit jak dlouho tato operace zabere. Aby bylo možné porovnat účinnost optimalizačních, byly pro tento účel navrženy dva testy. Oba v cyklu invokují metody přičemž měří čas o její invokace do doby než vrátí hodnotu. Rozdílem je, že první z testů umožní virtuálnímu stroji aby použil optimalizační techniku vkládání, zatím co v druhém testu se kód volané metody za běhu programu mění a optimalizace vkládáním zde není možná.

3.3.4 Rychlost rekurze

Rekurzivní volání je typickou konstrukcí pro některé typy výpočtů. Virtuální stroje disponují svými vlastními zásobníky které ukládají aktuální stav prováděné metody při invokování metody jiné. Pro prověření rychlosti jakou tyto zásobníky pracují, byl použit pro tento test rekurzivní výpočet faktoriálu.

3.3.5 Práce s řetězci

Rychlost práce s řetězci je důležitá zejména u serializace a deserializace objektů, což je často využívaná technika při komunikaci klientských programů se serverem či při napojení programu na databázi. Asi nejnáročnější operací kterou s řetězci jako objekty můžeme provádět je jejich konkatenace. Java nabízí dvě základní možnosti jak takovou operaci provést.

První možností je použití operátoru plus (+), tato varianta se z výkonnostního hlediska nedoporučuje. Při vyhodnocování výrazu se vytváří nová instance objektu, která je následně přiřazena jako výsledek. Samotné vytváření nového objektu má negativní dopad na rychlost této operace.

Druhou a doporučovanou možností je použití objektu třídy StringBuilder. Tento objekt následně umožňuje zřetězovat stringy pomocí jedné z jeho metod tak, že se při této operaci nevytváří další instance objektu.

Pro otestování rychlosti práce s řetězci byl zvolen řetězec „Hello my sunny world“, který je následně v cyklu konkatenován za sebe do jednoho dlouhého řetězce. Konkatenace je provedena pomocí operátoru plus (+) u prvního testu a pomocí objektu třídy StringBuilder u testu druhého.

3.3.6 Vyhodnocování podmínek

Podmínky jsou základním nástrojem pro větvení programu. Jejich použití je i v miniaturních projektech prakticky nevyhnutelné. U takovéto základní programové konstrukce očekává programátor maximální možný výkon. Pro otestování byl navržen program který v sobě

různě zanořuje podmínky a testuje hodnoty proměnných. Hodnoty některých proměnných jsou testovány opakovaně, aby virtuální stroje měly možnost provést své optimalizační techniky (pokud jimi disponují).

3.3.7 Rychlost startu virtuálního stroje

Rychlost spuštění virtuálního stroje se nemusí jevit jako důležitá vlastnost. Opak je však pravdou. Zejména u kratších programů, které jsou opakovaně spouštěny nějakým skriptem, může start virtuálního stroje trvat déle než běh programu samotného. Pro změření délky startu byl použit program s prázdnou hlavní metodou a je měřen čas od doby kdy byl virtuální stroj spuštěn, do doby kdy, virtuální stroj vrátí návratovou hodnotu konce spuštěného programu.

3.3.8 Výpočet v plovoucí desetinné čárce

Výpočty v plovoucí desetinné čárce patří na hardwarové úrovni k těm nejnáročnějším operacím. Virtuální stroj, který spouští Javovské programy, běží na stejném hardwaru jako každý jiný program kompilovaný do nativního kódu dané platformy. Výpočetní výkon tohoto stroje tedy přímo souvisí s hardwarem na kterém je spuštěn. Tento test by měl odhalit, jaké rozdíly tedy jsou mezi jednotlivými virtuálními stroji běžícími na tom samém fyzickém zařízení. Jako testovací algoritmus byl použit iterační výpočet čísla π na stanovený počet přesných desetinných míst.

3.3.9 Velikost virtuálního stroje v operační paměti

Virtuální stroj jako takový je software, který na počítači běží a zabírá určité množství systémových prostředků. Na některých vestavěných systémech, kde se Java používá, jsou systémové prostředky (zejména operační paměť) velice omezené. Test velikosti virtuálního stroje má odhalit kolik těchto prostředků si jednotlivé virtuální stroje pro svůj chod zaberou. Tento test na virtuálním stroji spustí program s nekonečnou smyčkou. Následně se z operačního systému odečte velikost operační paměti zabrané virtuálním strojem. Po odečtení je virtuální stroj vypnut.

3.3.10 Práce s haldou

Virtuální stroje Javy využívají ke správě haldy automatického správce paměti (anglicky garbage collector). Správce paměti je proces, který běží na pozadí a uklízí po programátorovi objekty na které již neexistuje v programu žádný aktuální odkaz. Aktivita tohoto správce nicméně zvyšuje nároky na systémové prostředky a navíc může při úklidu docházet k dočasnému zastavení aktuálně běžícího programu ve virtuálním stroji. Aktivnější správce paměti tedy může ubírat výpočetní výkon virtuálního stroje, ale na druhou stranu může šetřit operační paměť protože nepotřebné zdroje častěji uvolňuje. Pro otestování chování správců paměti jednotlivých virtuálních strojů byl použit algoritmu quick-sort (stejný jako u testu práce s objekty v podkapitole 3.3.2), při jehož běhu běží na pozadí další proces který monitoruje velikost haldy a změny zaznamenává.

3.4 Shrnutí

V podkapitole 3.1 se čtenář seznámil s technologiemi a technikami použitými při vývoji virtuálních strojů HotSpot a JamVM. Poukázala na problémy a vlastnosti, které se při použití virtuálních strojů vyskytují. Čtenář byl také seznámen s některými možnostmi, které řeší výkonnostní problémy virtuálních strojů v podkapitole 3.2. Na základě použitých technologií a výhod / nevýhod jednotlivých implementací virtuálních strojů, byly navrženy výkonnostní testy. Návrh těchto testů je uveden v podkapitole 3.3. Konkrétní implementace a výsledky navržených testů jsou uvedeny v následující kapitole 4.

Kapitola 4

Výkonnostní testy

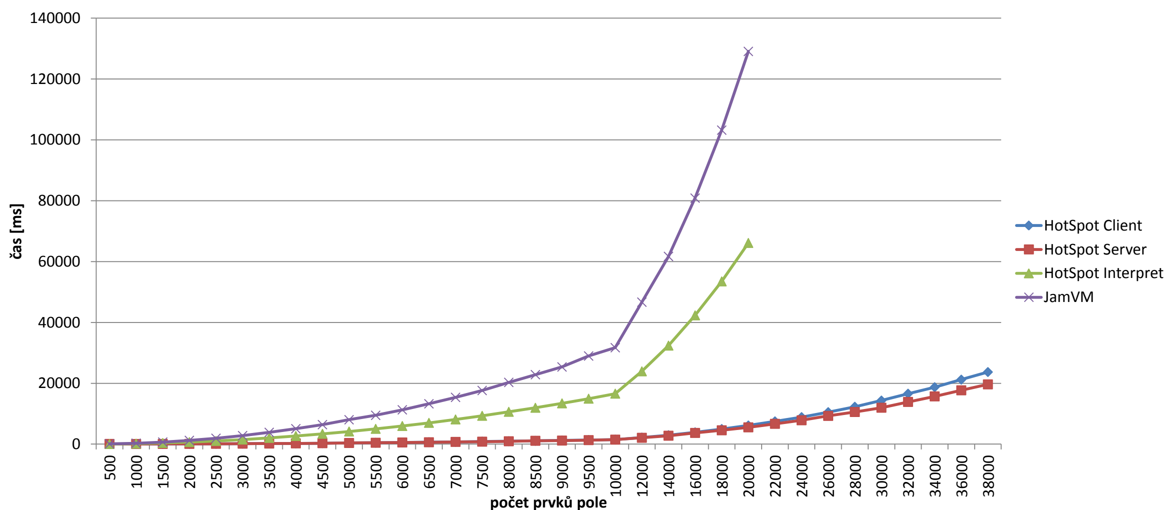
Cílem sady testů navržených v předešlé kapitole 3 je otestovat vlastnosti a změřit výkonnost zvolených virtuálních strojů. Při návrhu bylo přihlédnuto k použitým optimalizačním technikám jednotlivých strojů, zároveň se však klade důraz na použití naprosto běžných a elementárních programových konstrukcí, které se ve zdrojových kódech jiných programů nachází ve větší míře. Navržené testy byly prováděny na této testovací konfiguraci:

- Procesor: Intel Core i3 330M 2,1GHz
- Operační paměť: 8GB DDR3 1333MHz
- Pevný disk: Samsung Spinpoint 500GB 2,5", 5400 Ot/min, 8MB Cache
- Operační Systém: Linux, Fedora 32-bit.

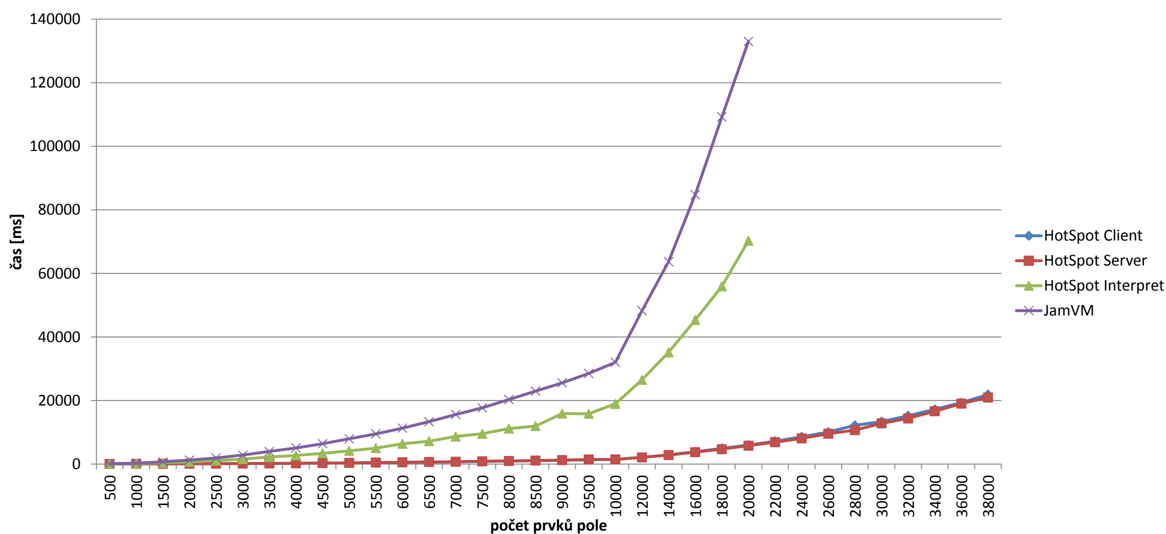
Každá podkapitola se zabývá jedním typem ze všech provedených testů. Nejprve je uveden stručný popis implementace testu a následně jsou vykresleny grafy s vynesenými naměřenými hodnotami. Závěrem každé podkapitoly je vyhodnocení daného testu. Konkrétní naměřené hodnoty jsou uvedeny v tabulkách, které jsou součástí přílohy této práce. V poslední podkapitole jsou potom výsledky testů shrnuty a souhrnně vyhodnoceny.

4.1 Řazení pole algoritmem bubble-sort

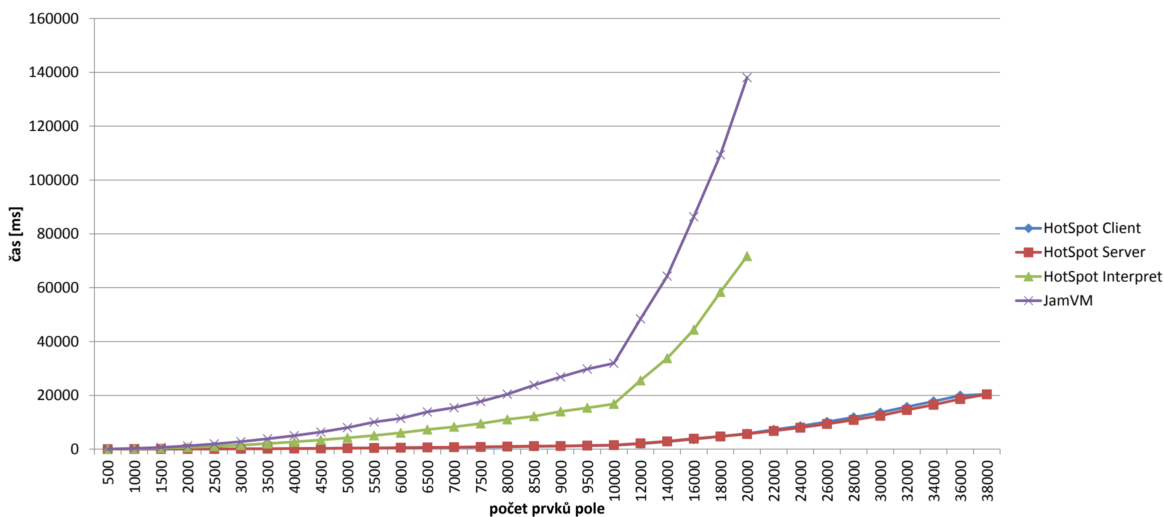
Sada testů bubble-sort byla navržena a na implementována k tomu aby otestovala rychlost cyklení a rychlost práce s polem. Při použití čistého interpretu jakým je JamVM a HotSpot přepnutý do "interpret only"režimu, byly testy z časových důvodů zkráceny. Při měření se používá proměnná délka kroku, který navyšuje velikost pole v další pětici testů, ze začátku je krok menší, aby bylo možné pozorovat případné výkonnostní odchylky způsobené just-in-time kompilací. Řazení probíhá pokaždé nad totožným vzorkem dat, který se nachází v předpřipraveném souboru a který obsahuje 10 milionů pseudonáhodných celých čísel (integerů). Samotný algoritmus bubble sortu neobsahuje žádné heuristické techniky. Do testu taktéž není zahrnuta doba, kdy se do paměti načítají data ze souboru. V grafech uvedených níže v této podkapitole je na ose x vyznačena velikost řazeného pole a na ose y čas, který virtuální stroj k jeho seřazení potřeboval. Výsledky jednotlivý virtuálních strojů a různých konfigurací HotSpotu jsou v grafech barevně odlišeny.



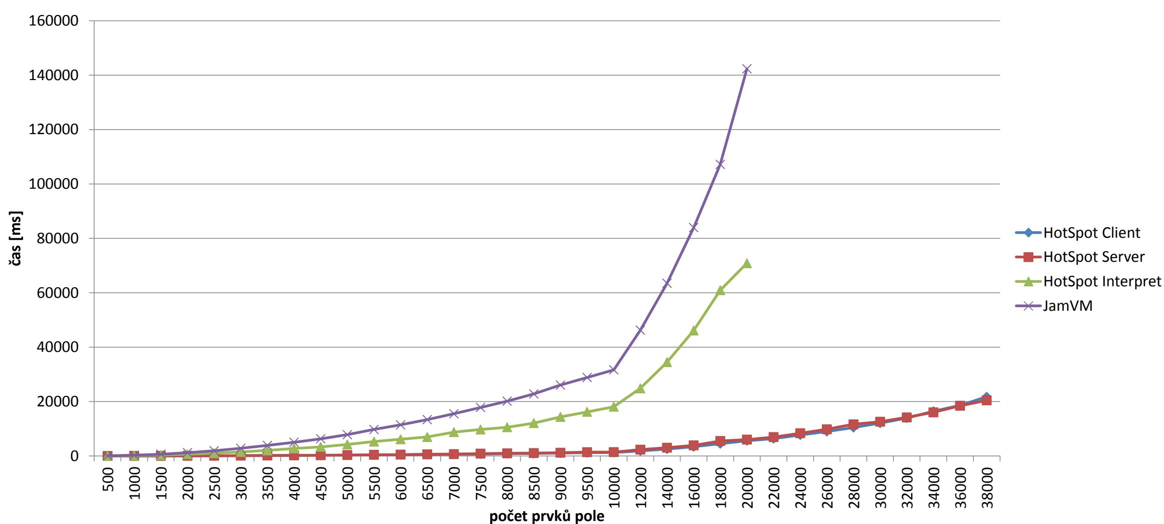
Obrázek 4.1: Výsledky testů pro algoritmus bubble-sort s použitím for cyklu



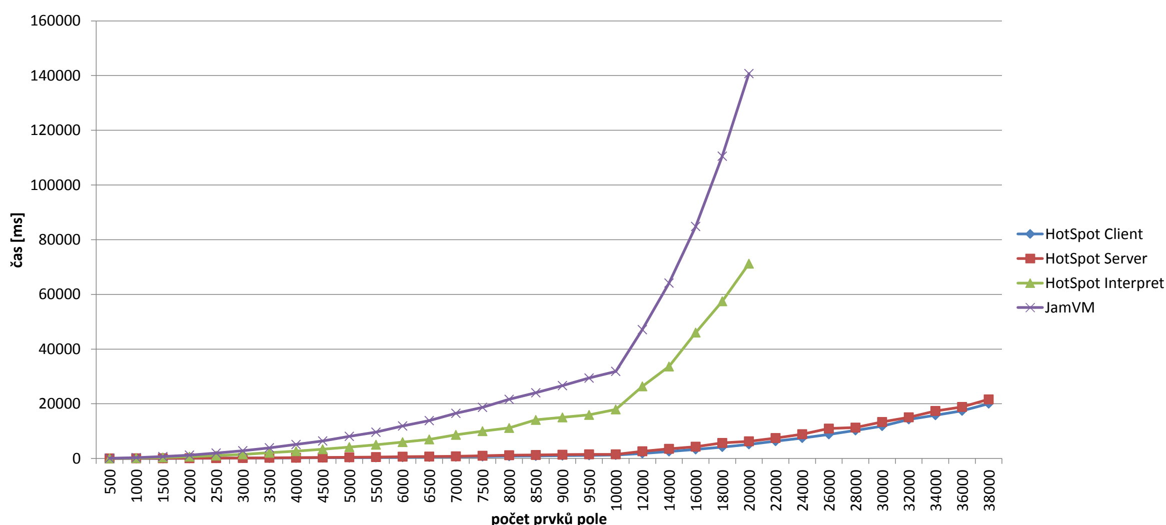
Obrázek 4.2: Výsledky testů pro algoritmus bubble-sort s použitím while-do cyklu s post inkrementací počítadla



Obrázek 4.3: Výsledky testů pro algoritmus bubble-sort s použitím while-do cyklu s pre inkrementací počítadla



Obrázek 4.4: Výsledky testů pro algoritmus bubble-sort s použitím do-while cyklu s post inkrementací počítadla



Obrázek 4.5: Výsledky testů pro algoritmus bubble-sort s použitím do-while cyklu s pre inkrementací počítadla

Z výše uvedených grafů a naměřených hodnot uvedených v podkapitole A.1, poměrně jednoznačně vyplývá, že virtuální stroj HotSpot s jakoukoli konfigurací má oproti JamVM výkonnostní převahu. Rychlost řazení HotSpotu se zapnutým JIT kompilátorem je až 23× větší než u konkurenčního JamVM. U konfigurací s použitým JIT kompilátorem je důvod výkonnostního rozdílu naprosto zřejmý. Cyklení v nativním kódu je oproti cyklení pomocí interpretovaného kódu nesrovnatelně rychlejší. JIT kompilátory navíc používají optimalizační techniky k omezení počtu prováděných podmínek při kontrole velikosti procházeného pole. Pokud bychom se zaměřili na srovnání interpretů tak můžeme vidět, že interpret HotSpotu je přibližně 2× rychlejší. To je zapříčiněno odlišnou konstrukcí interpretu a optimálnějším překladem instrukcí. U testů s malou velikostí řazených polí (do 500 prvků) se projevuje nevýhoda Server kompilátoru, jehož kompilace je natolik pomalá, že ve výkonnostních testech mnohdy prohrává i s interprety. U takto krátkých běhů programů pro Server kompilátor jsou naměřené hodnoty pouze orientační. Variační koeficient přesahuje v tabulkách A.4,A.8 40% a v tabulce A.12 přesahuje variační koeficient hodnotu 10%. Tyto nepřesnosti jsou způsobeny JIT kompilací a profilováním kódu, jehož rychlost může být značně ovlivněna aktuálním stavem operačního systému. Zbývající naměřené hodnoty jsou již stabilní, jejich variační koeficient má hodnotu pod 10% a průměrné hodnoty vynesené v grafech jsou tedy reprezentativní. Rozdíly mezi implementací jednotlivých cyklů jsou minimální a výkonnostní rozdíly mezi různou implementací počítadel jsou na hranici chyby měření. Pokud bychom se drželi naměřených výsledků, tak nejrychlejší je implementace pomocí for cyklu, na druhém místě je implementace pomocí while-do cyklu a na třetím implementace pomocí do-while cyklu. Zajímavostí tohoto testu je, že pro cykly do-while provedl optimalizaci bajtkódu už kompilátor Javy a tyto bajtkódy se od sebe ani při rozdílné práci s počítadlem neliší.

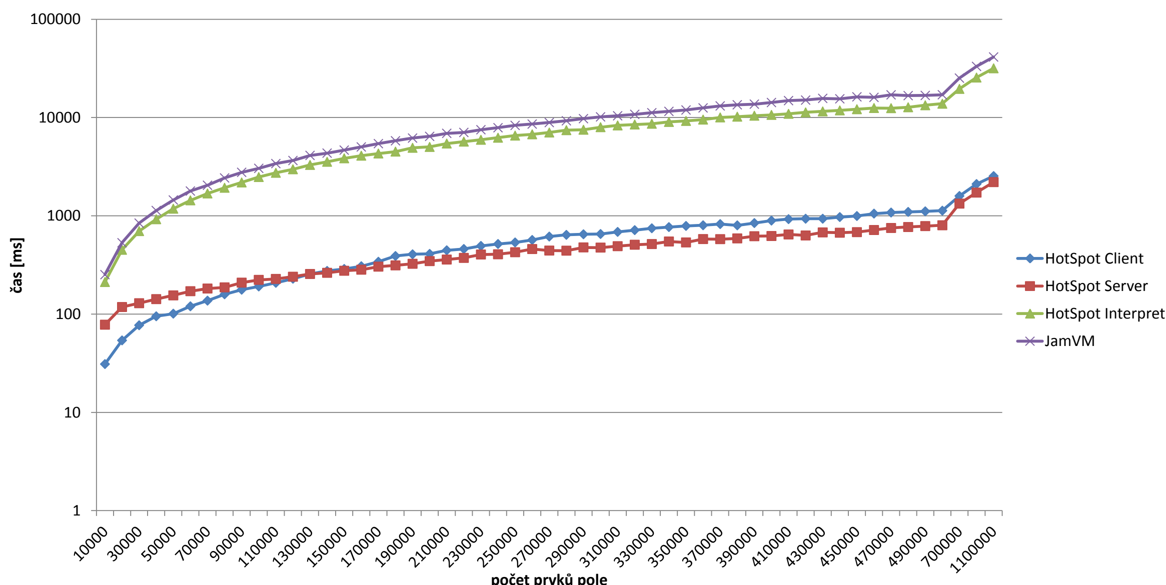
4.2 Řazení pole algorimem quick-sort

Tento test je zaměřen na otestování rychlosti vnitřních datových struktur virtuálních strojů. Řadící algoritmus quick-sort má více možných implementací, přičemž různé implementace

mohou dosahovat rozdílných výsledků v testu. Pro daný test byla implementace quick-sortu upravena tak aby maximalizovala práci s constant-pooly. Pseudokód použitého quick-sortu je následující:

1. *Vlož do zásobníku Stack neseřazený List jdi na bod 2.*
2. *Pokud zásobník Stack není prázdný jdi na bod 3. jinak jdi na bod 10.*
3. *Vytáhni ze zásobníku Stack první list a přiřaď jej do listu Act.*
4. *Pokud je délka Act rovna 1 přidej jej do listu Result a jdi na bod 2. Jinak jdi na bod 5.*
5. *Vygeneruj Pivot a jdi na bod 6.*
6. *Každou položku v listu Act porovnej s Pivotem. Pokud je menší vlož ji do listu Left, pokud je stejný vlož ji do listu Center a pokud je větší vlož jis do listu Right. Jdi na bod 7.*
7. *Pokud list Right není prázdný vlož ho do zásobníku Stack a jdi na bod 8, jinak jdi rovnou na bod 8.*
8. *Pokud jsou listy Right i Left prázdné, vlož list Center do listu Result a jdi na bod 2. Jinak vlož list Center do zásobníku Stack a jdi na bod 9.*
9. *Pokud list Left není prázdný vlož ho na zásobník Stack a jdi na bod 2. Jinak jdi rovnou na bod 2.*
10. *Vrať list Result.*

Vzhledem k rychlosti řadícího algoritmu nebylo u tohoto testu nutné odlišovat práci virtuálních strojů a maximální velikost řazeného vzorku dat mohla být pro všechny virtuální stroje stejná. Stejně jako u testu Bubble-Sortem byla i zde použita proměnná délka kroku. Rozdílem však je velikost řazených vzorků, které u tohoto testu musely být kvůli přesnosti testů větší, protože řazení quick-sortem je příliš rychlé, na to aby se v jeho průběhu eliminovaly odchylky u malého vzorku dat. V grafu níže jsou vyneseny zprůměrované naměřené hodnoty pro jednotlivé virtuální stroje, případně jejich různé konfigurace. Na ose x se nachází hodnoty představující velikost řazeného pole a na ose y čas, který byl potřebný k jeho seřazení. Pro lepší ilustraci naměřených výsledků je časová osa zobrazena pomocí logaritmického měřítka o základu 10. Přesné naměřené hodnoty jsou uvedeny v příloze v podkapitole A.2. U toho testu variační koeficient nepřesáhl hodnotu 10% a všechny naměřené hodnoty jsou tedy reprezentativní.



Obrázek 4.6: Výsledky testů pro algortimus quick-sort

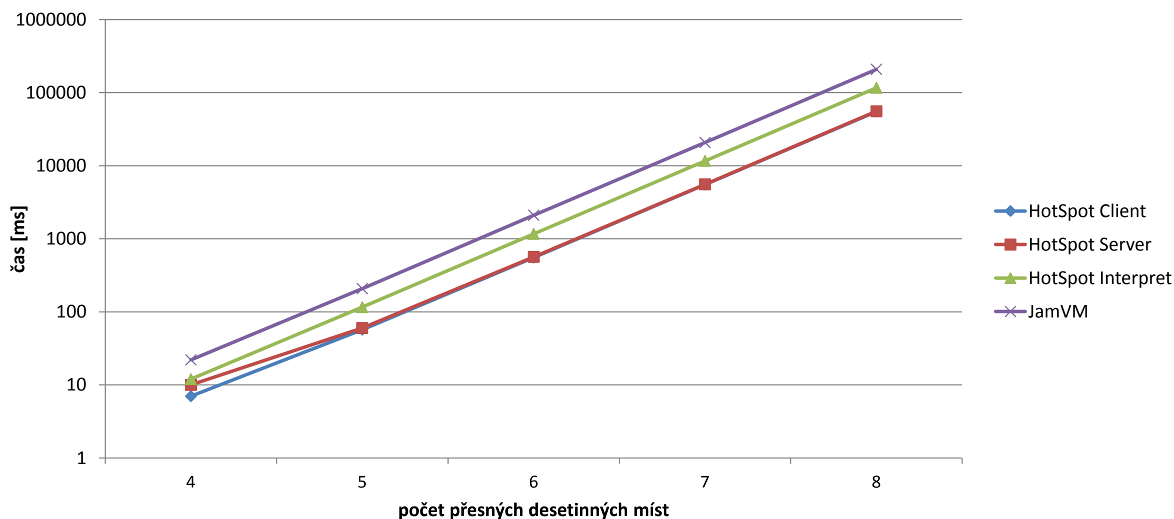
Z naměřených výsledků je zřejmé, že virtuální stroj HotSpot se zapnutým JIT kompilátorem má nad interprety jasnou výkonnostní převahu a oproti virtuálnímu stroji JamVM dosahuje až 20× vyšší rychlosti řazení. Výkonnostní rozdíl pramení z provádění programu v nativním kódu. U tohoto testu nelze tak snadno provést optimalizaci cyklu ve kterém se řadí, protože výsledek podmínky, která ho ukončuje, není snadno předvídatelný a proto je výkonnostní rozdíl mezi HotSpotem se zapnutým JIT kompilátorem menší než v předešlém testu bubble-sortu viz. 4.1. Při porovnání výsledků HotSpot interpretu a JamVM vychází, že HotSpot je přibližně 1.3× rychlejší. To je podstatně menší rozdíl nežli, u algoritmu bubble-sort. Rozdíl může být zapříčiněn nižší mírou optimalizace šablon při práci s kolekcemi a v použití pouze softwarových zásobníků při práci s constant-pooly.

4.3 Výpočet čísla π

Vyčíslení čísla π na stanovený počet desetinných míst testuje výkon virtuálního stroje pro výpočty v plovoucí desetinné čárce. Maximální přesnost osmi desetinných míst byla zvolena s ohledem na časovou náročnost tohoto testu. Pseudokód vyýpočtu čísla π je následující:

1. *Nastav proměnnou counter na 1, proměnnou act na 0, proměnnou prev na 0 a jdi na bod 2.*
2. *Vypočítej proměnnou act jako výraz* $act + (4/counter)$ *a jdi na bod 3.*
3. *přiřaď hodnotu proměnné act do proměnné prev*
4. *Vypočítej proměnnou act jako výraz* $prev + (4/(counter + 2))$ *a jdi na bod 5.*
5. *Zvyš hodnotu proměnné counter o 4 a jdi na bod 5.*
6. *Pokud je absolutní hodnota rozdílu manší nežli požadovaná přesnost, ukonči výpočet a vrať hodnotu proměnné act, jinak jdi na bod 2.*

V grafu níže jsou vynesené naměřené zprůměrované časy pro různou úroveň přesnosti výpočtu. Na ose x je vynesen počet přesných desetinných míst na které se π počítá a osa y znázorňuje čas, který je k výpočtu potřeba. Pro lepší ilustraci naměřených hodnot je časová osa v grafu znázorněna v logaritmickém měřítku o základu 10. Naměřené hodnoty použité v grafu jsou uvedeny v příloze A.3. Opět zde variační koeficient nepřesáhl hodnotu 10% a graf tedy obsahuje reprezentativní hodnoty naměřených dat.

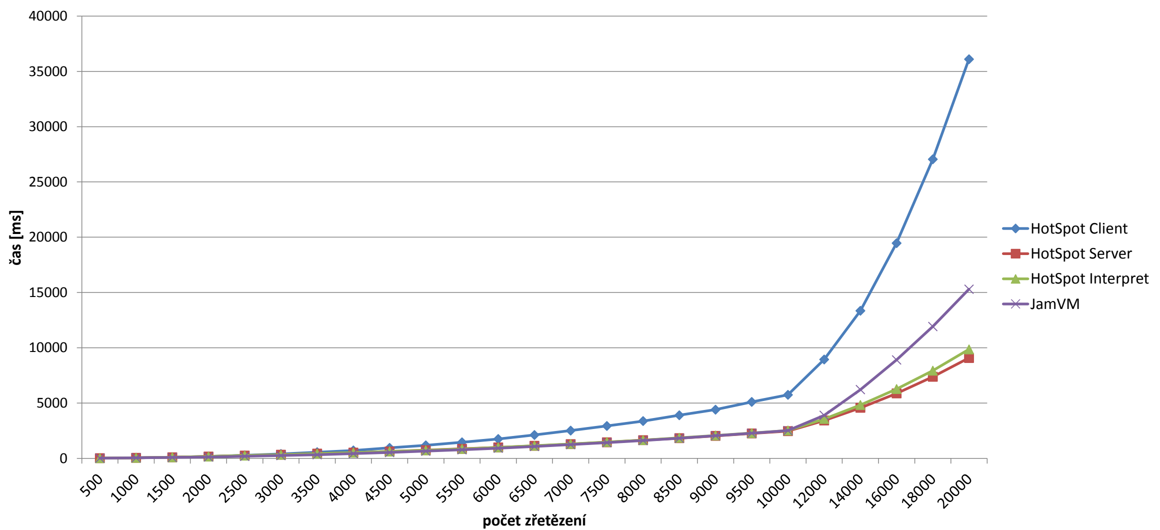


Obrázek 4.7: Výsledky testů výpočtu π

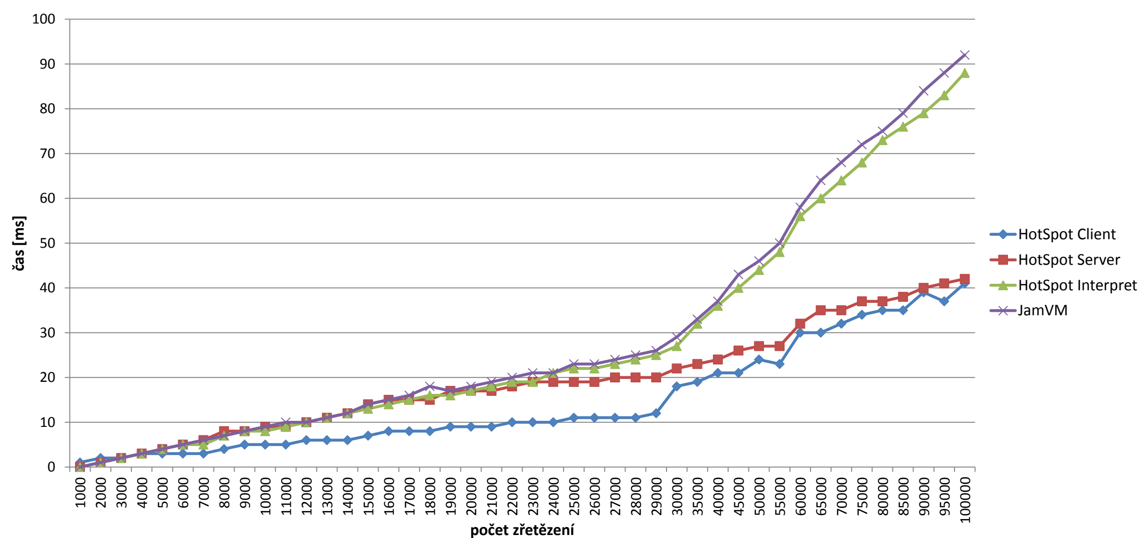
Z naměřených hodnot vyplývá, že i pro maximální přesnost osmi desetinných míst nejsou rozdíly ve výkonu jednotlivých virtuálních strojů příliš dramatické. HotSpot se zapnutým JIT kompilátorem je přibližně 3.7× rychlejší, nežli interpretovaný JamVM. Poměrně nízký rozdíl mezi JIT kompilátorem a čistým interpretem vyplývá z nemožnosti použití optimalizačních technik zaměřených na cykly, protože se konec smyčky nedá jednoduše předpovědět. Samotné výpočty jsou pro procesor také poměrně náročnou operací, protože se zde dělí desetinná čísla. Tento výpočet se provádí stejně pro interpret i pro výpočet v nativním kódu. Jediné místo kde JIT kompilátory šetří čas je práce s instrukcemi bajtkódu, kterých však není mnoho, protože výpočet π má velice krátký algoritmus (v případě testů navržených k této práci je to 5 řádků, bez započítaného cyklu).Výkonnostní rozdíl mezi HotSpot interpretem a JamVM je přibližně dvojnásobný, což odpovídá zpomalení naměřeném u iteračního algoritmu bubble-sortu, tedy výkonnostnímu rozdílu v interpretování jednotlivých instrukcí.

4.4 Zřetězení stringů

Zpracování řetězců a práce s nimi je základním kamenem serializace dat. Rychlost zpracování může mít výrazný vliv na chod programu, zejména pokud se jedná o serializovanou komunikaci se serverem. Testovány jsou dva základní způsoby zřetězení, pomocí operátoru plus(+) a pomocí třídy StringBuilder. V grafech níže jsou vyneseny naměřené hodnoty pro JamVM a různé konfigurace virtuálního stroje HotSpot. Hodnoty na ose x reprezentují počet provedených zřetězení a hodnoty na ose y čas, který by pro daný počet zřetězení potřeba.



Obrázek 4.8: Výsledky testů zřetězení stringů pomocí operátoru +



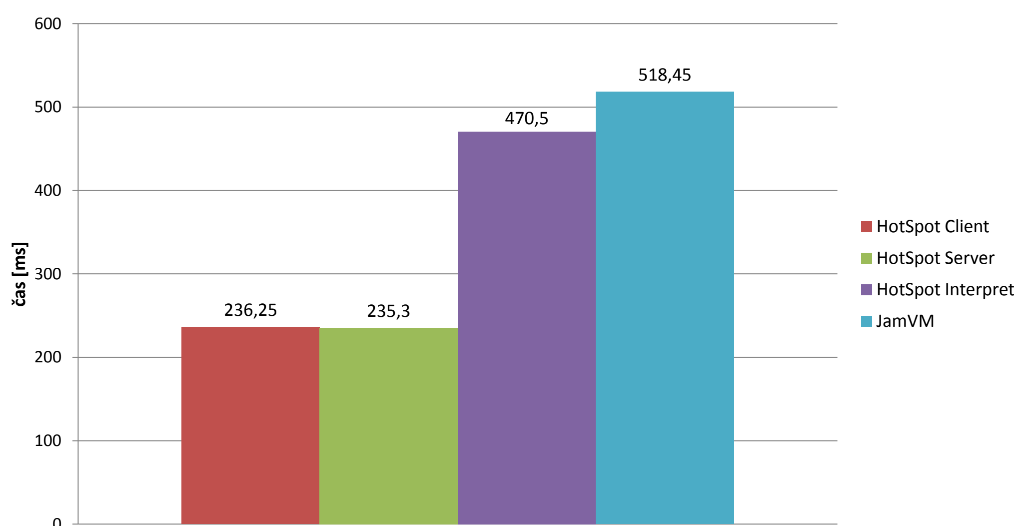
Obrázek 4.9: Výsledky testů zřetězení stringů pomocí třídy StringBuilder

Z výsledků testů, jejichž konkrétní hodnoty jsou uvedeny v příloze v podkapitole A.4, vyplývá, že použití operátoru plus (+) je pro zřetězení z výkonnostního hlediska nevhodné. Všechny virtuální stroje podávaly při jeho použití řádově horší výsledky než při použití objektu třídy StringBuilder. Největší výkonnostní propad byl zaznamenán u konfigurace HotSpotu s použitým Client JIT kompilátorem, který byl překonán i stroji s čistě interpretovaným prostředím. Tento test tedy odhalil velikou slabinu jinak zatím vždy poměrně spolehlivého a výkonného JIT kompilátoru. Při použití třídy StringBuilder podávají oba JIT kompilátory velice podobné výsledky. Nicméně Client kompilátor má po celou dobu testu mírnou převahu, což je způsobeno kratší dobou kompilace a optimální implementací kódu StringBuilderu. Ta Serveru neumožňuje provést další optimalizace, které by počáteční ztrátu kompenzovaly. Virtuální stroje JamVM a HotSpot s čistým interpretem po-

dávají taktéž vyrovnané výsledky, což je opět dáno jednoduchou a efektivní implementací StringBuilderu. Ze srovnání virtuálních strojů s čistým interpretem a strojů používajících JIT je patrné, že JIT poskytuje zhruba 2× vyšší výkon. Nicméně s přihlédnutím k počtu provedených zřetězení je možné konstatovat, že všechny virtuální stroje podávají při použití třídy StringBuilder vysoký výkon. Naměřené hodnoty při použití operátoru plus (+) jsou reprezentativní ve všech testech, jejich variační koeficient nepřesahuje 10%. Nicméně při použití třídy StringBuilder se v ojedinělých případech, zejména u menších počtů zřetězení (do 10000), vyskytují hodnoty orientační až nepřesné (variační koeficient přesáhne 40% viz. A.33, A.35, A.36). Tyto chyby jsou způsobené vysokým výkonem, který virtuální stroje podávají a pro nižší počty zřetězujících operací se testy pohybují na hranici nebo za hranicí měřitelnosti výkonu.

4.5 Test podmínek

Tento test se zaměřuje čistě na rychlost vyhodnocení několika po sobě jdoucích podmínek. Některé proměnné jsou testovány opakovaně aby se mohla projevit možná optimalizace redundantního testování hodnot. Testování jediného bloku podmínek bez použití cyklu by bylo příliš zatíženo chybou, kterou může způsobit práce operačního systému a běh programů třetích stran, z tohoto důvodu je blok podmínek v cyklu testován vícekrát. Graf níže zobrazuje průměrný čas, který byl třeba k dokončení tohoto testu.



Obrázek 4.10: Výsledky testů vyhodnocení podmínek

Výsledky testů vynesené ve výše uvedeném grafu a uvedené v příloze v podkapitole A.5 ukazují, že mezi virtuálním strojem HotSpot s použitým Client JIT kompilátorem podává naprosto stejný výkon jako HotSpot s použitým Server JIT kompilátorem. Rozdíl ve výsledcích lze vzhledem k vypočteným směrodatným odchylkám považovat za chybu měření. Podobně si proti sobě stojí oba interprety, tedy jak interpret HotSpotu tak JamVM. Ačkoli je zde naměřený rozdíl větší, tak podle velikosti vypočtených odchylek měření podávají oba stroje velice vyrovnané výkony. Testy ukázaly, že optimalizace JIT kompilátorů má na rychlost vyhodnocování podmínek poměrně velký vliv a podávají tak přibližně 2× lepší výkony. Vyšší výkon pochopitelně neplyne pouze z provedených optimalizací, ale i z použití

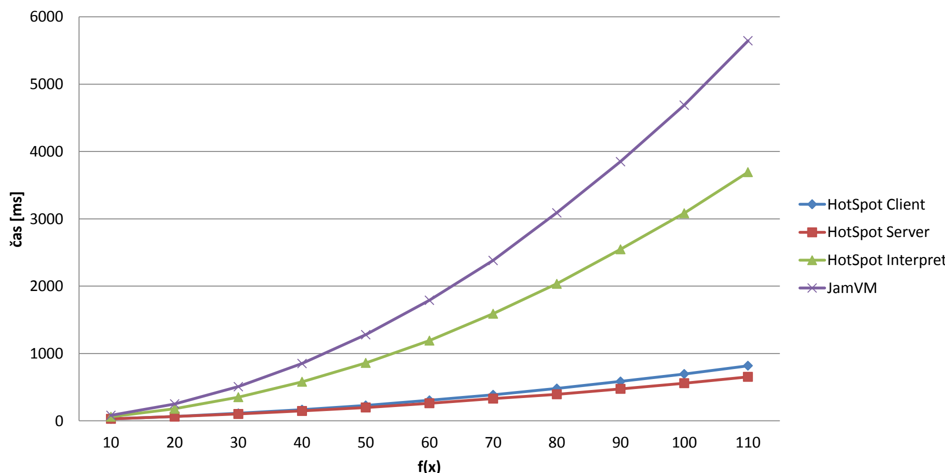
nativního kódu při jejich vyhodnocování a možnosti využití hardwarových chache pamětí počítače. Výsledek testu pro JamVM se pohybuje na hranici reprezentativnosti, jeho variační koeficient činí 11.2%, pro testy HotSpotu vychází pod 10% a naměřené hodnoty tak lze považovat za reprezentativní.

4.6 Rychlost rekurze

Tento test se zaměřuje na rychlost rekurzivního volání metod. Testuje výkon použitých mechanismů rekurze, zejména rychlost práce virtuálního stroje s vnitřním zásobníkem. Pro testování je použit algoritmus pro výpočet faktoriálu. Velikost zásobníku pro rekurzi je na virtuálních strojích poměrně limitujícím faktorem. Při návrhu testu a i při maximální možné hloubce zanoření je rekurze provedena velice rychle. Z tohoto důvodu je pro zvýšení přesnosti faktoriál postupně počítán pro každý člen posloupnosti samostatně, toto je realizováno cyklem, jehož běh se do výsledného naměřeného času nepočítá. Tímto mechanismem se zvýší počet provedených zanoření během jednoho testu. Pseudokód pro rekurzivní výpočet je následující:

```
main(param){
 for(i=0; i<10000; i++){
  for(j=1; j<=param)fact(j);
 }
}
fact(j){
  if(j==1 || j==0){
    return 1;
  }else{
    return j* fact(j-1);
  }
}
```

V grafu níže jsou znázorněny a barevně odlišeny naměřené hodnoty pro virtuální stroj JamVM a různé konfigurace virtuálního stroje HotSpot. Hodnoty na ose x představují hodnotu parametru se kterým je test spouštěn a ze kterého se potom odvíjí délka a náročnost testu. Hodnoty na ose y představují čas, který byl k vykonání daného testu potřeba.

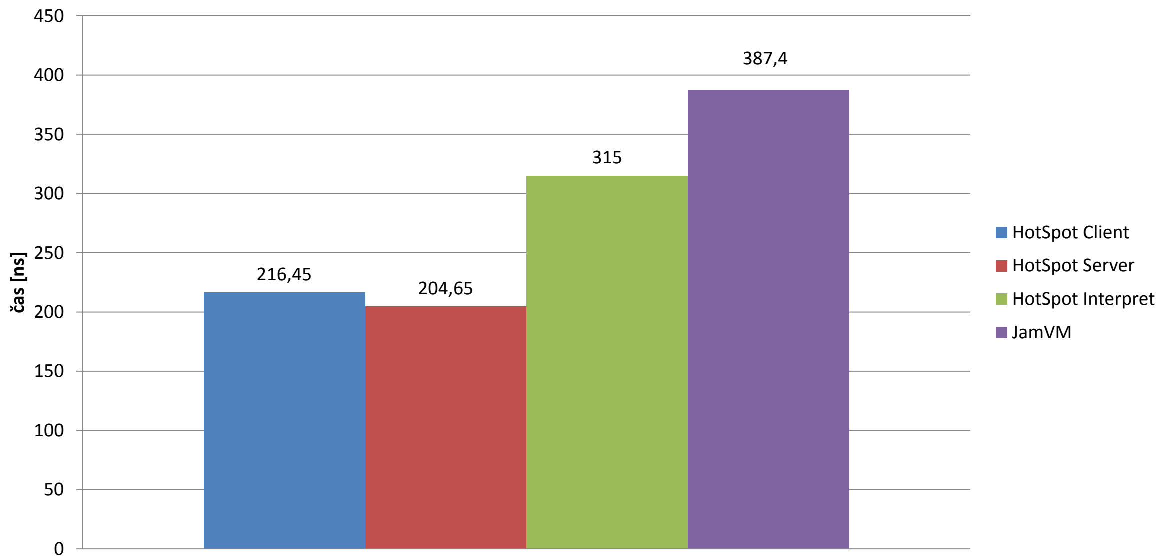


Obrázek 4.11: Výsledky testů rekurze

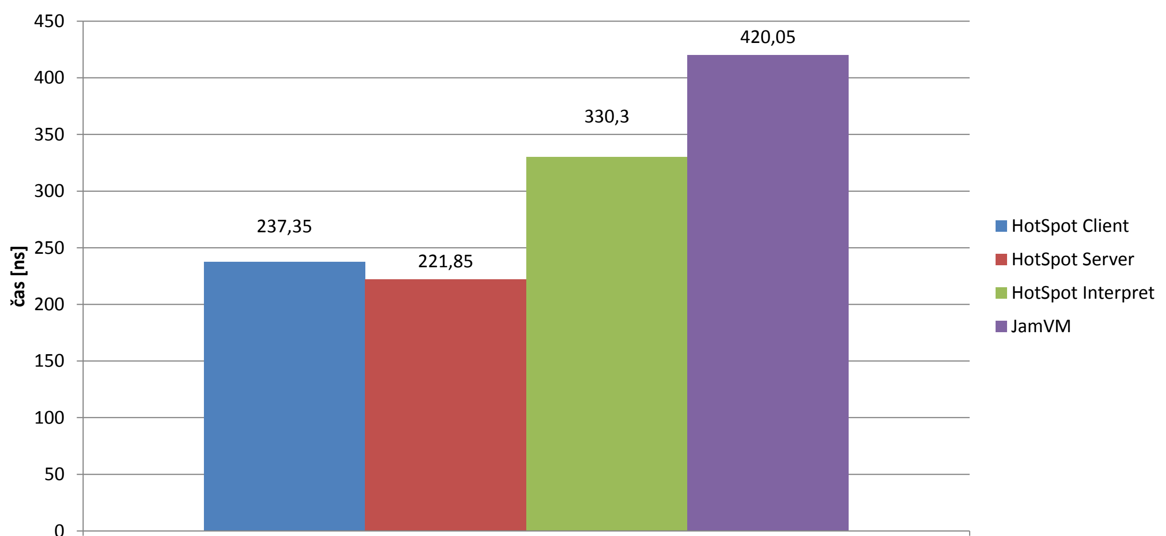
Z výsledků testu, jejichž konkrétní naměřené hodnoty jsou uvedeny v příloze v podkapitole A.6, je patrné, že i v tomto ohledu má virtuální stroj s použitým JIT kompilátorem značnou výkonnostní výhodu. Kromě rychlejšího vykonávání samotných instrukcí je to způsobeno použitím hardwarových zásobníků a cache pamětí procesoru. Softwarová implementace těchto struktur je samozřejmě pomalejší a to je jedním z hlavních důvodů proč interprety v tomto testu prohrávají. HotSpot s použitým JIT kompilátorem ať už Serverem či Clientem dosahuje více jak 6× většího výkonu oproti konkurenčnímu JamVM. Srovnání výkonu čistého interpretu HotSpotu s JamVM ukazuje, že i zde HotSpot vítězí. Vyššího výkonu může být dosaženo optimálnější implementací vnitřních datových struktur interpretu, zejména softwarového zásobníku a samozřejmě rychlejším zpracováváním bajtkódu samotného. Vynesené hodnoty v grafu je možné označit za reprezentativní. Variační koeficient naměřených hodnot nepřesáhl 10%.

4.7 Invokace metod

Pro testování rychlosti invokace metod byly použity dva testy. První z nich měří v cyklu rychlost invokace stále stejné metody, to umožní virtuálnímu stroji aby nad kódem provedl optimalizaci v podobě vložení metody přímo na místo kde je volaná. Tuto optimalizační techniku podporují dle dokumentací oba virtuální stroje. Druhý test je navržen tak aby optimalizace vkládáním možná nebyla. Toho je dosaženo tak, že se metoda invokuje vůči rozhraní jehož implementační kód se v průběhu testu mění. Invokace metody je i v neoptimalizovaném případě poměrně rychlý proces proto je použito cyklu a čas invokací je měřen několikrát. Výsledkem jednoho spuštěného testu je zprůměrovaný čas ze všech provedených invokací v nanosekundách. Testy invokace jsou tedy zaměřeny nejen na porovnání výkonu jednotlivých virtuálních strojů, ale i na srovnání účinnosti optimalizační techniky vkládání metod a jejího dopadu na výkon v rámci jednoho virtuálního stroje. V grafech níže jsou znázorněny naměřené a zprůměrované výsledky testů, které představují dobu potřebnou k invokaci jedné metody a navrácení jejího výsledku.



Obrázek 4.12: Výsledky testů invokace metod s možností optimalizace pomocí vkládání



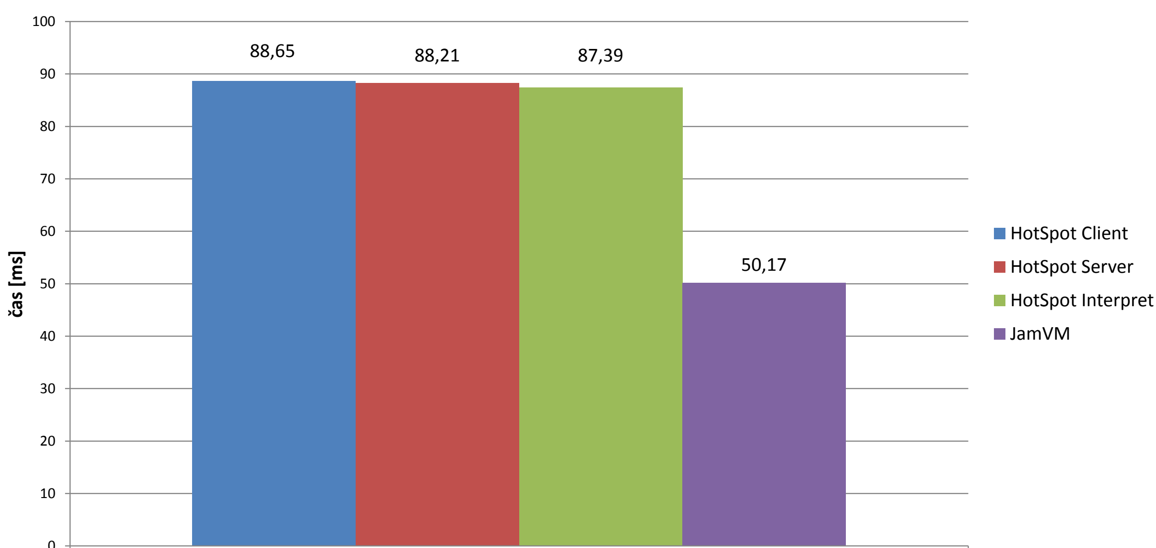
Obrázek 4.13: Výsledky testů invokace metod bez možností optimalizace pomocí vkládání

Testy invokací, jejichž přesné naměřené hodnoty jsou uvedeny v příloze v podkapitole A.7 ukázaly, že virtuální stroj HotSpot má při použití JIT kompilátoru přibližně o třetinu vyšší výkon, než při použití čistého interpretu a oproti konkurenčnímu virtuálnímu stroji JamVM má výkon takřka dvojnásobný. To platí v obou případech ať už virtuální stroje můžou používat techniku vkládání nebo ne. Zároveň se ale ukázalo, že vkládání metod celkový čas jejich provádění výrazně neurychluje. Ačkoli testy dopadly příznivěji pro případ kdy ke vkládání byly vhodné podmínky, tak nebyl nárůst výkonu nijak dramatický. U všech strojů se jedná pouze o jednotky procent. Tento malý rozdíl může být zapříčiněn dnes už velice rychlou operační pamětí, a značným množstvím cache paměti procesoru, které mají na rychlost invokace metody podstatný vliv. Na starších strojích by tedy mohly testy

dopadnout výrazně jinak. Hodnoty uvedené v grafech jsou opět reprezentativní. Hodnota variačního koeficientu u naměřených dat nepřesáhla 10%.

4.8 Start virtuálního stroje

Tento test prověřuje rychlost spuštění virtuálního stroje. Měří se doba od spuštění shellovského příkazu po získání návratové hodnoty z prázdné main funkce obsahující pouze příkaz return. Z principu použité metody nemůže být výsledek testu zcela přesný, protože vykonání příkazu return a navrácení návratové hodnoty systému zabere část naměřeného času. Pro porovnání jednotlivých virtuálních strojů je ale tato technika dostačující a poměrně dobře ilustruje jak rychle si virtuální stroje poradí s kratšími scripty a kolik času z celkové doby od spuštění programu zabere spuštění virtuálního stroje a načtení hlavní třídy. Graf níže znázorňuje průměrnou dobu stratu jednotlivých virtuálních strojů.

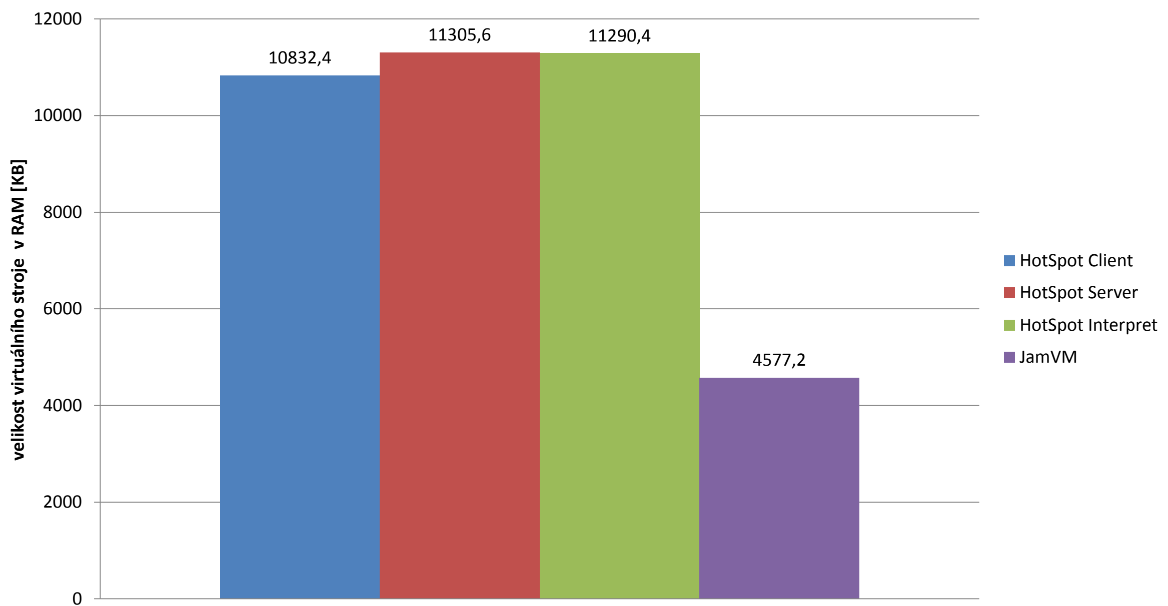


Obrázek 4.14: Výsledky testů rychlosti startu virtuálních strojů

Test rychlosti startu je jeden ze tří testů, který se neorientuje na výpočetní výkon virtuálního stroje. Z výsledků testů vynesencých v grafu a uvedených v příloze v podkapitole A.8, je vidět, že JamVM výrazně překonává HotSpot a to v jakékoli jeho konfiguraci. Naměřené hodnoty všech tří konfigurací HotSpotu jsou v rámci odchylky měření stejné. JamVM tedy všechny překonává přibližně stejným rozdílem, délka jeho stratu je o 44% kratší, než u HotSpotu. Důvodem rychlejšího startu je především konstrukce interpretu HotSpotu, který se používá jak u konfigurací se zapnutým JIT kompilátorem tak bez něj. Jak je uvedeno v podkapitole 3.1 teoretického srovnání, interpret se u HotSpotu generuje do operační paměti z připravených šablon a tato operace je výpočetně i časově náročná. Použití JamVM u krátkých programů či skriptů tedy může značně urychlit jejich vykonání a šetří systémové prostředky. Variační koeficient naměřených hodnot je do 10%, lze je tedy považovat za reprezentativní.

4.9 Velikost virtuálního stroje

Cílem tohoto testu bylo změřit velikost samotného virtuálního stroje v operační paměti. Vzhledem k tomu, že na některý platformách či integrovaných systémech kde se Java provozuje nejsou k dispozici výpočetní zdroje srovnatelné se současnými počítači a kapacita operační paměti může být velice omezená. Je poměrně zajímavé podívat se na to, jak s cennými kilobajty jednotlivé virtuální stroje naloží. Zejména virtuální stroj JamVM byl navržen tak aby s pamětí šetřil, test tedy ukáže nakolik se autorovi povedlo svého cíle dosáhnout. Naměřené hodnoty představují velikost virtuálního stroje v operační paměti při jeho startu, velikost alokované haldy se do nich nepočítá. V následující grafu jsou znázorněny průměrné naměřené velikosti virtuálních strojů v operační paměti.



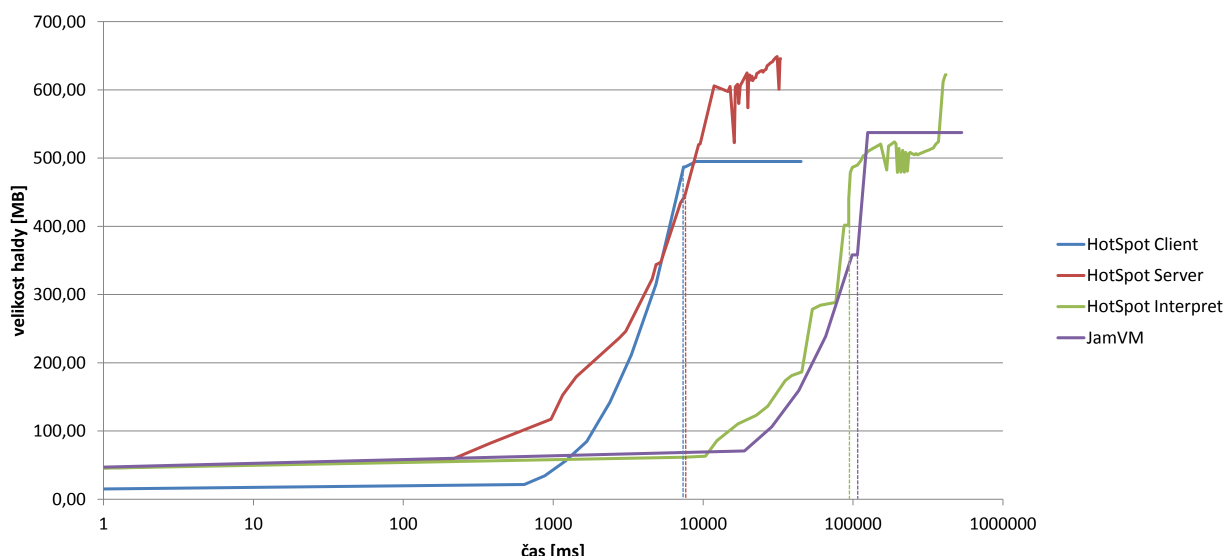
Obrázek 4.15: Výsledky měření velikosti virtuálních strojů

Z výše uvedeného grafu i z tabulky naměřených hodnot uvedených v příloze v podkapitole A.9, je patrné, že velikosti obou konkurenčních virtuálních strojů se velice liší. Zatím co, změna konfigurace HotSpotu má na výslednou velikost stroje jen malý vliv tak při použití JamVM ušetříme až 60% operační paměti. Hlavní podíl na její úspoře má minimalistický návrh virtuálního stroje JamVM a také použitý programovací jazyk.

4.10 Práce s haldou

Posledním, ze série testů je monitorování haldy při řazení pole o velikost 10 milionů prvků pomocí již zmíněného quick-sortu, jehož test je uveden v podkapitole 4.2. Práce automatického správce paměti (anglicky označovaný jako: garbage collector) je při běhu Javovských programů důležitá. Jako každý proces běžící na pozadí zabírá i tento správce paměti při své práci systémové prostředky a to především procesorový čas, čímž může snížit výpočetní výkon virtuálního stroje. Hlavní nevýhodou automatického správce paměti však nadále zůstává nutnost čas od času pozastavit vykonávání spuštěného programu během „úklidu“ paměti a to i u moderně navržených správců paměti, běžících v samostatném vlákně pro-

gramu. Čím častěji tedy správce paměti uklízí, tím pravděpodobnější je, že se běh programu na okamžik pozastaví. Automatická správa paměti má však i své výhody, jednou z nich je usnadnění práce programátorovi, druhá snad ještě podstatnější je zabránění úniku paměti (anglicky označovaného jako memory leak). Ani automatická správa paměti však neřeší problém úniku paměti úplně a při nesprávné kombinaci statických instancí a vnořených tříd je možné haldu na celý zbytek běhu virtuálního stroje obsadit nepotřebnými daty. Průběžné měření velikosti haldy probíhá tak, že se ihned po startu testu spustí nové vlákno, které každou milisekundu pomocí *java.lang.Runtime* třídy čte aktuální velikost haldy. Pokud se její velikost liší od naposledy zaznamenané hodnoty alespoň o 1 MB, je tato hodnota také zaznamenána. Monitorující vlákno je spuštěno ihned po zpracování parametrů programu a běží i v době kdy jsou ze souboru načítány hodnoty do pole, které se následně bude řadit. V grafu níže jsou znázorněny průběžné velikosti hald jednotlivých virtuálních strojů v čase. Osa x představuje čas po který virtuální stroje běží a hodnoty na ose y představují velikost haldy kterou disponují. V grafu jsou také pomocí svislých čar vyznačeny body ve kterých bylo načtení dat ze souboru dokončeno a program začal pole seřazovat. Pro lepší názornost grafu, je časová osa vykreslena v logaritmickém měřítku o základu 10.



Obrázek 4.16: Výsledky měření práce s heapem virtualních strojů

Výsledky měření heapu, jejichž přesné výsledky jsou uvedeny v příloze A.10, ukazují, že různé konfigurace HotSpotu používají různé správce paměti. Virtuální stroj HotSpot se Server JIT kompilátorem a v režimu čistého interpretu používá konfiguraci automatického správce paměti, která haldu uklízí častěji a důkladněji. Z haldy jsou v průběhu výpočtu dealokovány i menší bloky dat. U virtuálního stroje HotSpot s nastaveným Client JIT kompilátorem k úklidu paměti během řazení nedocházelo a halda se pouze s rostoucími potřebami virtuálního stroje zvětšovala. V případě virtuálního stroje JamVM jsou výsledky testů velice podobné konfiguraci HotSpotu s Clitent JIT kompilátorem. Odlišný je také způsob navyšování celkové velikosti haldy, HotSpot s Client JIT kompilátorem a v režimu čistého interpretu realokují paměť po menší blocích a častěji, HotSpot s Client JIT kompilátorem a JamVM realokují paměť méně často a po větších paměťových blocích. Chování HotSpotu s Client JIT kompilátorem a JamVM šetří výpočetní výkon počítače, nicméně

halda je v někdy větší než by musela být. HotSpot se Server JIT kompilátorem či jako čistý interpret se tedy k operační paměti chovají úsporněji a neplýtvají zbytečně nevyužitými zdroji.

Měření velikosti haldy není z principu možné několikrát po sobě zopakovat tak aby se spolu daly jednotlivé naměřené hodnoty porovnat a vypočítat z nich odchylku měření. Práce s heapem je ovlivněna i aktuálním stavem operační paměti a délkou výpočtu který se může u jednotlivých testů mírně lišit a časové hodnoty jednotlivých naměřených bodů si nemusí odpovídat. Z toho důvodu jsou výsledky tohoto testu jen ilustrační.

4.11 Shrnutí

Výsledky testů všeobecně potvrdily výkonnostní převahu virtuálního stroje HotSpot, který zejména při použití JIT kompilátorů svého konkurenta JamVM výkonnostně několikanásobně převyšuje. Srovnání čistých interpretů taktéž prokázalo, výkonnostní převahu HotSpotu, kde se pozitivně projevila šablonová konstrukce interpretu. Nicméně výkonnostní rozdíly mezi interprety již byly podstatně menší. Virtuální stroj JamVM potvrdil svou kvalitu při testu provozních vlastností, kde HotSpot výrazně překonal jak v rychlosti svého startu tak v paměťové (ne)náročnosti.

Kapitola 5

Závěr

Bakalářská práce se věnuje srovnáním dvou konkurenčních virtuálních strojů, virtuálního stroje HotSpot a virtuálního stroje JamVM. Nejprve jsem shrnul nejdůležitější vlastnosti Javy a virtuálních strojů, které používá. Nastínil jsem problematiku výkonnosti zpracovávání bajtkódu a zmínil některé optimalizační techniky, které se nejčastěji používají. V teoretickém srovnání virtuálních strojů byly popsány vlastnosti, na kterých si jejich vývojáři zakládají nejvíce a které by měly být pro daný virtuální stroj typické. Podle prostudovaných technologií a jejich vlastností, které jsou v implementaci virtuálních strojů použity, byla navržena sada výkonnostních testů.

Podle výsledků výkonnostních testů, které jsem v rámci této práce naimplementoval a provedl, se ukázalo, že virtuální stroj HotSpot má nad konkurenčním JamVM z hlediska čistého výpočetního výkonu značnou převahu. Vysoký výpočetní výkon byl jedním ze základních požadavků při vývoji HotSpotu, takže se dá bez jakýchkoli pochybností konstatovat, že se vývojářům jejich cíle podařilo dosáhnout. Virtuální stroj JamVM byl navržen tak, aby byl jeho výkon dostatečný pro běžné použití, ale hlavně aby byl minimalistický a mezi platformami snadno přenositelný. Vzhledem k tomu, že v operační paměti zabírá kód virtuálního stroje JamVM pouze 40% velikosti HotSpotu, není o jeho minimalističnosti pochyb. Z hlediska výpočetního výkonu na tom JamVM není také nikterak špatně. Až na testy bubble-sortem, což je algoritmus, který se nedá označit za vhodný pro řazení jakkoli velkého či malého vzorku dat, si se všemi testy, které se by se daly snadno označit za extrémní, dokázal v rozumném výpočetním čase poradit. Co se přenositelnosti týče i zde vývojář svůj cíl splnil, podpora různých hardwarových platforem je širší nežli u HotSpotu a portace na nové platformy je podstatně jednodušší. Na závěr lze tedy doporučit použití virtuálního stroje HotSpot tam, kde je dostatek systémových prostředků a kde je třeba vysokého výpočetního výkonu. Použití JamVM je vhodné na exotických a méně výkonných platformách, které disponují omezeným množstvím operační paměti.

Tato práce by se dala dále rozšířit o testování běhu virtuálních strojů na dalších hardwarových platformách, jako je například platforma ARM. Další možností rozšíření je testování výkonu virtuálních strojů při použití více vláken, to by ale vyžadovalo návrh dalších testů.

Literatura

[1] Corporation, O.: Java EE at a Glance.
`http://www.oracle.com/technetwork/java/javaee/overview/index.html`, [cit. 2014-02-03].

[2] Corporation, O.: Java ME and Java Card Technology.
`http://www.oracle.com/technetwork/java/javame/index.html`, [cit. 2014-02-03].

[3] Corporation, O.: Java SE at a Glance.
`http://www.oracle.com/technetwork/java/javase/overview/index.html`, [cit. 2014-02-03].

[4] Herout, P.: *Učebnice Jazyka Java*. Kopp, 2008, iSBN 978-80-7232-355-5.

[5] Lougher, R.: JamVM. `http://jamvm.sourceforge.net/`, 2010-02-01 [cit. 2014-03-09].

[6] Spell, B.: *Java Programujeme profesionálně*. Computer Press, 2002, iSBN 80-7226-667-5.

[7] Tim Lindhon, F. Y.: *The Java Virtual Machine Specification Second Edition*. Addison-Wesley, 2003, iSBN 0-201-43294-3.

Příloha A

Naměřené hodnoty

A.1 Výsledky testů pro algoritmy bubble-sort

Tabulka A.1: Tabulka naměřených hodnot pro bubble-sort s použitým for cyklem, spuštěný na JamVM

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	73	75	74	0,63
1000	297	302	299	1,72
1500	680	683	681	1,01
2000	1222	1228	1223	2,24
2500	1918	1928	1922	3,66
3000	2783	2809	2791	9,13
3500	3859	3870	3863	4,70
4000	5061	5075	5067	5,54
4500	6397	6431	6410	11,65
5000	7903	8436	8019	208,65
5500	9473	9482	9478	3,28
6000	11223	11330	11280	40,39
6500	13219	13287	13260	25,79
7000	15332	15453	15377	52,92
7500	17568	17763	17633	71,21
8000	20137	20594	20295	172,56
8500	22732	22895	22834	57,11
9000	25319	25451	25370	49,54
9500	28875	29040	28970	62,13
10000	31595	31773	31696	57,49
12000	45472	48642	46663	1339,65
14000	61469	61834	61691	119,88
16000	80677	81162	80858	180,89
18000	103011	103554	103201	192,55
20000	126842	132345	129067	1865,06

Tabulka A.2: Tabulka naměřených hodnot pro bubble-sort s použitým for cyklem, spuštěný na HostSpot (Interpret only)

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	41	41	41	0
1000	166	169	167	1,16
1500	377	379	377	0,74
2000	668	673	671	1,83
2500	1044	1053	1047	3,49
3000	1504	1513	1508	3,13
3500	2049	2054	2050	1,93
4000	2666	2693	2676	9,52
4500	3369	3471	3395	38,05
5000	4145	4165	4154	6,62
5500	5005	5052	5028	19,74
6000	5950	5993	5968	16,13
6500	6984	7030	7000	16,44
7000	8092	8170	8123	25,99
7500	9249	9412	9311	61,22
8000	10567	10736	10625	58,82
8500	11935	11974	11955	13,25
9000	13369	13477	13404	38,85
9500	14887	14989	14940	41,96
10000	16507	16845	16595	125,77
12000	23845	24007	23929	66,81
14000	32311	32416	32380	37,42
16000	42144	42414	42325	99,15
18000	53384	53564	53486	61,36
20000	66021	66254	66129	88,55

Tabulka A.3: Tabulka naměřených hodnot pro bubble-sort s použitým for cyklem, spuštěný na HotSpot(Client kompilátor)

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	9	10	9	0,48
1000	24	29	26	1,62
1500	46	54	49	3,26
2000	76	82	78	2,09
2500	106	116	111	3,34
3000	143	156	149	4,58
3500	189	201	195	4,74
4000	240	250	244	3,70
4500	301	319	309	6,01
5000	364	401	378	12,92
5500	440	466	454	9,69

Pokračování na následující straně

Tabulka A.3 – *Pokračování z předešlé strany*

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
6000	519	553	541	12,27
6500	611	651	634	15,26
7000	718	737	729	7,22
7500	830	865	848	11,16
8000	935	979	960	15,78
8500	1046	1141	1089	32,21
9000	1199	1256	1219	21,99
9500	1350	1421	1376	24,18
10000	1471	1522	1495	17,67
12000	2143	2205	2168	22,43
14000	2852	3154	2942	109,31
16000	3841	4068	3896	86,14
18000	4848	5225	4974	131,96
20000	6130	6150	6138	6,85
22000	7344	7754	7498	136,78
24000	8759	8969	8888	71,05
26000	10414	10586	10525	63
28000	12282	12315	12295	12,44
30000	14135	15024	14328	348,51
32000	16243	17356	16591	391,29
34000	18608	18804	18720	67,40
36000	21102	21413	21205	114,26
38000	23595	23801	23697	71,59

Tabulka A.4: Tabulka naměřených hodnot pro bubble-sort s použitým for cyklem, spuštěný na HotSpot(Server kompilátor)

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	23	193	75	59,87
1000	54	65	57	4,35
1500	71	79	74	2,92
2000	83	94	87	4,49
2500	109	119	113	4,07
3000	152	158	155	1,95
3500	194	202	198	3,03
4000	245	255	250	4,07
4500	307	308	307	0,48
5000	375	387	381	4,07
5500	447	478	459	14,46
6000	541	561	552	6,52
6500	612	634	622	7,55
7000	729	741	734	4,33
7500	810	815	813	1,89

Pokračování na následující straně

Tabulka A.4 – *Pokračování z předešlé strany*

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
8000	933	976	944	15,80
8500	1097	1109	1103	4,17
9000	1171	1188	1179	6,46
9500	1317	1332	1323	5,62
10000	1483	1491	1487	3,57
12000	2076	2095	2087	6,61
14000	2724	2922	2772	75,10
16000	3620	3778	3684	71,94
18000	4507	4710	4555	77,79
20000	5515	5535	5524	7,84
22000	6624	6715	6679	35,68
24000	7759	7959	7847	89,42
26000	9247	9328	9280	28,29
28000	10399	10620	10564	83,27
30000	11907	12052	11971	49,76
32000	13546	14178	13876	277,59
34000	15268	15898	15693	221,38
36000	17632	17748	17690	45,67
38000	19554	19824	19644	108,07

Tabulka A.5: Tabulka naměřených hodnot pro bubble-sort s použitým while-do cyklem a post inkrementací, spuštěný na JamVM

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	73	75	74	0,632455532
1000	297	301	299	1,41
1500	680	684	681	1,74
2000	1218	1231	1225	5,35
2500	1917	1925	1921	2,56
3000	2794	2851	2811	20,38
3500	3851	4267	3944	161,69
4000	5057	5098	5074	13,37
4500	6391	6448	6416	20,52
5000	7910	7960	7934	18,01
5500	9460	9496	9486	13,43
6000	11266	11321	11298	24,82
6500	13299	13373	13329	26,35
7000	15348	16474	15594	440,02
7500	17644	17705	17667	27,24
8000	20233	20465	20314	85,36
8500	22896	23055	22971	62,46
9000	25507	25722	25574	77,44
9500	28401	28639	28516	85,80

Pokračování na následující straně

Tabulka A.5 – *Pokračování z předešlé strany*

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
10000	31842	32378	32040	191,90
12000	46159	49785	48256	1213,82
14000	63171	65199	63676	767,47
16000	82738	87614	84748	2136,45
18000	107646	111228	109262	1476,03
20000	132599	133318	133017	251,93

Tabulka A.6: Tabulka naměřených hodnot pro bubble-sort s použitým while-do cyklem a post inkrementací, spuštěný na HotSpot (Interpret only)

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	41	41	41	0
1000	166	171	168	1,72
1500	378	382	379	1,50
2000	669	671	669	0,8
2500	1051	1177	1096	53,1
3000	1502	1764	1556	104,01
3500	2041	2429	2196	188,78
4000	2655	2807	2694	56,58
4500	3356	3374	3363	6,24
5000	4124	4270	4160	55,10
5500	4982	5027	5002	15,46
6000	5937	7652	6385	659,99
6500	6951	8127	7201	463,22
7000	8054	10860	8657	1102,43
7500	9257	10345	9516	418,39
8000	10532	11890	11166	537,13
8500	11892	12056	11958	54,03
9000	13345	17768	15866	1698,33
9500	14888	17540	15815	1021,62
10000	16449	21160	18930	2083,58
12000	23879	28831	26492	1706,20
14000	32521	39607	35130	2482,03
16000	42300	50311	45333	2963,12
18000	53652	59194	55916	2529,67
20000	66477	72409	70274	2020,64

Tabulka A.7: Tabulka naměřených hodnot pro bubble-sort s použitým while-do cyklem a post inkrementací, spuštěný na HotSpot (Client kompilátor)

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	9	12	10	1,356465997
1000	24	24	24	0
1500	45	51	46	2,4
2000	74	82	77	2,73
2500	102	109	106	2,58
3000	142	150	144	2,71
3500	183	193	189	4,13
4000	240	245	243	1,90
4500	301	369	318	25,57
5000	355	370	365	5,38
5500	444	457	449	4,34
6000	524	538	531	4,98
6500	618	626	621	3,07
7000	713	763	729	17,42
7500	813	831	822	6,93
8000	927	992	944	24,86
8500	1051	1075	1063	8,36
9000	1192	1282	1218	32,44
9500	1304	1317	1312	4,47
10000	1447	1532	1470	31,51
12000	2094	2142	2119	17,07
14000	2851	2994	2902	51,96
16000	3758	4020	3843	91,05
18000	4822	5103	4931	118,33
20000	5920	6011	5961	31,31
22000	7117	7263	7203	48,81
24000	8361	8641	8543	97,66
26000	9265	10391	10067	410,15
28000	11987	12409	12160	148,82
30000	12883	13917	13337	350,32
32000	14600	15685	15207	394,52
34000	16527	18018	17205	563,95
36000	18725	19888	19322	375,74
38000	21467	22476	21867	371,96

Tabulka A.8: Tabulka naměřených hodnot pro bubble-sort s použitým while-do cyklem a post inkrementací, spuštěný na HotSpot (Server kompilátor)

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	25	69	37	16,14

Pokračování na následující straně

Tabulka A.8 – *Pokračování z předešlé strany*

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
1000	54	68	60	5,11
1500	76	89	82	5,31
2000	95	106	99	3,90
2500	114	118	116	1,6
3000	152	153	152	0,4
3500	194	196	194	0,8
4000	244	256	247	4,53
4500	298	334	309	12,60
5000	373	375	374	0,75
5500	438	554	477	41,96
6000	530	602	551	26,30
6500	615	790	662	64,39
7000	700	735	720	12,91
7500	809	1016	863	78,85
8000	931	973	957	17,30
8500	1047	1251	1098	78,39
9000	1176	1456	1241	107,66
9500	1357	1547	1433	72,94
10000	1417	1512	1460	38,02
12000	2063	2092	2076	9,72
14000	2727	2993	2826	108,20
16000	3605	3872	3728	91,43
18000	4516	4790	4671	107,73
20000	5500	6073	5776	190,46
22000	6817	7026	6889	79,26
24000	7917	8611	8070	271,15
26000	9213	9759	9538	203,63
28000	10385	11031	10677	213,71
30000	12376	13170	12815	295,12
32000	13872	14651	14338	262,03
34000	15819	17072	16629	458,90
36000	18513	19788	19015	445,25
38000	20003	22026	20936	683,35

Tabulka A.9: Tabulka naměřených hodnot pro bubble-sort s použitým while-do cyklem a pre inkrementací, spuštěný na JamVM

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	74	92	77	7,2
1000	298	301	299	1,47
1500	681	686	682	1,94
2000	1221	1232	1224	3,87
2500	1973	1984	1977	3,83

Pokračování na následující straně

Tabulka A.9 – *Pokračování z předešlé strany*

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
3000	2792	2804	2798	4,41
3500	3869	3893	3882	8,44
4000	5021	5055	5040	13,75
4500	6365	6371	6367	2,8
5000	7880	8288	8024	162,78
5500	9604	10814	10072	453,73
6000	11374	11464	11417	33,19
6500	13279	15746	13834	960,46
7000	15356	15518	15420	58,11
7500	17700	17820	17737	42,99
8000	20192	20840	20438	259,61
8500	22929	25097	23805	710,56
9000	25945	28194	26829	752,14
9500	28829	31654	29780	1154,12
10000	31741	32232	31898	176,05
12000	46618	49858	48408	1320,43
14000	63367	66653	64288	1201,12
16000	83223	90977	86340	2550,62
18000	107558	112237	109400	1604,14
20000	134157	146727	138062	4774,19

Tabulka A.10: Tabulka naměřených hodnot pro bubble-sort s použitým while-do cyklem a pre inkrementací, spuštěný na HotSpot (Interpret Only)

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	41	42	41	0,4
1000	165	167	166	0,75
1500	378	426	398	21,93
2000	669	717	686	19,28
2500	1046	1080	1056	12,24
3000	1510	1593	1530	31,47
3500	2076	2089	2082	4,88
4000	2680	2763	2704	29,73
4500	3361	3541	3420	62,36
5000	4155	4460	4256	107,45
5500	4995	5422	5105	159,64
6000	5977	6300	6087	111,16
6500	7041	7656	7286	224
7000	8233	8450	8336	72,21
7500	9405	9661	9532	96,23
8000	10707	11489	11067	279,55
8500	11999	12574	12243	203,55
9000	13718	14468	14004	265,37

Pokračování na následující straně

Tabulka A.10 – *Pokračování z předešlé strany*

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
9500	15211	15753	15395	190,87
10000	16613	17159	16805	194,07
12000	24607	26509	25541	676,26
14000	33093	34883	33795	590,31
16000	42828	45940	44369	1197,19
18000	53980	62732	58406	3161,41
20000	67062	76108	71710	3328,82

Tabulka A.11: Tabulka naměřených hodnot pro bubble-sort s použitým while-do cyklem a pre inkrementací, spuštěný na HotSpot (Client kompilátor)

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	9	10	9	0,4
1000	23	25	23	0,8
1500	43	49	45	2,24
2000	70	76	72	2,24
2500	100	107	103	2,24
3000	137	147	141	3,54
3500	183	190	187	2,37
4000	229	237	234	2,94
4500	289	298	294	3,01
5000	352	362	358	3,50
5500	438	448	441	3,56
6000	513	521	517	2,71
6500	585	599	594	4,96
7000	683	696	690	4,34
7500	778	808	797	10,37
8000	903	924	915	7,06
8500	1032	1046	1040	4,94
9000	1150	1174	1164	9,54
9500	1264	1315	1285	16,94
10000	1434	1442	1438	2,83
12000	2025	2085	2057	19,17
14000	2781	2840	2808	18,79
16000	3642	3743	3710	35,08
18000	4681	4794	4761	41
20000	5791	5886	5839	38,22
22000	7101	7352	7222	88,59
24000	8425	8674	8561	99,11
26000	10128	10249	10185	49,22
28000	11757	12020	11868	88,35
30000	13503	13735	13632	80,77
32000	15502	15831	15708	110,22

Pokračování na následující straně

Tabulka A.11 – *Pokračování z předešlé strany*

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
34000	17393	18258	17802	293,03
36000	19793	20080	19939	103,29
38000	19176	22431	20400	1363,48

Tabulka A.12: Tabulka naměřených hodnot pro bubble-sort s použitým while-do cyklem a pre inkrementací, spuštěný na HotSpot (Server kompilátor)

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	26	60	37	12,26
1000	61	65	63	1,41
1500	81	87	84	2,32
2000	91	110	102	6,43
2500	108	119	111	4,22
3000	152	159	154	2,48
3500	197	215	204	6,12
4000	247	254	249	2,61
4500	299	305	302	2,48
5000	374	388	381	4,58
5500	463	505	477	15,07
6000	560	580	572	6,97
6500	640	698	670	20,26
7000	735	793	770	20,10
7500	823	892	854	23,16
8000	962	1017	1001	20,37
8500	1122	1224	1171	37,84
9000	1193	1206	1199	4,66
9500	1318	1380	1359	22,18
10000	1485	1591	1535	37,66
12000	2084	2353	2200	101,43
14000	2735	3035	2920	101,87
16000	3614	4118	3917	190,68
18000	4536	5185	4739	245,40
20000	5484	5932	5630	162,99
22000	6630	6908	6781	88,59
24000	7831	8151	8000	124,36
26000	9193	9555	9392	134,01
28000	10449	11154	10878	264,75
30000	12186	12795	12408	205,45
32000	14369	14820	14574	145,54
34000	16281	16668	16496	154,14
36000	17686	19073	18637	507,68
38000	20112	20738	20394	259,39

Tabulka A.13: Tabulka naměřených hodnot pro bubble-sort s použitým do-while cyklem a post inkrementací, spuštěný na JamVM

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	74	75	74	0,4
1000	301	302	301	0,49
1500	687	690	688	1,17
2000	1227	1235	1231	3,03
2500	1941	1998	1963	23,39
3000	2822	2973	2855	58,69
3500	3896	3936	3913	14,36
4000	5027	5150	5062	44,69
4500	6327	6377	6343	17,45
5000	7842	7872	7856	10,83
5500	9549	10591	9768	411,71
6000	11376	11793	11467	162,71
6500	13338	13568	13406	83,44
7000	15500	15619	15550	46,10
7500	17784	18001	17865	73,49
8000	20134	20211	20186	30,85
8500	22722	22981	22818	90,43
9000	25741	27390	26092	649,88
9500	28577	29138	28861	221,86
10000	31642	31889	31709	91,84
12000	46067	46616	46267	189,10
14000	63018	64809	63503	658,74
16000	83447	84510	83965	407,54
18000	106402	109402	107194	1126,15
20000	139602	145855	142339	2252,87

Tabulka A.14: Tabulka naměřených hodnot pro bubble-sort s použitým do-while cyklem a post inkrementací, spuštěný na HotSpot (Interpret only)

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	41	42	41	0,4
1000	166	170	167	1,50
1500	378	380	378	0,8
2000	667	700	675	12,57
2500	1042	1054	1047	4,72
3000	1508	1556	1521	17,78
3500	2046	2112	2061	25,53
4000	2658	3300	2789	255,51
4500	3358	3467	3386	40,45
5000	4131	4790	4265	262,50

Pokračování na následující straně

Tabulka A.14 – *Pokračování z předešlé strany*

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
5500	4998	6424	5302	561,37
6000	5952	6821	6165	334,84
6500	6957	7083	6997	44,34
7000	8091	9670	8788	632,25
7500	9247	11519	9748	886,06
8000	10533	10636	10592	39,05
8500	11903	12664	12107	283,57
9000	13360	15725	14383	922,46
9500	14853	20005	16240	1951,53
10000	16491	19649	18131	1199,37
12000	23769	26609	24877	1137,49
14000	32367	38133	34515	2406,91
16000	42174	48215	46136	2175,54
18000	55455	68174	60956	4765,08
20000	67201	73930	70863	2165,23

Tabulka A.15: Tabulka naměřených hodnot pro bubble-sort s použitým do-while cyklem a post inkrementací, spuštěný na HotSpot (Client kompilátor)

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	9	10	9	0,49
1000	25	27	25	0,75
1500	43	46	43	1,17
2000	66	67	66	0,4
2500	94	96	95	0,63
3000	127	129	127	0,8
3500	168	181	171	4,84
4000	217	219	217	0,8
4500	269	290	274	7,79
5000	329	354	335	9,39
5500	403	405	404	0,75
6000	474	479	475	1,85
6500	554	593	562	15,14
7000	642	864	687	88,31
7500	741	747	742	2,14
8000	842	854	845	4,53
8500	953	956	954	0,98
9000	1065	1070	1067	1,67
9500	1181	1270	1199	35,21
10000	1309	1406	1330	37,94
12000	1891	1898	1893	2,42
14000	2574	2618	2583	17,21
16000	3384	3427	3394	16,32

Pokračování na následující straně

Tabulka A.15 – *Pokračování z předešlé strany*

Velikost pole	Min [ms]	Max [ms]	Průměr [ms]	Sm. Odchylka [ms]
18000	4296	4826	4522	186,22
20000	5314	6513	5643	458,85
22000	6398	6476	6419	28,74
24000	7623	8183	7747	218,86
26000	9005	9104	9033	36,43
28000	10456	10493	10480	12,92
30000	12048	12211	12101	58,30
32000	13949	14006	13979	19,01
34000	15754	19107	16477	1315,50
36000	17809	21170	18577	1299,84
38000	19893	26428	21781	2491,31

Tabulka A.16: Tabulka naměřených hodnot pro bubble-sort s použitým do-while cyklem a post inkrementací, spuštěný na HotSpot (Server kompilátor)

Velikost pole	Min [ms]	Max [ms]	Průměr [ms]	Sm. Odchylka [ms]
500	27	29	27	0,75
1000	60	61	60	0,49
1500	80	84	81	1,36
2000	94	99	95	1,85
2500	113	115	113	0,8
3000	150	155	152	1,67
3500	200	265	213	26
4000	251	258	255	3,2
4500	299	312	304	5,53
5000	373	376	374	0,98
5500	440	535	471	34,69
6000	530	548	537	8,03
6500	613	665	635	16,65
7000	704	725	716	9,28
7500	802	1032	901	103,12
8000	969	1080	1010	47,82
8500	1052	1279	1098	90,40
9000	1176	1401	1222	89,32
9500	1345	1722	1428	146,81
10000	1413	1486	1455	33,39
12000	2221	2772	2353	212,85
14000	2921	3625	3066	279,52
16000	3825	3986	3941	59
18000	4895	6095	5544	509,29
20000	5764	6736	6060	353,31
22000	6925	7004	6977	27,05
24000	8227	8462	8393	84,98

Pokračování na následující straně

Tabulka A.16 – *Pokračování z předešlé strany*

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
26000	9655	10454	9835	309,72
28000	11113	13423	11655	891,94
30000	12042	12906	12617	301,85
32000	14105	14333	14202	82,68
34000	15902	16271	16060	126,44
36000	17894	20582	18491	1047,39
38000	19975	21214	20478	449,78

Tabulka A.17: Tabulka naměřených hodnot pro bubble-sort s použitým do-while cyklem a pre inkrementací, spuštěný na JamVM

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	74	75	74	0,49
1000	302	303	302	0,49
1500	687	700	693	5,2
2000	1228	1235	1231	2,58
2500	1974	2007	1988	11,02
3000	2825	2842	2832	5,46
3500	3905	3917	3911	4,02
4000	5061	5148	5107	33,36
4500	6384	6413	6399	11,02
5000	8045	8130	8096	32,06
5500	9577	9707	9618	45,68
6000	11382	12704	11896	520,46
6500	13350	14925	13850	601,68
7000	15569	17638	16514	846,67
7500	17927	19458	18692	485,06
8000	20437	22882	21627	949,48
8500	22887	25649	24027	998,36
9000	25490	29299	26652	1453,73
9500	28641	31756	29413	1188,31
10000	31541	32527	31858	363,56
12000	46214	48350	47103	923,98
14000	62841	66487	64149	1567,66
16000	83891	86114	84842	922,65
18000	107107	113448	110537	2495,72
20000	135506	146325	140744	3924,75

Tabulka A.18: Tabulka naměřených hodnot pro bubble-sort s použitým do-while cyklem a pre inkrementací, spuštěný na HotSpot (Interpret only)

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	41	42	41	0,49
1000	165	170	166	1,94
1500	379	383	380	1,47
2000	667	670	669	1,17
2500	1042	1074	1051	11,87
3000	1504	1518	1509	5,28
3500	2040	2324	2100	111,74
4000	2657	2667	2661	3,71
4500	3354	3464	3385	40,11
5000	4128	4308	4175	67,72
5500	4987	5158	5028	65,36
6000	5938	6124	5987	69,23
6500	6941	7046	6972	37,38
7000	8090	9628	8634	587,33
7500	9272	12338	9987	1186,29
8000	10578	12056	11159	564,56
8500	11935	15611	14103	1438,96
9000	13917	17373	15023	1303,58
9500	14909	17850	15922	1011,43
10000	16434	19217	17944	935,45
12000	23847	29151	26375	1942,13
14000	32485	35132	33628	957,87
16000	43640	49796	46035	2177,03
18000	53893	63391	57485	3771,02
20000	70169	71815	71234	608,95

Tabulka A.19: Tabulka naměřených hodnot pro bubble-sort s použitým do-while cyklem a pre inkrementací, spuštěný na HotSpot (Cleint kompilátor)

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	9	10	9	0,49
1000	25	25	25	0
1500	42	42	42	0
2000	65	65	65	0
2500	93	94	93	0,4
3000	125	125	125	0
3500	165	165	165	0
4000	212	215	213	0,98
4500	263	264	263	0,4
5000	322	324	323	0,89

Pokračování na následující straně

Tabulka A.19 – *Pokračování z předešlé strany*

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
5500	393	394	393	0,4
6000	462	463	462	0,49
6500	541	544	542	1,47
7000	628	629	628	0,4
7500	725	725	725	0
8000	825	827	825	0,8
8500	931	936	933	1,72
9000	1042	1044	1042	0,75
9500	1156	1159	1157	1,2
10000	1280	1285	1281	1,74
12000	1847	1853	1848	2,22
14000	2510	2681	2547	67,03
16000	3275	3307	3283	12,28
18000	4167	4193	4177	10,68
20000	5144	5167	5152	8,13
22000	6295	6352	6311	20,49
24000	7447	7545	7472	36,84
26000	8777	8854	8807	26,70
28000	10233	10256	10243	7,74
30000	11792	11904	11835	37,81
32000	13499	17291	14288	1502,22
34000	15378	16897	15807	579,02
36000	17344	17501	17417	60,19
38000	19402	21380	20049	712,16

Tabulka A.20: Tabulka naměřených hodnot pro bubble-sort s použitým do-while cyklem a pre inkrementací, spuštěný na HotSpot (Server kompilátor)

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	24	29	27	1,85
1000	60	63	61	1,02
1500	80	85	82	1,62
2000	94	106	102	4,56
2500	127	135	131	2,93
3000	150	185	176	13,34
3500	217	268	255	19,18
4000	258	316	279	26,42
4500	300	382	349	38,18
5000	372	491	445	45,2
5500	472	595	522	50,72
6000	609	694	665	30,49
6500	635	818	710	87,39
7000	702	955	787	106,58

Pokračování na následující straně

Tabulka A.20 – *Pokračování z předešlé strany*

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
7500	822	1070	1000	94,47
8000	974	1266	1188	108,31
8500	1051	1346	1284	116,82
9000	1184	1520	1384	144,35
9500	1365	1723	1520	157,40
10000	1426	1747	1522	119,60
12000	2239	2867	2637	278,21
14000	3132	3790	3536	278,98
16000	3872	4824	4321	320,49
18000	5192	6103	5689	349,19
20000	5804	6769	6286	343,27
22000	7012	8353	7484	500,62
24000	8429	9272	8885	269,67
26000	10037	11935	10937	819,70
28000	10910	11952	11293	351,93
30000	12058	14511	13343	818,57
32000	14471	15754	15055	504,60
34000	16180	17895	17382	622,41
36000	18628	19110	18848	200,40
38000	20709	23271	21611	957,20

A.2 Výsledky testů pro algoritmus quick-sort

Tabulka A.21: Tabulka naměřených hodnot pro quick-sort, spuštěný na JamVM

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
10000	246	257	252	3,52
20000	523	550	532	9,70
30000	818	867	843	16,99
40000	1102	1188	1132	30,31
50000	1423	1481	1449	21,06
60000	1745	1839	1787	30,16
70000	2004	2089	2046	33,48
80000	2327	2517	2429	76
90000	2694	2842	2770	46,92
100000	2998	3077	3041	28,12
110000	3283	3539	3400	91,39
120000	3558	3802	3652	88,50
130000	3985	4225	4118	96,31
140000	4254	4484	4344	81,08
150000	4562	4868	4663	121,44
160000	4895	5116	5021	83,51

Pokračování na následující straně

Tabulka A.21 – *Pokračování z předešlé strany*

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
170000	5296	5496	5425	73,44
180000	5616	5914	5800	99,01
190000	6046	6392	6180	145,62
200000	6359	6476	6432	44,03
210000	6719	7117	6906	163,59
220000	6978	7158	7054	69,84
230000	7376	7667	7517	101
240000	7583	8258	7874	224
250000	8211	8470	8320	97,86
260000	8372	8755	8583	124,65
270000	8731	9199	8892	166,50
280000	8901	9831	9261	325,98
290000	9543	9955	9738	148,90
300000	9999	10343	10158	144,38
310000	10243	10636	10417	144,16
320000	10598	11105	10777	177,12
330000	10728	11575	11188	283,79
340000	11313	12009	11550	251,50
350000	11689	12206	11937	215,42
360000	12279	12778	12531	187,96
370000	12737	13860	13129	396,76
380000	13135	14022	13482	355,13
390000	13468	13945	13692	169,09
400000	14032	14468	14211	195,14
410000	14588	15144	14863	215,92
420000	14669	15283	15052	217,90
430000	15394	16217	15666	286,33
440000	14963	16112	15509	501,18
450000	14882	17463	16243	851,98
460000	15707	16632	16041	316,45
470000	15880	18471	17023	894,62
480000	16170	16951	16715	287,03
490000	16672	16919	16783	109,59
500000	16715	17555	17058	311,54
700000	24921	25598	25193	223,54
900000	32225	34212	33138	751,89
1100000	40421	42548	41321	686,54

Tabulka A.22: Tabulka naměřených hodnot pro quick-sort, spuštěný na HotSpot (Interpret only)

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
10000	203	228	212	9,11
20000	448	457	451	3

Pokračování na následující straně

Tabulka A.22 – *Pokračování z předešlé strany*

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
30000	679	719	699	14,38
40000	889	956	923	22,07
50000	1148	1239	1181	38,14
60000	1406	1478	1437	27,48
70000	1641	1715	1688	26,12
80000	1823	1988	1929	55,73
90000	2111	2223	2185	38,99
100000	2417	2555	2481	44,82
110000	2689	2802	2739	36,55
120000	2923	3032	2976	36,74
130000	3266	3345	3294	26,91
140000	3496	3592	3545	34,24
150000	3759	3917	3847	53,11
160000	3883	4204	4091	117,1
170000	4173	4374	4305	71,83
180000	4346	4695	4504	128,48
190000	4840	4994	4921	58,98
200000	4907	5174	5022	101,45
210000	5283	5620	5439	117,63
220000	5509	5990	5688	175,64
230000	5764	6055	5938	95,88
240000	6028	6330	6231	109,51
250000	6365	6802	6550	196,42
260000	6663	6925	6749	101,20
270000	6779	7319	7042	184,18
280000	7251	7620	7439	130,34
290000	7440	7591	7515	60,97
300000	7803	8110	7978	122,38
310000	8040	8613	8311	186,38
320000	8454	8548	8495	34,71
330000	8551	8756	8646	74,04
340000	8863	9172	9014	106,37
350000	8978	9540	9233	180,52
360000	9428	9659	9531	104,35
370000	9876	10223	10028	117,99
380000	10094	10457	10206	131,97
390000	10289	10625	10440	116,66
400000	10399	11081	10636	240,06
410000	10607	11234	10912	228,23
420000	11185	11345	11287	56,99
430000	11319	11820	11547	159,96
440000	11546	12231	11825	266,50
450000	11792	12464	12152	240,90
460000	12429	12651	12502	76,65

Pokračování na následující straně

Tabulka A.22 – *Pokračování z předešlé strany*

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
470000	11911	12740	12456	300,84
480000	12412	13121	12719	282,28
490000	13093	13545	13328	160,30
500000	13323	14474	13821	435,89
700000	18551	20238	19562	649,75
900000	24922	26041	25518	395
1100000	31387	32175	31696	317,97

Tabulka A.23: Tabulka naměřených hodnot pro quick-sort, spuštěný na HotSpot (Client kompilátor)

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
10000	29	37	31	2,94
20000	51	59	54	3,01
30000	76	79	77	1,1
40000	91	98	95	2,64
50000	96	105	101	2,94
60000	115	128	120	4,50
70000	132	143	137	3,87
80000	155	164	159	3,88
90000	169	182	177	4,69
100000	189	194	191	1,72
110000	206	212	208	2,71
120000	227	233	229	2,15
130000	250	262	256	4,45
140000	271	282	275	3,85
150000	283	296	288	4,58
160000	299	319	307	7,53
170000	330	370	341	15,05
180000	366	400	390	12,35
190000	379	433	406	22,14
200000	386	429	410	16,12
210000	438	460	446	7,79
220000	439	470	460	11,41
230000	487	504	494	6,58
240000	507	524	517	5,91
250000	522	560	536	12,69
260000	535	597	568	24,65
270000	605	629	615	8,68
280000	604	660	640	19,01
290000	610	677	649	24,02
300000	625	690	653	24,80
310000	647	703	686	20,43
320000	697	739	715	13,96

Pokračování na následující straně

Tabulka A.23 – *Pokračování z předešlé strany*

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
330000	735	756	747	6,86
340000	766	771	768	1,67
350000	781	796	789	5,16
360000	795	810	801	5,11
370000	800	840	824	14,04
380000	785	811	800	8,89
390000	827	877	844	18,26
400000	849	923	894	29,52
410000	876	949	925	26,74
420000	884	963	934	26,78
430000	914	968	934	18,61
440000	929	1020	971	35,03
450000	953	1044	995	38,38
460000	1019	1083	1053	26
470000	1064	1108	1080	15,40
480000	1082	1106	1096	10,02
490000	1095	1122	1111	9,67
500000	1114	1138	1128	9,33
700000	1517	1647	1592	54,03
900000	2072	2132	2106	21,44
1100000	2467	2727	2539	95,38

Tabulka A.24: Tabulka naměřených hodnot pro quick-sort, spuštěný na HotSpot (Server kompilátor)

Velikost pole	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
10000	72	89	78	5,91
20000	115	122	118	2,24
30000	123	132	129	3,32
40000	130	149	142	6,53
50000	144	161	155	6,54
60000	168	175	171	2,24
70000	179	185	182	2,32
80000	176	197	187	7,19
90000	201	214	209	4,86
100000	214	233	223	6,81
110000	219	234	228	5,64
120000	226	250	240	9,93
130000	247	265	256	7,55
140000	245	283	263	12,61
150000	272	283	276	3,93
160000	265	301	283	14,29
170000	283	316	304	13,15
180000	307	332	313	9,39

Pokračování na následující straně

Tabulka A.24 – *Pokračování z předešlé strany*

Velikost pole	Min [ms]	Max [ms]	Průměr [ms]	Sm. Odchylka [ms]
190000	316	344	325	10,35
200000	339	352	346	4,22
210000	349	369	359	8,82
220000	367	383	373	5,83
230000	391	412	404	7,43
240000	397	414	406	6,08
250000	419	437	426	5,91
260000	428	547	459	45,37
270000	423	473	443	17,53
280000	429	448	442	6,78
290000	450	510	477	21,28
300000	455	498	475	13,85
310000	476	522	491	16,12
320000	482	536	508	17,94
330000	504	523	516	6,96
340000	536	569	548	12,92
350000	529	541	536	4,34
360000	553	604	580	19,6
370000	558	598	579	13,51
380000	578	610	589	11,45
390000	586	663	620	27,24
400000	604	649	623	17,17
410000	625	685	646	22,28
420000	620	646	631	10,29
430000	657	692	678	13,63
440000	665	688	673	7,78
450000	662	698	682	13,24
460000	700	734	717	14,31
470000	717	777	752	21,22
480000	732	807	769	28,02
490000	775	805	786	11
500000	786	833	801	17,35
700000	1309	1373	1331	23,2
900000	1706	1757	1723	17,9
1100000	2183	2216	2202	11,77

A.3 Výsledky testů výpočtu π

Tabulka A.25: Tabulka naměřených hodnot pro výpočet PI, spuštěný na JamVM

Přesnost	Min [ms]	Max [ms]	Průměr [ms]	Sm. Odchylka [ms]
4	21	25	22	1,36

Pokračování na následující straně

Tabulka A.25 – *Pokračování z předešlé strany*

Přesnost	**Min** [ms]	**Max** [ms]	**Průměr** [ms]	**Sm. Odchylka** [ms]
5	208	210	208	0,75
6	2077	2158	2099	30,66
7	20770	20789	20778	7,36
8	207927	210613	208620	1011,69

Tabulka A.26: Tabulka naměřených hodnot pro výpočet PI, spuštěný na HotSpot (Interpret Only)

Přesnost	**Min** [ms]	**Max** [ms]	**Průměr** [ms]	**Sm. Odchylka** [ms]
Přesnost	Min	Max	Průměr	Sm. odchylka
4	12	13	12	0,4
5	116	117	116	0,4
6	1164	1171	1167	3,01
7	11649	11652	11650	1,02
8	116426	116518	116478	35,92

Tabulka A.27: Tabulka naměřených hodnot pro výpočet PI, spuštěný na HotSpot (Client kompilátor)

Přesnost	**Min** [ms]	**Max** [ms]	**Průměr** [ms]	**Sm. Odchylka** [ms]
Přesnost	Min	Max	Průměr	Sm. odchylka
4	7	8	7	0,49
5	57	58	57	0,4
6	553	555	553	0,75
7	5517	5532	5521	5,31
8	55143	55170	55154	11,50

Tabulka A.28: Tabulka naměřených hodnot pro výpočet PI, spuštěný na HotSpot (Server kompilátor)

Přesnost	**Min** [ms]	**Max** [ms]	**Průměr** [ms]	**Sm. Odchylka** [ms]
Přesnost	Min	Max	Průměr	Sm. odchylka
4	10	11	10	0,49
5	60	61	60	0,49
6	561	576	565	5,39
7	5567	5569	5567	0,8
8	55609	55632	55614	8,7

A.4 Výsledky testů zřetězení stringů

Tabulka A.29: Tabulka naměřených hodnot pro zřetězení stringů pomocí operátoru +, spuštěný na JamVM

Počet zřetězení	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	15	16	15	0,49
1000	46	47	46	0,49
1500	80	80	80	0
2000	123	124	123	0,4
2500	179	181	180	0,75
3000	248	250	249	0,8
3500	329	330	329	0,4
4000	422	423	422	0,4
4500	528	535	530	2,53
5000	645	650	647	1,85
5500	775	778	776	1,36
6000	919	921	920	0,63
6500	1072	1077	1074	1,62
7000	1240	1246	1243	2,28
7500	1420	1428	1423	3,25
8000	1614	1616	1614	0,75
8500	1818	1823	1820	1,85
9000	2036	2086	2048	18,69
9500	2268	2274	2270	2,1
10000	2507	2517	2511	3,2
12000	3885	3897	3891	5,02
14000	6214	6219	6217	1,67
16000	8887	8931	8899	16,25
18000	11912	11930	11921	7,14
20000	15288	15309	15296	6,89

Tabulka A.30: Tabulka naměřených hodnot pro zřetězení stringů pomocí operátoru +, spuštěný na HotSpot (Interpret only)

Počet zřetězení	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	13	15	14	0,75
1000	45	50	47	1,85
1500	93	96	94	1,02
2000	159	165	161	2,15
2500	247	257	250	3,50
3000	322	331	325	3,38
3500	418	425	422	2,61
4000	512	524	514	4,62

Pokračování na následující straně

Tabulka A.30 – *Pokračování z předešlé strany*

Počet zřetězení	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
4500	614	619	616	1,94
5000	723	733	727	3,50
5500	846	858	852	4,73
6000	984	998	991	4,49
6500	1137	1145	1142	2,87
7000	1293	1315	1304	7,03
7500	1474	1490	1479	5,81
8000	1644	1659	1653	5,04
8500	1844	1874	1859	10,75
9000	2060	2076	2069	5,95
9500	2271	2311	2281	15,49
10000	2506	2530	2519	9,48
12000	3546	3589	3562	14,74
14000	4810	4842	4825	11,17
16000	6253	6283	6268	11,32
18000	7906	7954	7930	17,81
20000	9842	9862	9853	7,09

Tabulka A.31: Tabulka naměřených hodnot pro zřetězení stringů pomocí operátoru +, spuštěný na HotSpot (Client kompilátor)

Počet zřetězení	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
500	17	22	20	1,72
1000	47	76	55	10,46
1500	94	107	100	5,39
2000	167	181	171	5,18
2500	265	286	276	8,8
3000	386	417	394	11,65
3500	523	578	558	20,33
4000	704	743	716	14,64
4500	903	981	947	31,75
5000	1105	1210	1176	39,22
5500	1394	1492	1451	43,88
6000	1683	1792	1751	36,9
6500	2073	2143	2110	24,56
7000	2480	2530	2507	17,96
7500	2897	2959	2926	23,91
8000	3320	3405	3371	31,75
8500	3788	3928	3896	54,28
9000	4317	4485	4400	66,45
9500	5076	5128	5103	20,44
10000	5712	5763	5743	20,25
12000	8879	8962	8940	30,87

Pokračování na následující straně

Tabulka A.31 – *Pokračování z předešlé strany*

Počet zřetězení	Min [ms]	Max [ms]	Průměr [ms]	Sm. Odchylka [ms]
14000	13302	13389	13346	34,65
16000	19443	19487	19461	14,54
18000	26943	27116	27049	57,73
20000	35994	36184	36098	78,73

Tabulka A.32: Tabulka naměřených hodnot pro zřetězení stringů pomocí operátoru +, spuštěný na HotSpot (Server kompilátor)

Počet zřetězení	Min [ms]	Max [ms]	Průměr [ms]	Sm. Odchylka [ms]
500	14	15	14	0,4
1000	44	49	46	1,72
1500	91	99	94	2,65
2000	160	164	161	1,6
2500	245	258	250	4,76
3000	322	334	327	4,65
3500	427	430	428	0,98
4000	514	525	518	3,67
4500	615	623	618	2,99
5000	723	752	736	9,44
5500	850	866	858	6,09
6000	981	1003	989	7,58
6500	1122	1140	1133	6,24
7000	1279	1296	1286	5,49
7500	1446	1459	1450	4,67
8000	1632	1660	1643	9,44
8500	1813	1848	1825	13,05
9000	2017	2054	2030	13,73
9500	2242	2262	2252	8,01
10000	2448	2471	2460	7,66
12000	3419	3435	3425	5,74
14000	4563	4584	4571	7,36
16000	5861	5892	5875	10,38
18000	7356	7388	7377	11,69
20000	9067	9088	9075	7,78

Tabulka A.33: Tabulka naměřených hodnot pro zřetězení stringů pomocí StringBuilderu, spuštěný na JamVM

Počet zřetězení	Min [ms]	Max [ms]	Průměr [ms]	Sm. Odchylka [ms]
1000	0	1	0	0,4
2000	1	2	1	0,49
3000	2	2	2	0

Pokračování na následující straně

Tabulka A.33 – *Pokračování z předešlé strany*

Počet zřetězení	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
4000	3	3	3	0
5000	4	4	4	0
6000	5	6	5	0,4
7000	6	6	6	0
8000	7	7	7	0
9000	8	8	8	0
10000	9	9	9	0
11000	10	11	10	0,4
12000	10	11	10	0,4
13000	11	12	11	0,49
14000	12	14	12	0,75
15000	14	14	14	0
16000	15	15	15	0
17000	16	17	16	0,4
18000	16	19	18	1,26
19000	17	18	17	0,49
20000	18	19	18	0,4
21000	19	20	19	0,4
22000	20	20	20	0
23000	21	22	21	0,4
24000	21	22	21	0,4
25000	23	23	23	0
26000	23	24	23	0,49
27000	24	25	24	0,4
28000	25	25	25	0
29000	26	26	26	0
30000	29	33	29	1,6
35000	33	36	33	1,2
40000	37	38	37	0,49
45000	42	44	43	0,90
50000	46	48	46	0,98
55000	50	51	50	0,4
60000	58	60	58	0,75
65000	62	70	64	2,93
70000	67	74	68	2,8
75000	71	75	72	1,50
80000	75	76	75	0,49
85000	79	80	79	0,49
90000	84	86	84	0,8
95000	88	89	88	0,4
100000	92	93	92	0,49

Tabulka A.34: Tabulka naměřených hodnot pro zřetězení stringů pomocí StringBuilderu, spuštěný na HotSpot (Interpret only)

Počet zřetězení	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
1000	0	0	0	0
2000	1	1	1	0
3000	2	2	2	0
4000	3	3	3	0
5000	4	4	4	0
6000	5	5	5	0
7000	5	6	5	0,49
8000	7	7	7	0
9000	8	8	8	0
10000	8	8	8	0
11000	9	10	9	0,4
12000	10	10	10	0
13000	11	11	11	0
14000	12	12	12	0
15000	13	14	13	0,49
16000	14	14	14	0
17000	15	15	15	0
18000	16	17	16	0,49
19000	16	17	16	0,49
20000	17	18	17	0,49
21000	18	19	18	0,4
22000	19	20	19	0,4
23000	19	20	19	0,4
24000	21	21	21	0
25000	21	25	22	1,50
26000	22	23	22	0,4
27000	23	24	23	0,4
28000	24	24	24	0
29000	24	27	25	0,98
30000	27	28	27	0,4
35000	31	33	32	0,63
40000	35	37	36	0,75
45000	39	41	40	0,63
50000	43	46	44	1,02
55000	48	48	48	0
60000	55	58	56	0,98
65000	59	62	60	1,02
70000	63	67	64	1,36
75000	67	69	68	0,75
80000	71	75	73	1,41
85000	75	78	76	0,98

Pokračování na následující straně

Tabulka A.34 – *Pokračování z předešlé strany*

Počet zřetězení	Min [ms]	Max [ms]	Průměr [ms]	Sm. Odchylka [ms]
90000	78	82	79	1,33
95000	83	86	83	1,2
100000	88	89	88	0,49

Tabulka A.35: Tabulka naměřených hodnot pro zřetězení stringů pomocí StringBuilderu, spuštěný na HotSpot (Client kompilátor)

Počet zřetězení	Min [ms]	Max [ms]	Průměr [ms]	Sm. Odchylka [ms]
1000	1	1	1	0
2000	2	2	2	0
3000	2	2	2	0
4000	3	3	3	0
5000	3	3	3	0
6000	3	3	3	0
7000	3	4	3	0,4
8000	4	4	4	0
9000	5	5	5	0
10000	5	5	5	0
11000	5	5	5	0
12000	6	6	6	0
13000	6	6	6	0
14000	6	6	6	0
15000	7	8	7	0,4
16000	8	8	8	0
17000	8	8	8	0
18000	8	9	8	0,4
19000	9	9	9	0
20000	9	9	9	0
21000	9	10	9	0,4
22000	10	12	10	0,8
23000	10	11	10	0,4
24000	10	11	10	0,4
25000	11	11	11	0
26000	11	11	11	0
27000	11	12	11	0,49
28000	11	12	11	0,49
29000	12	13	12	0,4
30000	16	21	18	1,9
35000	17	21	19	1,67
40000	19	25	21	2,4
45000	20	26	21	2,33
50000	22	27	24	1,79
55000	23	24	23	0,4

Pokračování na následující straně

Tabulka A.35 – *Pokračování z předešlé strany*

Počet zřetězení	Min [ms]	Max [ms]	Průměr [ms]	Sm. Odchylka [ms]
60000	28	34	30	2,42
65000	30	31	30	0,49
70000	32	35	32	1,2
75000	31	37	34	2,33
80000	33	39	35	2,42
85000	34	40	35	2,24
90000	36	43	39	2,42
95000	37	38	37	0,4
100000	38	43	41	1,85

Tabulka A.36: Tabulka naměřených hodnot pro zřetězení stringů pomocí StringBuilderu, spuštěný na HotSpot (Server kompilátor)

Počet zřetězení	Min [ms]	Max [ms]	Průměr [ms]	Sm. Odchylka [ms]
1000	0	0	0	0
2000	1	1	1	0
3000	2	2	2	0
4000	3	5	3	0,8
5000	4	4	4	0
6000	5	5	5	0
7000	6	6	6	0
8000	7	11	8	1,55
9000	8	8	8	0
10000	9	9	9	0
11000	9	10	9	0,49
12000	10	10	10	0
13000	11	13	11	0,8
14000	12	12	12	0
15000	14	14	14	0
16000	15	15	15	0
17000	15	16	15	0,49
18000	15	16	15	0,49
19000	16	18	17	0,63
20000	17	17	17	0
21000	17	19	17	0,75
22000	18	18	18	0
23000	18	20	19	0,89
24000	19	19	19	0
25000	19	20	19	0,4
26000	19	20	19	0,49
27000	20	21	20	0,4
28000	20	22	20	0,8
29000	20	20	20	0

Pokračování na následující straně

Tabulka A.36 – *Pokračování z předešlé strany*

Počet zřetězení	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
30000	21	23	22	0,63
35000	23	24	23	0,4
40000	23	25	24	0,8
45000	24	28	26	1,33
50000	26	29	27	0,98
55000	27	28	27	0,49
60000	32	33	32	0,4
65000	34	37	35	1,26
70000	35	36	35	0,4
75000	36	38	37	0,63
80000	37	40	37	1,17
85000	38	39	38	0,4
90000	39	42	40	0,98
95000	41	42	41	0,4
100000	42	42	42	0

A.5 Výsledky testů rychlosti provádení podmínek

Tabulka A.37: Tabulka naměřených hodnot pro test provádění podmínek

	HS Client	**HS Server**	**HS Interpret**	**JamVM**
Min	200	210	448	453
Max	265	318	563	687
Průměr	236,25	235,3	470,5	518,45
Sm. odchylka	16,92	22,33	25,16	58,41

A.6 Výsledky testů rychlosti rekurze

Tabulka A.38: Tabulka naměřených hodnot pro test rychlosti rekurze, spuštěný na JamVM

f(x)	**Min [ms]**	**Max [ms]**	**Průměr [ms]**	**Sm. Odchylka [ms]**
10	80	85	82	1,79
20	249	255	252	2,28
30	502	516	508	4,77
40	843	862	852	6,66
50	1273	1287	1279	4,67
60	1782	1800	1791	6,69
70	2373	2393	2382	8,29
80	3078	3115	3089	13,82
90	3838	3860	3849	7,93

Pokračování na následující straně

Tabulka A.38 – *Pokračování z předešlé strany*

f(x)	Min [ms]	Max [ms]	Průměr [ms]	Sm. Odchylka [ms]
100	4674	4714	4688	13,38
110	5628	5665	5645	14,29

Tabulka A.39: Tabulka naměřených hodnot pro test rychlosti rekurze, spuštěný na HotSpot (Interpret only)

f(x)	Min [ms]	Max [ms]	Průměr [ms]	Sm. Odchylka [ms]
10	61	62	61	0,4
20	179	181	179	0,8
30	351	354	352	1,17
40	578	584	581	2,32
50	857	868	861	4,17
60	1190	1197	1192	2,56
70	1588	1594	1591	2,53
80	2027	2047	2035	7,93
90	2528	2595	2548	23,89
100	3077	3091	3083	5
110	3682	3700	3693	6,52

Tabulka A.40: Tabulka naměřených hodnot pro test rychlosti rekurze, spuštěný na HotSpot (Client kompilátor)

f(x)	Min [ms]	Max [ms]	Průměr [ms]	Sm. Odchylka [ms]
10	29	30	29	0,49
20	64	65	64	0,49
30	111	113	112	0,63
40	166	168	167	0,63
50	228	232	229	1,47
60	305	307	305	0,75
70	384	391	388	2,61
80	477	484	481	2,28
90	582	589	585	2,24
100	693	699	696	2,24
110	815	826	819	3,88

Tabulka A.41: Tabulka naměřených hodnot pro test rychlosti rekurze, spuštěný na HotSpot (Server kompilátor)

f(x)	Min [ms]	Max [ms]	Průměr [ms]	Sm. Odchylka [ms]
10	29	32	30	1,02
20	62	64	63	0,9
30	102	104	102	0,8

Pokračování na následující straně

Tabulka A.41 – *Pokračování z předešlé strany*

f(x)	Min [ms]	Max [ms]	Průměr [ms]	Sm. Odchylka [ms]
40	146	148	147	0,75
50	198	201	199	1,16
60	256	273	262	5,71
70	321	352	330	11,09
80	391	398	395	2,42
90	471	480	476	3,41
100	557	565	560	2,68
110	652	660	654	2,99

A.7 Výsledky testů invokací metod

Tabulka A.42: Tabulka naměřených hodnot pro test rychlosti invokace s možností vkládní

	HS Client	HS Server	HS Interpret	JamVM
Min [ns]	206	203	309	364
Max [ns]	259	208	327	438
Průměr [ns]	216,45	204,65	315	387,4
Sm. odchylka [ns]	14,59	1,71	4,04	26,45

Tabulka A.43: Tabulka naměřených hodnot pro test rychlosti invokace bez možnosti vkládní

	HS Client	HS Server	HS Interpret	JamVM
Min [ns]	229	219	322	382
Max [ns]	298	224	346	500
Průměr [ns]	237,35	221,85	330,3	420,05
Sm. odchylka [ns]	15,57	1,35	6,02	31,5

A.8 Výsledky testu pro měření rychlosti strartu virtuálního stroje

Tabulka A.44: Tabulka naměřených hodnot pro test rychlosti startu virtuálního stroje

	HS Client	HS Server	HS Interpret	JamVM
Min [ms]	88	87	87	50
Max [ms]	98	96	91	52
Průměr [ms]	88,65	88,21	87,39	50,17
Sm. odchylka [ms]	1,25	0,91	0,77	0,43

A.9 Výsledky měření velikosti virtuálního stroje

Tabulka A.45: Tabulka naměřených hodnot pro test velikosti virtuálního stroje

	HS Client	**HS Server**	**HS Interpret**	**JamVM**
Min [kB]	10800	11304	11288	4576
Max [kB]	10856	11308	11296	4580
Průměr [kB]	10832,4	11305,6	11290,4	4577,2
Sm. odchylka [kB]	20,35	1,96	2,65	1,83

A.10 Výsledky monitorování heapu

Tabulka A.46: Tabulka naměřených hodnot při monitorování heapu pro JamVM

time [ms]	**velikost haldy [MB]**
1	47,19
18895	70,78
28727	106,17
43495	159,25
65659	238,88
98859	358,31
107099	358,31
125225	537,47
531182	537,47

Tabulka A.47: Tabulka naměřených hodnot při monitorování heapu pro HotSpot (Interpret only)

time [ms]	**velikost haldy [MB]**
1	45,5
7460	61,5
10356	63
12384	86
17065	110,5
22556	123
26990	136,25
35358	173,75
38989	181,25
45583	186,5
53607	278,5
60567	284,5
76778	288,5

Pokračování na následující straně

Tabulka A.47 – *Pokračování z předešlé strany*

time [ms]	velikost haldy [MB]
86611	396
87768	401,75
93602	401,75
93640	440
96011	479
99178	486,5
106881	489,75
112760	496,25
116965	503,25
121240	505,25
125472	509,5
134566	513,75
143444	517
152909	520,5
167644	482,75
172850	517,5
178076	519,75
183433	521,75
188791	523,75
193673	520,75
197901	479
203203	514
208774	479,5
214332	511,25
220048	479,25
224905	508,5
230265	480,75
236024	506,5
240574	508,5
245808	507
257563	505
263740	506,75
269830	504,75
276093	506
305686	510,25
318803	511,75
343866	515
350793	517,75
357749	521
371804	524
399029	612,25
406175	617,5
413368	622,25
417555	622,25

Tabulka A.48: Tabulka naměřených hodnot při monitorování heapu pro HotSpot (Client kompilátor)

time [ms]	**velikost haldy [MB]**
1	15,10
641	21,66
879	34,35
1206	55,96
1677	84,77
2386	142,04
3314	211,34
4840	314,55
7403	486,95
7598	486,95
8944	494,94
44942	494,94

Tabulka A.49: Tabulka naměřených hodnot při monitorování heapu pro HotSpot (Server kompilátor)

time [ms]	**velikost haldy [MB]**
1	45,5
207	57,75
382	82,25
968	117,25
1159	153
1426	179,5
2749	236
3049	246,25
4584	323,25
4851	344
5219	347,25
7075	434,75
7414	441,5
7488	441,5
9327	519,75
9545	521
11825	606
14690	597,5
14883	601,75
15137	605
16150	522,5
16407	604,5
16990	608,25
17322	580
17667	606

Pokračování na následující straně

Tabulka A.49 – *Pokračování z předešlé strany*

time [ms]	**velikost haldy [MB]**
19644	625
19913	573,75
20170	619
20459	622
20781	614,5
21132	620
21425	613,5
21772	617
22422	618
22801	624
23515	625,75
23888	627
24680	628,5
25083	626
25482	628,5
26288	630,25
26694	635,25
27634	637,75
28114	639,25
29069	641,5
29578	644,25
30620	647,75
31155	649
32098	600,75
32607	645,5
32926	645,5