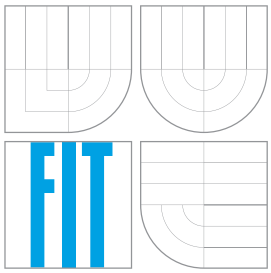
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

OPTIMALIZACE VELIKOSTI BAJTKÓDU JAVY

JAVA BYTECODE SIZE OPTIMIZATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. VENDULA PONCOVÁ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK KOČÍ, Ph.D.

BRNO 2016

Zadání diplomové práce/14380/2015/xponco00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Poncová Vendula, Bc.**
Obor: Matematické metody v informačních technologiích
Téma: **Optimalizace velikosti bajtkódu Javy**
Java Bytecode Size Optimization
Kategorie: Operační systémy

Pokyny:

1. Nastudujte vlastnosti instrukčního souboru JRE a strukturu bajtkódu použitého v JRE.
2. Seznamte se s existujícími nástroji, které lze použít pro manipulaci s bajtkódem (BCEL atd.).
3. V dostatečně rozsáhlém testovacím vzorku přeložených aplikací nalezněte a analyzujte typické vzory (sekvence instrukcí), které se v bajtkódu vyskytují.
4. Navrhněte možnosti optimalizace velikosti bajtkódu.
5. Implementujte nástroj pro optimalizaci bajtkódu na základě výše navržených metod.
6. Ověřte správnou funkčnost optimalizovaného bajtkódu. Navrhěte vhodnou sadu testů ověřující chování optimalizovaného bajtkódu při ladění aplikací.
7. Zhodnoťte dosažené výsledky a navrhněte možnosti dalšího vývoje projektu.

Literatura:

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

Při obhajobě semestrální části projektu je požadováno:

- První 3 body zadání.

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

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

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

Vedoucí: **Kočí Radek, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Tato práce se zabývá optimalizací bajtkódu jazyka Java z hlediska jeho velikosti. Popisuje virtuální stroj Javy a formát jeho instrukčního souboru a uvádí přehled některých knihoven pro manipulaci s bajtkódem. S pomocí těchto knihoven byla provedena analýza vybraného vzorku dat a nalezeny sekvence instrukcí, které by bylo možné optimalizovat. Na základě výsledků analýzy byly navrhnuty a implementovány metody pro optimalizaci velikosti bajtkódu. Velikost bajtkódu zkoumaného vzorku dat se po aplikaci metod snížila o zhruba 25%.

Abstract

This paper deals with the Java bytecode size optimization. It describes the Java Virtual Machine and the Java class file format. It also presents some tools for the bytecode manipulation. Using these tools, I have analyzed selected data and found sequences of instructions, that could be optimized. Based on the results of the analysis, I have designed and implemented methods for bytecode size optimization. The bytecode size of the selected data was reduced by roughly 25%.

Klíčová slova

Java, JVM, bajtkód, ASM, BCEL, Javassist, optimalizace velikosti, peephole optimalizace

Keywords

Java, JVM, bytecode, ASM, BCEL, Javassist, size optimization, peephole optimization

Citace

PONCOVÁ, Vendula. *Optimalizace velikosti bajtkódu Javy.* Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Kočí Radek.

Optimalizace velikosti bajtkódu Javy

Prohlášení

Prohlašuji, že jsem tuto semestrální práci vypracovala samostatně pod vedením pana Ing. Radka Kočího, Ph.D. a pana Ing. Pavla Tišnovského, Ph.D. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Vendula Poncová
25. května 2016

Poděkování

Děkuji panu Ing. Radkovi Kočímu za odborné vedení a pomoc při zpracování této práce. Mé poděkování patří též panu Ing. Pavlu Tišnovskému za cenné rady a připomínky.

Obsah

Kapitola 1

Úvod

Kód programu je obvykle optimalizován s cílem minimalizovat dobu běhu programu, ale hlavním požadavkem může být i kratší kód či efektivnější práce s dostupnými prostředky, jak uvádí Aho [1]. Zatímco optimalizace rychlosti je u jazyka Java dobře zpracovaným tématem [7], optimalizace velikosti kódu se řeší především v souvislosti s vestavěnými systémy [5] a obfuskací [3]. Ve vestavěných systémech se však používají specializované edice Javy a obfuskace kódu vede k modifikaci struktury programu. Kratší kód přitom zabírá méně paměti, rychleji se přenáší po síti a může vést i k rychlejšímu běhu programu. Cílem této práce je proto studium bajtkódu Javy SE z hlediska jeho velikosti a návrh metod pro optimalizaci velikosti bajtkódu. Výstupem práce jsou nástroje `jbyca` a `jbyco` pro analýzu a optimalizaci bajtkódu.

V kapitole 2 se věnuji obecné speficikaci bajtkódu Javy. Popisuji virtuální stroj Java Virtual Machine, způsob, jakým je bajtkód interpretován, a zabývám se formátem, v jakém je bajtkód uložen v instrukčních souborech. V kapitole 3 uvádím stručný popis existujících nástrojů pro manipulaci s bajtkódem a shrnuji jejich výhody a nevýhody. Konkrétně zmiňuji BCEL, ASM a Javassist. Tyto nástroje jsem využila při návrhu a implementaci nástroje `jbyca` pro analýzu bajtkódu popsaného v kapitole 4. Nástroj jsem aplikovala na vybraný vzorek dat a získané výstupy zpracovala a vyhodnotila v kapitole 5. Na základě výsledků jsem navrhla metody pro optimalizaci velikosti a implementovala je v nástroji `jbyco`, kterému je věnovaná kapitola 6. Nakonec jsem s užitím tohoto nástroje optimalizovala vzorová data a vyhodnotila účinky optimalizačních metod.

Kapitola 2

Bajtkód jazyka Java

Architektura Javy se podle Vennerse [15] skládá z programovacího jazyka Java, formátu instrukčního souboru, aplikačního programového rozhraní Java Application Programming Interface (Java API) a virtuálního stroje Java Virtual Machine (JVM). Pro psaní zdrojových kódů a jejich spouštění je zapotřebí všech těchto částí. Zdrojový kód zapsaný v programovacím jazyce Java je uložený v souboru s příponou `.java` (dále `java` souboru). Tento kód je při kompilaci převeden na mezikód, tzv. bajtkód, a uložen v souborech s příponou `.class` (dále `class` souborech). Bajtkód lze následně spustit pomocí virtuálního stroje, který má přístup k Java API. V této kapitole popisuji základní charakteristiky JVM a formát jeho instrukčního souboru dle specifikace ve verzi Java SE 8 Edition [9].

2.1 Virtuální stroj JVM

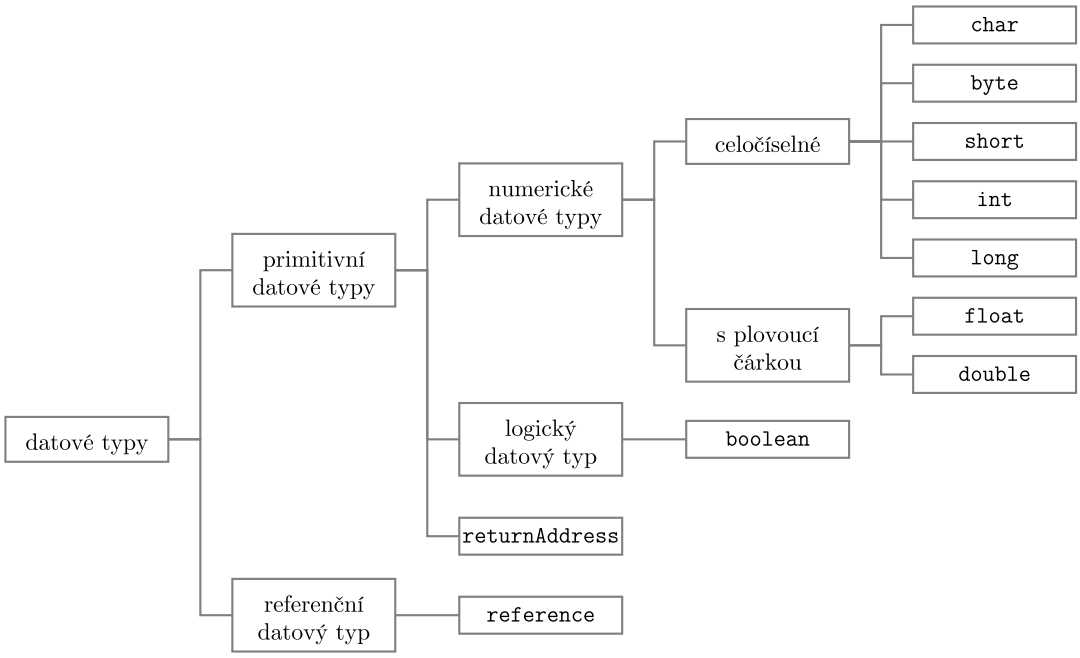
O JVM lze hovořit z hlediska abstraktní specifikace nebo konkrétní implementace. Konkrétní implementace je závislá na daném systému a hardwaru, ale jednotná interpretace `class` souborů napříč platformami je zajištěna dodržením specifikace a platformově nezávislým formátem `class` souborů. Vzhledem k zaměření práce se v této kapitole zabývám abstraktní specifikací JVM.

2.1.1 Datové typy a hodnoty

Datové typy a hodnoty podporované JVM jsou znázorněné na obrázku 2.1. Celá čísla jsou reprezentovaná datovými typy `byte` o šířce 8 bitů, `short` o šířce 16 bitů, `int` o šířce 32 bitů nebo `long` o šířce 64 bitů. Pro čísla s plovoucí řádovou čárkou jsou definovány typy `float` o 32 bitech a `double` o 64 bitech. Typ `boolean` reprezentuje pravdivostní hodnoty pravda a nepravda. Typ `returnAddress` reprezentuje ukazatel na instrukci v instrukčním souboru.

Znaky a řetězce jsou v Javě kódované podle standardu Unicode v kódování UTF-16, kde jeden znak je kódovaný jednou nebo dvěma kódovými jednotkami. Kódovou jednotku reprezentuje typ `char`. Jedná se o 16-bitové nezáporné číslo. Řetězec je pak reprezentovaný pomocí pole hodnot typu `char`. V instrukčním souboru jsou řetězcové konstanty kódované v modifikovaném kódování UTF-8.

Typ `reference` označuje referenční datový typ a reprezentuje referenci na dynamicky vytvořený objekt. Podle toho, zda objekt je instancí třídy, pole nebo instancí třídy či pole, které implementují nějaké rozhraní, se rozlišuje typ reference. Hodnotou typu `reference` může být též speciální hodnota `null`, tedy reference na žádný objekt.



Obrázek 2.1: Datové typy a hodnoty podporované JVM.

Instrukční sada JVM je omezená a nenabízí u všech instrukcí podporu pro všechny datové typy. Celá čísla jsou primárně reprezentovaná datovými typy `int` a `long` a případně přetypovaná na jeden z typů `byte`, `short` či `char`. Pro datový typ `boolean` existuje jen podpora pro přetypování a pro vytvoření pole hodnot typu `boolean`. Pro výpočet logických výrazů se používá typ `int` s hodnotami 0 a 1.

2.1.2 Paměťové oblasti

JVM pracuje s několika typy paměťových oblastí. Velikosti těchto oblastí mohou být pevně dané, nebo se mohou měnit dynamicky podle potřeby. Při spuštění JVM vzniká halda a oblast metod. Halda je paměť určená pro alokaci instancí tříd a polí. Alokovanou paměť nelze dealokovat explicitně. Haldu automaticky spravuje tzv. *garbage collector*. Oblast metod slouží k ukládání kompilovaného kódu. Pro každou načtenou třídu se do této paměti ukládají struktury definující tuto třídu. Jednou z těchto struktur je tzv. *run-time constant pool*. Jedná se o tabulku konstant z `class` souboru, v níž jsou symbolické reference na třídy, metody a členské proměnné nahrazeny konkrétními referencemi. Více se o tabulce konstant zmiňuje kapitola 2.2.2.

Každá aplikace je spuštěna v samostatném vlákně a při běhu mohou vznikat a zanikat i další vlákna. Všechna taková vlákna sdílí přístup k haldě a oblasti metod. Navíc má každé vlákno k dispozici vlastní `pc` registr a zásobník rámců. V `pc` registru je uchováván ukazatel na aktuálně vykonávanou instrukci, není-li aktuálně vykonávaná metoda nativní. Zásobník rámců obsahuje data zavolaných metod. Při každém volání metody je vytvořen nový rámec. Rámec má vlastní pole lokálních proměnných a zásobník operandů. V poli lokálních proměnných se uchovávají hodnoty parametrů a lokálních proměnných. Hodnoty lze vkládat na zásobník operandů, provádět nad nimi výpočty a ukládat zpět do pole. Operační zásobník slouží k předávání operandů instrukcím a k uchovávání mezivýsledků. Podílí se také na předávání parametrů a návratových hodnot volaných metod.

Před voláním metody je třeba nejprve vložit parametry na zásobník operandů aktuálního rámce. Zavoláním metody se vytvoří nový rámec a umístí se na vrchol zásobníku rámců. Parametry se následně přesunou ze zásobníku operandů předchozího rámce do pole lokálních proměnných nového rámce. Registr `pc` se nastaví na první instrukci volané metody a začnou se vykonávat jednotlivé instrukce. Při návratu z metody je návratová hodnota umístěná na vrcholu zásobníku operandů, vrací-li metoda nějakou hodnotu. Tato hodnota je přesunuta na zásobník operandů předcházejícího rámce a aktuální rámec je ze zásobníku rámců odstraněn. Dojde k obnovení stavu volající metody. Registr `pc` je nastaven na index instrukce, která bezprostředně následuje za instrukcí volající metodu. Pokud je metoda ukončena vyvoláním nezachycené výjimky, pak k předání hodnoty nedochází. Aktuální rámec je odstraněn a výjimka je znovu vyvolaná ve volající metodě. Zpracování výjimek je více vysvětleno v kapitole 2.2.5.

Nejmenším prvkem, se kterým pracuje zásobník operandů, je 32-bitová hodnota. Hodnoty většiny datových typů lze vyjádřit pomocí jediného prvku, ale hodnoty typů `long` a `double` je třeba reprezentovat dvěma prvky. S takovou dvojicí je třeba vždy manipulovat jako s celkem. Datový typ hodnoty na zásobníku je daný instrukcí, která ho tam vložila, a na hodnotu nelze nahlížet jinak. Hloubka zásobníku operandů je určená počtem prvků na zásobníku. Lze tedy definovat pojem jednotka hloubky zásobníku, kdy jedna jednotka odpovídá jednomu prvku za zásobníku.

Lokální proměnná v poli lokálních proměnných je 32-bitová hodnota. Proměnnou lze adresovat pomocí indexu do pole, kde pole je indexováno od nuly. Lokální proměnná může být typu `byte`, `short`, `int`, `char`, `float`, `boolean`, `reference` nebo `returnAddress`. Hodnoty typu `long` a `double` jsou uchovávané pomocí dvojice lokálních proměnných. V tom případě k adresaci slouží nižší z indexů a na větší index se nesmí přistupovat.

2.1.3 Kontrola instrukčního souboru

Při načítání `class` souboru je ověřeno správné formátování tak, jak je popsáno v kapitole 2.2. Jsou zkontrolovány první čtyři bajty souboru, předdefinované atributy musí být správné délky, názvy a typy tříd, rozhraní, metod a proměnných musí být validní, indexy do tabulky konstant musí adresovat správný typ položky. Dále jsou kladena jistá omezení na kód metod. Mimo jiné, argumenty instrukcí musí mít správný typ a musí být správného počtu, integrita hodnot typu `long` a `double` nemůže být nikdy narušena, k hodnotě lokální proměnné se nesmí přistupovat před inicializací proměnné a nesmí dojít k načtení hodnoty z prázdného zásobníku operandů, či překročení jeho maximální velikosti. Metody musí být ukončené některou z instrukcí pro návrat z metody. Součástí verifikace těchto omezení je typová kontrola nebo typová inference. Díky kontrole instrukčního souboru se omezení nemusí kontrolovat za běhu programu.

2.2 Formát instrukčního souboru

Při kompilaci `java` souboru překladač pro každou definovanou třídu a rozhraní vytvoří jeden `class` soubor. Tento soubor obsahuje binární reprezentaci kompilovaného mezikódu, který lze interpretovat prostřednictvím JVM. Tato kapitola je věnovaná popisu formátu `class` souboru.

Pro popis formátu jsem zvolila rozšířenou Backus-Naurovu formu, která umožňuje zapsat syntaxi formálního jazyka pomocí pravidel a terminálních a neterminálních symbolů.

Nonterminální symboly jsou definovány pomocí definujícího symbolu `:=`, symbolu pro konkatenaci `,`, symbolu pro alternaci |, symbolů pro nula a více opakování `{}`, ukončujícího symbolu `;` a pomocí graficky odlišených `terminálních` a *nonterminálních* symbolů. Symboly popisují jednotlivé struktury, ze kterých se `class` soubor skládá.

2.2.1 Základní struktura

Položky `class` souboru tvoří posloupnost bajtů. Základní stavební jednotkou je tedy bajt, který je v pravidlech reprezentovaný symbolem B. Symbol $\langle n \rangle B$, kde $\langle n \rangle \in \{2, 3, \dots\}$, reprezentuje n bajtů. Terminály začínají prefixem `0x` a jsou hexadecimální reprezentací posloupnosti bajtů.

Základní struktura souboru je popsaná symbolem *classfile*. Soubor obsahuje informace o svém typu a verzi (*version*), disponuje tabulkou všech konstant, které se v souboru vyskytují (*constants*), nese informace o třídě či rozhraní (*class*), které reprezentuje, obsahuje seznam rozhraní (*interface_list*), které reprezentovaná třída implementuje, případně rozhraní rozšiřuje, seznam členských proměnných (*field_list*), seznam metod (*method_list*) a seznam atributů (*attribute_list*).

classfile := *version*, *constants*, *class*, *interface_list*, *field_list*, *method_list*, *attribute_list*;

Typ souboru je definován prvními čtyřmi bajty, které jsou popsané symbolem *magic_number*. Verze souboru je tvořena hodnotou M symbolu *major_version* a hodnotou m symbolu *minor_version* jako $M.m$.

version := *magic_number*, *minor_version*, *major_version*;
magic_number := `0xCAFEBABE`;
minor_version := *2B*;
major_version := *2B*;

2.2.2 Konstanty

Tabulka konstant obsahuje některé číselné konstanty, všechny řetězce a symbolické informace o všech třídách, rozhraních, metodách a členských proměnných, které se v souboru, instrukcích i atributech vyskytují. Tato tabulka se nazývá *constant pool* a slouží v podstatě jako databáze dat, do které se pomocí indexů odkazují další položky souboru. Odkazem do tabulky konstant je tedy v dalším textu myšlen platný index do tabulky konstant adresující položku očekávaného typu.

Struktura tabulky konstant je popsána symbolem *constants*. Symbol *constant_pool_count* reprezentuje hodnotu $1 + n$, kde n je počet položek v tabulce konstant. Položky tabulky jsou indexované od jedné, neboť nultý index je vyhrazen pro odkaz na žádnou z položek. Samotná tabulka je reprezentovaná symbolem *constant_pool*.

Každá položka tabulky je tvořena označením typu a posloupností bajtů s informacemi o položce. Položky mohou mít různou velikost v závislosti na svém typu a obsahu. Stejným způsobem jsou definovány všechny tabulky v `class` souboru. Jestliže se jedná o pole, pak prvky pole jsou stejného typu, a proto označení typu v prvcích chybí.

Číselné konstanty jsou reprezentované symboly *constant_integer*, *constant_float*, *constant_long* a *constant_double* a skládají se jen z typu a číselné hodnoty. Symbol *constant_utf8* reprezentuje řetězec v upraveném kódování `UTF-8` a skládá se z typu, délky

$$
\begin{aligned}
\textit{constants} &:= \textit{constant\_pool\_count}, \textit{constant\_pool};\\
\textit{constant\_pool\_count} &:= \textit{2B};\\
\textit{constant\_pool} &:= \{\ \textit{constant\_integer}\\
&\quad |\ \textit{constant\_float}\\
&\quad |\ \textit{constant\_long}\\
&\quad |\ \textit{constant\_double}\\
&\quad |\ \textit{constant\_utf8}\\
&\quad |\ \textit{constant\_string}\\
&\quad |\ \textit{constant\_nameAndType}\\
&\quad |\ \textit{constant\_class}\\
&\quad |\ \textit{constant\_fieldref}\\
&\quad |\ \textit{constant\_methodref}\\
&\quad |\ \textit{constant\_interfaceMethodref}\\
&\quad |\ \textit{constant\_methodHandle}\\
&\quad |\ \textit{constant\_methodType}\\
&\quad |\ \textit{constant\_invokeDynamic}\\
&\quad \};
\end{aligned}
$$

pole bajtů a pole bajtů nesoucích reprezentaci řetězce. Znaky řetězce mohou být vzhledem ke kódování tvořeny různými počty bajtů. Symbol *constant_string* je reprezentací řetězcové konstanty a kromě typu obsahuje index do tabulky konstant na položku *constant_utf8*. Třídy a rozhraní jsou reprezentované položkami *constant_class* s odkazy na jejich název (*constant_utf8*). Entity jako členské proměnné (i statické), metody třídy a metody rozhraní jsou reprezentované položkami *constant_fieldref*, *constant_methodref* a *constant_interfaceMethodref* obsahujícími odkaz na třídu, případně rozhraní, dané entity (*constant_class*), a odkaz na položku se jménem a typem této entity (*constant_nameAndType*). Jméno a typ v položce *constant_nameAndType* jsou odkazy na řetězce (*constant_utf8*). Položky *constant_methodHandle*, *constant_methodType* a *constant_invokeDynamic* souvisí s podporou dynamických jazyků.

Názvy tříd a rozhraní jsou interně uváděné v úplném tvaru, ale z historických důvodů se tečky nahrazují lomítky. Například, třída `Object` má úplný název `java.lang.Object` a interní název `java/lang/Object`. Typ proměnné nebo metody je specifikován řetězcem. Základní datové typy jsou reprezentované písmeny `B` pro `byte`, `C` pro `char`, `D` pro `double`, `F` pro `float`, `I` pro `int`, `J` pro `long`, `S` pro `short`, `Z` pro `boolean`. Typ reference na objekt je reprezentovaný řetězcem `L`$ClassName$`;`, kde $ClassName$ je interní název třídy nebo rozhraní. Typ reference na jednorozměrné pole je reprezentovaný řetězcem `[`$ComponentType$, kde $ComponentType$ je řetězec reprezentující základní datový typ, referenci na objekt nebo referenci na pole. Pomocí zanoření referencí na pole lze definovat referenci na vícerozměrné pole. Například, řetězec `Ljava/lang/Object;` označuje referenci na objekt typu `Object` a `[[[I` označuje referenci na trojrozměrné pole typu `int`. Typ metody je reprezentován řetězcem, který se skládá z výčtu typů formálních parametrů metody a typu její návratové hodnoty. Tedy například, typ metody s hlavičkou `int method(boolean b, Object o)` bude specifikovaný řetězcem `(BLjava/lang/Object;)I`. Nemá-li metoda žádné parametry či nevrací žádnou hodnotu, pak je chybějící typ nahrazen písmenem `V`.

2.2.3 Třída

Každý `class` soubor reprezentuje jednu třídu nebo rozhraní. Informace o reprezentované entitě jsou definované symbolem *class*. Položka *this_class* je odkazem na tuto entitu v tabulce konstant, *super_class* je odkaz na nadřazenou třídu a *access_flags* je bitové pole příznaků pro přístup k této entitě.

$$
\begin{aligned}
\textit{class} &:= \textit{access\_flags}, \textit{this\_class}, \textit{super\_class};\\
\textit{access\_flags} &:= \textit{2B};\\
\textit{this\_class} &:= \textit{class\_ref};\\
\textit{super\_class} &:= \textit{class\_ref};\\
\textit{class\_ref} &:= \textit{constant\_pool\_index};\\
\textit{constant\_pool\_index} &:= \textit{2B};
\end{aligned}
$$

Seznam rozhraní, které reprezentovaná třída implementuje, případně reprezentované rozhraní rozšiřuje, je definované v poli *interfaces* o *interface_count* prvcích. Prvky jsou odkazy do tabulky konstant na položky *constant_class* reprezentující nějaké rozhraní.

$$
\begin{aligned}
\textit{interface\_list} &:= \textit{interface\_count}, \textit{interfaces};\\
\textit{interfaces} &:= \{\ \textit{class\_ref}\ \};\\
\textit{interface\_count} &:= \textit{2B};
\end{aligned}
$$

2.2.4 Členské proměnné

Členské proměnné (proměnné třídy i proměnné instance) jsou definované v poli členských proměnných *fields* o *fields_count* prvcích. Každá proměnná má pole příznaků dané symbolem *access_flags*, jméno dané symbolem *name_ref*, typ daný symbolem *descriptor_ref* a seznam atributů daný symbolem *attribute_list*. Jméno a typ jsou reprezentované odkazem do tabulky konstant na položku *utf8_ref*. Atributům se věnuje kapitola 2.2.6.

$$
\begin{aligned}
\textit{field\_list} &:= \textit{fields\_count}, \textit{fields};\\
\textit{fields} &:= \{\ \textit{field\_info}\ \};\\
\textit{field\_info} &:= \textit{access\_flags}, \textit{name\_ref}, \textit{descriptor\_ref}, \textit{attribute\_list};\\
\textit{fields\_count} &:= \textit{2B};\\
\textit{name\_ref} &:= \textit{utf8\_ref};\\
\textit{descriptor\_ref} &:= \textit{utf8\_ref};\\
\textit{utf8\_ref} &:= \textit{constant\_pool\_index};
\end{aligned}
$$

2.2.5 Metody

Metody reprezentované třídy či rozhraní jsou definované v poli metod *methods* o *methods_count* prvcích. Stejně jako u členských proměnných jsou metody popsané bitovým polem příznaků, jménem, typem a seznamem atributů.

$$
\begin{aligned}
\textit{method\_list} &:= \textit{methods\_count}, \textit{methods};\\
\textit{methods} &:= \{\ \textit{method\_info}\ \};\\
\textit{method\_info} &:= \textit{access\_flags}, \textit{name\_ref}, \textit{descriptor\_ref}, \textit{attribute\_list};\\
\textit{methods\_count} &:= \textit{2B};
\end{aligned}
$$

Pokud metoda není abstraktní, pak jedním z jejích atributů je `Code` s kódem metody. Tento atribut je definován symbolem *code_attribute*. Položka *name_ref* je odkazem do tabulky konstant na řetězec „Code“. Položka *attribute_length* určuje délku atributu v bajtech bez prvních šesti bajtů. Dále položky *max_stack* a *max_locals* označují maximální hloubku zásobníku operandů přepočtenou na jednotku hloubky a maximální počet lokálních proměnných včetně parametrů. Kód metody je reprezentovaným polem bajtů *code* o délce *code_length*. Symbol *attribute_list* je seznamem atributů.

code_attribute	:=	*name_ref*, *attribute_length*, *code_info*
code_info	:=	*max_stack*, *max_locals*, *code_list*, *exception_list*, *attribute_list*;
code_list	:=	*code_length*, *code* ;
code	:=	{ *B* };
max_stack	:=	*2B*;
max_locals	:=	*4B*;
code_length	:=	*4B* ;

Informace o zpracování výjimek jsou dostupné v tabulce výjimek *exception_table* o délce *exception_table_length*. Každá položka tabulky obsahuje dva indexy *start_pc* a *end_pc* do pole *code*, jež společně definují blok instrukcí, pro který je odchycení dané výjimky aktivní. Dále obsahuje index *handler_pc* do pole *code* odkazující na začátek bloku pro zpracování výjimky a nakonec index *catch_type* do tabulky konstant na položku *constant_class* reprezentující typ odchycené výjimky. Jestliže je tento index nulový, pak jsou odchytávány všechny výjimky. Na pořadí položek v tabulce výjimek se nevztahují žádná omezení, neboť při odchytávání výjimky se postupuje od nejniternějšího bloku.

exception_list	:=	*exception_table_length*, *exception_table* ;
exception_table	:=	{ *start_pc, end_pc, handler_pc, catch_type* };
start_pc	:=	*code_index*;
end_pc	:=	*code_index*;
handler_pc	:=	*code_index*;
catch_pc	:=	*class_ref*;
exception_table_length	:=	*2B*;
code_index	:=	*2B*;

2.2.6 Atributy

Reprezentovaná třída, případně rozhraní, metody, členské proměnné a i některé atributy mají definovaný seznam atributů. Seznam se skládá z tabulky atributů *attributes* o *attributes_count* položkách. Typ atributu je daný odkazem *name_ref* na název atributu, délka atributu bez prvních šesti bajtů je daná hodnotou *attribute_length*. Další informace, které atribut nese v položce *info*, se liší podle typu atributu.

attribute_list	:=	*attributes_count*, *attributes*;
attributes	:=	{ *name_ref*, *attribute_length*, *info* };
info	:=	{ *B* };
attributes_count	:=	*2B*;
attribute_length	:=	*4B*;

Specifikace [9] definuje 23 atributů. Překladače však mohou definovat a vkládat do `class` souborů i atributy vlastní. Pokud je JVM neumí rozpoznat, pak je ignoruje. Atributy mají různou míru důležitosti vzhledem k interpretaci `class` souboru. Pro správnou interpretaci virtuálním strojem je důležitých následujících pět atributů.

`ConstantValue` může být atributem členské proměnné. Jeho součástí je index do tabulky konstant na položku s číselnou nebo řetězcovou konstantou. Jestliže je daná proměnná statická, pak je jí při inicializaci třídy přiřazena právě tato hodnota.

`Code` obsahuje kód metody, které je atributem, a byl představen v kapitole 2.2.5.

`StackMapTable` může být jedním z atributů `Code`. Je důležitý pro typovou kontrolu při verifikaci `class` souborů. Pro každý základní blok instrukcí obsahuje rámec s typy lokálních proměnných a s typy hodnot na zásobníku operandů. U starších verzí `class` souboru tento atribut chybí, a proto se provádí typová inference pomocí analýzy datového toku.

`Exceptions` je atribut metody. Obsahuje odkazy do tabulky konstant na typy kontrolovaných výjimek, které metoda může vyhodit.

`BootstrapMethods` souvisí s dynamickými jazyky.

Dalších dvanáct atributů je podstatných pro správnou interpretaci knihovnami Java API. Nesou informace o dané třídě, které mohou být dostupné za běhu programu prostřednictvím reflexe.

`InnerClasses` obsahuje seznam vnitřních tříd třídy reprezentované `class` souborem. Pro každou vnitřní třídu atribut uchovává bitové pole příznaků, ukazatel na název vnitřní třídy, ukazatel na vnější třídu a ukazatel na vnitřní třídu.

`EnclosingMethod` je atribut každé lokální nebo anonymní třídy. Obsahuje ukazatel na vnější třídu a ukazatel na metodu, která definici třídy uzavírá.

`Synthetic` reprezentuje příznak, že daný člen třídy se nevyskytuje ve zdrojovém kódu a zároveň není standardním členem.

`Signature` nese deklaraci třídy, rozhraní, členské proměnné nebo metody, v níž se vyskytují typové proměnné nebo parametrizované typy.

`Runtime`*VisibilityAnnotations* jsou atributy obsahující seznamy anotací s danou viditelností a z dané skupiny. *Annotations* označuje jednu z následujících skupin anotací: `Annotations` pro anotace tříd, členských proměnných a metod, `TypeAnnotations` pro anotace typů a `ParameterAnnotations` pro anotace formálních parametrů metod. *Visibility* určuje, zda je anotace za běhu programu viditelná `Visible` či neviditelná `Invisible`.

`AnnotationDefault` reprezentuje výchozí hodnotu elementu, který patří typu anotace.

`MethodParameters` může být atributem metody a obsahuje jména a přístupové příznaky jeho formálních parametrů.

Zbývající atributy jsou pouze informativní a mohou sloužit například k ladění chyb ve zdrojovém souboru. Obsahují informace o zdrojovém kódu a lokálních proměnných.

`SourceFile` obsahuje odkaz na název souboru se zdrojovým kódem, jehož překladem vznikl daný `class` soubor.

`SourceDebugExtension` v sobě nese řetězec s ladícími informacemi.

`LineNumberTable` reprezentuje mapování indexů do pole instrukcí na čísla řádků zdrojového kódu.

`LocalVariableTable` obsahuje informace o lokálních proměnných metody. Pro každou proměnnou uchovává rozsah instrukcí, ve kterém proměnná nese hodnotu, odkaz na název proměnné, odkaz na typ proměnné a index do pole lokálních proměnných.

`LocalVariableTypeTable` nese stejné informace jako atribut `LocalVariableTable`, ale pouze pro proměnné, jejichž typy používají typované proměnné nebo parametrizované typy.

`Deprecated` slouží k indikaci toho, že třída, metoda, členská proměnná či rozhraní jsou zastaralé.

2.2.7 Instrukce

Každá instrukce se skládá z jednobajtového operačního kódu *opcode* a nula a více operandů. Instrukce může dále pracovat s obsahem zásobníku operandů a má přístup do pole lokálních proměnných a tabulky konstant. Pro snazší orientaci je každému operačnímu kódu přiřazen jednoznačný název *mnemonic*. Součástí *mnemonic* může být označení datového typu, s jehož hodnotami instrukce pracuje. Typ je specifikován písmeny `i` pro `int`, `l` pro `long`, `s` pro `short`, `b` pro `byte`, `c` pro `char`, `f` pro `float`, `d` pro `double` a `a` pro `reference`. V následujícím textu jsou instrukce uvedené ve tvaru: *mnemonic* $operand_1$ $operand_2$ ... $operand_n$. Pokud nebude řečeno jinak, pak každý operand má velikost jednoho bajtu.

Konstantní hodnoty

Konstantní hodnotu lze dle jejího typu, velikosti a hodnoty vložit na zásobník několika způsoby. Instrukce `aconst_null` vloží na zásobník referenci na `null`. Instrukce `iconst_`*value*, kde *value* $\in \{\texttt{m1}, \texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}\}$ reprezentuje číselnou hodnotu v rozsahu -1 až 5, vloží na zásobník odpovídající hodnotu typu `int`. Obdobně lze instrukcí `fconst_`*value*, kde *value* $\in \{\texttt{0}, \texttt{1}, \texttt{2}\}$, a instrukcemi `lconst_`*value* a `dconst_`*value*, kde *value* $\in \{\texttt{0}, \texttt{1}\}$, vložit konstantní hodnoty typu `float`, `long` a `double`. Větší celočíselnou hodnotu typu `int` umožňují vložit instrukce `bipush` *value* a `sipush` *value*, kde *value* je jednobajtová, respektive dvoubajtová celočíselná hodnota se znaménkem, tj. -128 až 127 pro `bipush` a -32 768 až 32 767 pro `sipush`.

Ve všech ostatních případech je nutné vložit hodnotu z tabulky konstant. Pomocí instrukce `ldc` *index*, kde *index* je jednobajtový index do tabulky konstant, lze vložit hodnotu typu `int` či `float` nebo referenci na objekt. Instrukce `ldc_w` *index* umožňuje použít dvoubajtový index. Instrukce `ldc2_w` *index* vloží na zásobník hodnotu typu `long` nebo `double` danou dvoubajtovým indexem *index*.

Práce s lokálními proměnnými

Do lokální proměnné lze přiřadit hodnotu instrukcí *t*`store` *index*, kde $t \in \{\texttt{i}, \texttt{l}, \texttt{f}, \texttt{d}, \texttt{a}\}$ a *index* je index do pole lokálních proměnných. Dané proměnné se přiřadí hodnota, která

se odebere z vrcholu zásobníku. Na druhou stranu, instrukce *t*`load` *index*, načte hodnotu z dané lokální proměnné na zásobník. Pro lokální proměnné s indexy $index \in \{0, 1, 2, 3\}$ lze použít jednobajtové instrukce *t*`store_`*index* a *t*`load_`*index*. Celočíselné lokální proměnné lze inkrementovat instrukcí `iinc` *index value*, která k hodnotě proměnné na indexu *index* přičte jednobajtovou celočíselnou hodnotu *value*.

Práce s polem

Pole lze vytvořit instrukcí `newarray` *type*, kde *type* označuje typ pole. Instrukce načte ze zásobníku hodnotu *count* typu `int`, vytvoří pole typu *type* o délce *count* a referenci na toto pole vloží na zásobník. Pole objektů umožňuje vytvořit instrukce `anewarray` *index*, kde *index* je dvoubajtový index do tabulky konstant na typ vytvářeného pole. Typem zde může být třída, rozhraní nebo pole. Obdobně lze vytvořit vícerozměrné pole objektů instrukcí `multianewarray` *index dimension*, kde *index* je opět index do tabulky konstant a *dimension* udává počet dimenzí vytvářeného pole. Ze zásobníku jsou načteny délky pro jednotlivé dimenze a je vložena reference na vícerozměrné pole objektů daného typu.

Instrukce *t*`astore`, kde $t \in \{\texttt{b}, \texttt{c}, \texttt{s}, \texttt{i}, \texttt{l}, \texttt{f}, \texttt{d}, \texttt{a}\}$ určuje typ pole, umožňuje vložit hodnotu do pole. Ze zásobníku odebere hodnotu *value*, index *index* a referenci na pole *array* typu *t* a provede operaci $array[index] := value$. Instrukcí *t*`aload` lze hodnotu z pole načíst na zásobník. Ze zásobníku se odebere *index* a *array* a vloží se na něj hodnota $array[index]$.

Délku pole lze zjistit instrukcí `arraylength`, která ze zásobníku odebere referenci na pole a vloží na něj délku tohoto pole.

Metody a objekty

Nový objekt lze vytvořit instrukcí `new` *index*, kde *index* je dvoubajtový index do tabulky konstant na třídu nebo rozhraní. Instance třídy nebo rozhraní se inicializuje a její reference se vloží na zásobník.

Přístup k proměnným instance umožňují instrukce `getfield` *index* a `putfield` *index*, kde *index* je dvoubajtový index do tabulky konstant na členskou proměnnou. Instrukce `getfield` odebere ze zásobníku referenci na objekt, získá hodnotu dané členské proměnné a vloží ji na zásobník. Instrukce `putfield` odebere ze zásobníku hodnotu *value* a referenci na objekt a dané členské proměnné tohoto objektu přiřadí hodnotu *value*. Obdobně lze přistupovat k proměnným třídy pomocí instrukcí `getstatic` *index* a `putstatic` *index*.

Metodu lze zavolat instrukcí `invokevirtual` *index*, kde *index* je dvoubajtový index do tabulky konstant na metodu. Instrukce ze zásobníku odebere parametry metody včetně reference na objekt, který bude sloužit jako parametr `this`, a zavolá příslušnou metodu. Obdobně instrukce `invokestatic` volá statickou metodu, `invokeinterface` volá metodu rozhraní a `invokedynamic` volá dynamickou metodu. Pro ostatní metody je třeba použít instrukci `invokespecial`.

Instrukce `instanceof` *index* umožňuje ověřit, zda je objekt daný referencí z vrcholu zásobníku instancí třídy dané dvoubajtovým indexem do tabulky konstant *index*, případně zda objekt implementuje rozhraní dané tímto indexem. Výsledek ověření je vložen na zásobník v podobě hodnoty typu `int` (1 úspěch, 0 neúspěch). Obdobně se chová instrukce `checkcast` *index*, ale v případě úspěchu vloží referenci na objekt zpět na zásobník, v případě neúspěchu vyhodí výjimku `ClassCastException`.

Výjimku lze vyhodit instrukcí `athrow`. Ze zásobníku se odebere reference na instanci třídy `Throwable` nebo její podtřídu a v tabulce výjimek se vyhledá blok pro zpracování této instance. Pokud pro danou výjimku takový blok neexistuje, vykonávání aktuální metody

method se okamžitě bez předání návratové hodnoty ukončí a výjimka se znovu vyvolá v metodě, která metodu *method* zavolala.

Vstupu do synchronizovaného bloku instrukcí předchází vstup do monitoru daného objektu. Opuštění takového bloku znamená uvolnění tohoto monitoru. Toto chování zajišťují instrukce `monitorenter` a `monitorexit`. Referenci na objekt, s jehož monitorem budou pracovat, získávají ze zásobníku operandů.

Konverze hodnot

Hodnotu z vrcholu zásobníku lze konvertovat na jiný datový typ instrukcí typu $t_1 2 t_2$, kde $t_1, t_2 \in \{\texttt{i}, \texttt{l}, \texttt{f}, \texttt{d}\}$ a pro t_1 rovno `i` navíc $t_2 \in \{\texttt{b}, \texttt{c}, \texttt{s}\}$. Hodnota je pak konvertovaná z typu t_1 na typ t_2.

Matematické a bitové operace

Instrukce pro matematické operace jsou ve tvaru *toperation*, kde $t \in \{\texttt{i}, \texttt{l}, \texttt{f}, \texttt{d}\}$ specifikuje typ operandů a *operation* $\in \{\texttt{add}, \texttt{sub}, \texttt{mul}, \texttt{div}, \texttt{rem}, \texttt{neg}\}$ určuje jednu z matematických operací pro součet, rozdíl, násobení, dělení, zbytek po dělení a negaci. Instrukce pro bitové operace jsou ve tvaru *toperation*, kde $t \in \{\texttt{i}, \texttt{l}\}$ a *operation* $\in \{\texttt{shl}, \texttt{shr}, \texttt{ushr}, \texttt{and}, \texttt{or}, \texttt{xor}\}$ označuje bitový posuv doleva, aritmetický posuv doprava, logický posuv doprava, logický součin, logický součet nebo exkluzivní logický součet. Uvedené instrukce odeberou ze zásobníku příslušný počet operandů a vrátí na zásobník výsledek operace.

Podmíněné skoky

Instrukce `if`*condition next*, kde *condition* $\in \{\texttt{eq}, \texttt{ne}, \texttt{lt}, \texttt{le}, \texttt{ge}, \texttt{gt}\}$ a *next* je dvoubajtová znaménková hodnota, umožňuje provést podmíněný skok na jinou instrukci. Ze zásobníku odebere hodnotu typu `int` a porovná ji s nulou na rovnost, nerovnost, menší než, menší nebo rovno, větší než či větší nebo rovno dle *condition*. Pokud je podmínka pro skok splněna, pokračuje se instrukcí ve vzdálenosti *next* od pozice aktuální instrukce. Jinak se pokračuje následující instrukcí. Instrukce `if_icmp`*condition* umožňuje vzájemně porovnat dvě hodnoty typu `int`. Pro hodnoty typu `long`, `float` a `double` je nutné nejprve provést jednu z instrukcí `lcmp`, `fcmp`x a `dcmp`x, kde $x \in \{\texttt{l}, \texttt{g}\}$. Instrukce ze zásobníku odebere dvě hodnoty, porovná je a výsledek porovnání vloží na zásobník (1 pro větší než, 0 pro rovnost, -1 pro menší než). Podmíněný skok lze následně vykonat instrukcí `if`*condition*. Pro porovnání objektu s `null` jsou k dispozici instrukce `ifnull` a `ifnonnull`. Pro porovnání dvou objektů lze použít instrukce `if_acmpeq` a `if_acmpne`.

Příkaz `switch` se převádí na jednu z instrukcí `tableswitch` a `lookupswitch`. První instrukce pracuje s tabulkou relativních adres, kde vstupní hodnota lze přímo převést na index do tabulky. Druhá instrukce pracuje s tabulkou dvojic klíč-adresa, kde pro vstupní hodnotu je třeba nalézt dvojici s odpovídajícím klíčem. Tabulka adres je vhodnější, nejsou-li jednotlivé případy příkazu `switch` navzájem příliš rozptýlené, jinak je lepší použít tabulku dvojic.

První instrukce je definovaná jako `tableswitch` *pad default low high table*, kde *pad* je výplň o délce nula až tři bajty, která slouží ke správnému zarovnání dalších položek, *default*, *low* a *high* jsou čtyřbajtové hodnoty a *table* je sekvence čtyřbajtových hodnot o délce *high* - *low* + 1. Instrukce načte ze zásobníku hodnotu *value* typu `int` a ověří, zda leží v rozsahu hodnot *low* a *high*. Pokud ne, pak pro skok použije relativní adresu *default*,

pokud ano, pak použije adresu z tabulky relativních adres *table* na pozici *value* - *low*. Následně se provede skok.

Podoba druhé instrukce je `lookupswitch` *pad default count pairs*, kde *count* je čtyřbajtová hodnota označující počet dvojic v tabulce *pairs* a *pairs* je sekvence dvojic čtyřbajtových hodnot *key* a *next*. Dvojice jsou v tabulce seřazené podle hodnoty *key*. Instrukce načte ze zásobníku hodnotu *value* typu `int`, vyhledá v tabulce *pairs* dvojici, kde *key* je rovno *value*, a odpovídající hodnotu *next* použije jako relativní adresu skoku. Pokud takovou dvojici nenajde, skočí na relativní adresu *default*.

Nepodmíněné skoky

Návrat z metody umožňuje instrukce `return`. Jestliže metoda vrací hodnotu, pak je třeba tuto hodnotu vložit na zásobník a zavolat instrukci t`return`, kde $t \in \{$`i`, `l`, `f`, `d`, `a`$\}$ označuje typ návratové hodnoty. Instrukce `goto` *next*, kde *next* je dvoubajtová relativní adresa, provede skok na instrukci na dané adrese. Podobně instrukce `goto_w` *next* umožňuje skočit na čtyřbajtovou relativní adresu. Instrukce `jsr`, `jsr_w` a `ret` slouží k obsluze podprogramu. Používají se k implementaci bloku *finally*.

Práce se zásobníkem

Níže zmíněné instrukce umožňují manipulovat se zásobníkem. Způsob manipulace je popsaný pomocí tzv. jednotek délky zásobníku, přičemž některé hodnoty na zásobníku se mohou skládat ze dvou jednotek. Po provedení instrukce musí být vždy zachována integrita těchto hodnot. Porušení integrity by bylo odhaleno při verifikaci `class` souboru.

K odstranění hodnot z vrcholu zásobníku slouží instrukce `pop` a `pop2`, které odstraní jednu, respektive dvě jednotky. Instrukce `dup` duplikuje jednotku na vrcholu zásobníku, zatímco instrukce `dup_x1` a `dup_x2` duplikovanou jednotku navíc přesunou o dvě, respektive tři jednotky níže. Obdobně instrukce `dup2` duplikuje dvojici jednotek na vrcholu zásobníku a instrukce `dup2_x1` a `dup2_x2` dvojici navíc přesunou o tři, respektive čtyři jednotky níže. Instrukce `swap` prohodí pořadí dvou jednotek na vrcholu zásobníku.

Další instrukce

Instrukce `nop` nic nedělá a nemá žádný efekt. V instrukční sadě je zahrnuta zejména pro úplnost, ale dle Engela [6] může být užitečná například při generování kódu. Sada dále obsahuje instrukce `breakpoint`, `impdep1` a `impdep2` rezervované pro interní užití v JVM. Význam těchto instrukcí není a nebude definovaný, což dává prostor ke specifickému rozšíření funkcionality JVM. Instrukce `breakpoint` je zamýšlena k implementaci zarážek v ladících programech.

Kapitola 3

Nástroje pro manipulaci s bajtkódem

Pro další studium bajtkódu Javy bylo potřeba zvolit vhodný způsob, jakým lze s bajtkódem pracovat. Vzhledem k tomu, že bajtkód je strojový kód, tak je přímá manipulace prakticky nemožná. Proto je vhodnější využít některý z existujících nástrojů. Tato kapitola je věnovaná třem běžně užívaným knihovnám BCEL, ASM a Javassist. Zvolené knihovny jsou implementované v programovacím jazyce Java a liší se navzájem mírou abstrakce a způsobem manipulace s bajtkódem.

3.1 BCEL

BCEL [13] nebo-li Byte Code Engineering Library je knihovna, která je součástí projektu Apache Commons. Je poskytovaná pod licencí Apache License 2.0. Poslední verze BCEL 5.2 nepodporuje Javu 8, ale z repozitáře je dostupná verze 6.0, kde je podpora z větší části implementovaná. Vývoj knihovny však v posledních letech není příliš aktivní.

Programové rozhraní knihovny je dostupné v balíčku `org.apache.bcel`. Knihovna obsahuje třídy pro statický popis `class` souborů, třídy pro dynamické úpravy a vytváření bajtkódu a třídy s užitečnými nástroji. Syntaktickou analýzu `class` souboru a vytvoření reprezentace jeho obsahu v podobě instance třídy `JavaClass` umožňuje třída `ClassParser` z balíčku `org.apache.bcel.classfile`. Součástí balíčku jsou současně všechny třídy podílející se na popisu obsahu souboru. Pro každou položku souboru je tedy vytvořen nový objekt. Takový přístup může být velmi neefektivní, zejména pokud je třeba zpracovat velké množství souborů. Na druhou stranu třída `JavaClass` velmi přesně kopíruje formát `class` souboru tak, jak byl popsán v kapitole 2.2, včetně tabulky konstant. Pro dynamické vytváření a úpravu bajtkódu je třeba vyšší míra abstrakce. Tu poskytují třídy z balíčku `org.apache.bcel.generic`. Pomocí těchto tříd je třeba sestavit celý obsah `class` souboru včetně tabulky konstant. Korektnost výsledného bajtkódu lze zkontrolovat třídou `Verifier`.

Knihovna BCEL poskytuje pro bajtkód velmi nízkou úroveň abstrakce. Je třeba být seznámen s formátem `class` souborů a pracovat s tabulkou konstant. Bajtkód je navíc reprezentovaný velkým množstvím objektů a neexistuje efektivní způsob, jak zpracovat jen ty informace, které jsou pro danou aplikaci potřeba. Vhodnou alternativou je proto knihovna ASM.

3.2 ASM

ASM [12] je knihovna od OW2 Consortium poskytovaná pod licencí BSD. Na rozdíl od BCEL se jedná o aktivní projekt a Java 8 je oficiálně plně podporovaná. ASM si zakládá na snadné použitelnosti, výkonnosti a malé velikosti. Knihovna je založena na návrhovém vzoru Návštěvník (Visitor). Místo reprezentace `class` souboru pomocí objektů jsou při syntaktické analýze volány pro jednotlivé položky metody návštěvníka. Návštěvník může položky zpracovat a předat je dalšímu návštěvníkovi. Pomocí takového zřetězení lze jedním až dvěma průchody `class` souboru dosáhnout požadovaného zpracování bajtkódu. Pokud je třeba provést větší počet průchodů, může být vhodnější použít objektovou reprezentaci pomocí stromu. ASM umožňuje oba přístupy libovolně kombinovat.

Základní rozhraní je dostupné v balíčku `org.objectweb.asm`. Třída `ClassReader` analyzuje daný `class` soubor a volá metody návštěvníka, instance třídy rozšiřující abstraktní třídu `ClassVisitor`. Třída `ClassVisitor` umožňuje vytvořit sekvenci návštěvníků. Jedním z těchto návštěvníků může být i instance třídy `ClassWriter`, která z parametrů volaných metod vytvoří opět binární reprezentaci bajtkódu. Tato třída může být použita i samostatně pro dynamické generování bajtkódu. Při průchodu souborem i při jeho vytváření je třeba pamatovat na pořadí, ve kterém jsou jednotlivé položky navštíveny. Programové rozhraní pro objektovou reprezentaci pomocí stromu je v balíčku `org.objectweb.asm.tree`. Obsah `class` souboru je reprezentovaný třídou `ClassNode`, která tvoří kořen stromu. Jednotlivé položky tvoří uzly. S takto vytvořeným stromem lze libovolně manipulovat i vytvořit strom zcela nový. Jednotlivé uzly jsou současně návštěvníky daných položek. Díky tomu je možné libovolně přecházet mezi oběma přístupy k bajtkódu. Balíčky `org.objectweb.asm.util`, `org.objectweb.asm.commons` a `org.objectweb.asm.tree.analysis` obsahují některé zajímavé nástroje pro zpracování a analýzu bajtkódu.

ASM zaujme svým návrhem a možností výběru mezi dvěma způsoby práce s bajtkódem. Nabízí vyšší úroveň abstrakce než BCEL, neboť přístup k tabulce konstant je uživateli zcela odepřen. Na druhou stranu je práce s bajtkódem stále na úrovni blízké formátu `class` souboru. Z popisovaných nástrojů je ASM považován za nejrychlejší.

3.3 Javassist

Javassist [4] nebo-li Java Programming Assistant je knihovna poskytovaná pod trojitou licencí MPL, LGPL a Apache License. Je vhodná zejména pro úpravu bajtkódu za běhu programu. Knihovna umožňuje pracovat s `class` soubory na dvou úrovních. Úroveň zdrojového kódu nevyžaduje znalost bajtkódu, ale umožňuje s bajtkódem manipulovat pomocí slovníku programovacího jazyka Java. Úroveň bajtkódu umožňuje přístup k reprezentaci blízké formátu `class` souboru. Java 8 je podporovaná.

V balíčku `javassist` je dostupné základní rozhraní knihovny. Třída `CtClass` je reprezentací `class` souboru. Instanci této třídy je třeba získat z úložiště reprezentovaného třídou `ClassPool`. V tomto úložišti jsou k dispozici všechny takto načtené třídy. Získanou reprezentaci třídy lze modifikovat a uložit do souboru či pole bajtů, nebo lze vytvořit reprezentovanou třídu. Těla metod lze modifikovat pomocí tříd z balíčku `javassist.expr`. Manipulace na úrovni zdrojového kódu má však jistá omezení a nejsou podporovány všechny jazykové konstrukce. Proto balíček `javassist.bytecode` poskytuje rozhraní pro přímou editaci bajtkódu. Instrukční soubor je zde reprezentovaný třídou `ClassFile`. K dispozici je i tabulka konstant reprezentovaná třídou `ConstPool`.

S `class` souborem se v Javassist opět manipuluje prostřednictvím objektové reprezentace. Zajímavá je však možnost pracovat s bajtkódem jako s konstrukcemi programovacího jazyka Java. Javassist tak nabízí mnohem vyšší úroveň abstrakce než BCEL a ASM. Navíc má propracovanější podporu editace bajtkódu za běhu.

Kapitola 4

Nástroj pro analýzu bajtkódu

K získání dostatečně obecných dat, ze kterých bych mohla čerpat při návrhu metod pro optimalizaci velikosti bajtkódu, bylo potřeba zpracovat a analyzovat velké množství `class` souborů. Bylo proto výhodné navrhnout a implementovat nástroj, který tyto činnosti umožňuje zautomatizovat. Výslednému nástroji `jbyca` nebo-li Java Bytecode Analyzer je věnovaná tato kapitola.

4.1 Požadavky na program

Při návrhu programu jsem vycházela z požadavků na výstupy programu. Ty by měly umět zodpovědět následující otázky:

1. Kolik souborů, bajtů, tříd, metod a členských proměnných se zpracovalo?
2. Jaké jsou velikosti jednotlivých položek v souborech?
3. Jaká je maximální hloubka zásobníku v metodách?
4. Kolik lokálních proměnných metody používají?
5. Jaké je využití lokálních proměnných?
6. Jaké jsou typické sekvence instrukcí v souborech?
7. Jak vypadá obsah konkrétního `class` souboru?

Program je určen ke zpracování `class` souborů, přičemž důraz je kladen na jejich dávkové zpracování. Z toho plynou požadavky na vstup programu a jeho rychlost. Vstupem může být `class` soubor, `jar` soubor nebo adresář. Adresáře a `jar` soubory jsou dále prohledávány a každý nalezený `class` soubor je považován za další vstup programu.

4.2 Návrh programu

Program jsem rozdělila na několik podprogramů, přičemž každý z nich řeší jeden z požadavků. Většinu potřebných dat lze získat jednoduše pomocí knihoven ASM a BCEL. Výjimku tvoří nalezení typických sekvencí instrukcí. Sekvence instrukcí je třeba nějakým způsobem uchovávat v paměti a umožnit jejich zobecňování.

4.2.1 Reprezentace sekvencí instrukcí

Získání typických sekvencí instrukcí vyžaduje uchovávat v paměti všechny nalezené různé sekvence ze všech zpracovaných metod s četnostmi jejich výskytu. Každou novou sekvenci

je pak třeba porovnat s ostatními, a buď upravit četnost shodné sekvence, nebo vložit novou sekvenci mezi ostatní. To představuje velkou časovou a paměťovou zátěž a nelze zaručit, že program skončí dřív, než dojde k nedostatku paměti. Z těchto důvodů bylo třeba vymyslet úspornou datovou strukturu reprezentující sekvence a způsob zotavení se z nedostatku paměti.

Jako vhodná datová struktura se pro sekvence instrukcí nabízí strom. Kořenem stromu by byl prázdný uzel, uzly jednotlivé instrukce a žádný uzel by nesměl mít dva bezprostřední následníky se stejnou instrukcí. Každá cesta z kořene do nějakého uzlu stromu by představovala jednu sekvenci a poslední uzel této cesty by pak obsahoval hodnotu četnosti výskytu sekvence. Tuto stromovou strukturu lze vytvořit z postfixů seznamů instrukcí metod. Po vložení takového postfixu jsou z kořene stromu dostupné všechny jeho prefixy. Prefixy všech postfixů pak tvoří množinu všech sufixů, což jsou všechny různé sekvence seznamu instrukcí. Stromová reprezentace umožňuje snadné porovnávání i přidávání sekvencí instrukcí a šetří pamětí, neboť sekvence sdílejí společné prefixy.

Nedostatku paměti je třeba předcházet v každém případě. Možným řešením je pravidelně kontrolovat, kolik procent z dostupné paměti se již využilo, a při překročení určité hodnoty zmenšit velikost vytvořeného stromu. Zmenšení stromu je možné dosáhnout jedním průchodem, při kterém se odstraní všechny hrany do uzlů s nižší hodnotou četnosti výskytu, než je stanovený práh. Tento práh se následně zvedne. V krajním případě bude strom po ukončení výpočtu obsahovat pouze svůj kořen, ale výpočet neskončí nedostatkem paměti.

4.2.2 Zobecnění instrukcí a jejich sekvencí

Každá instrukce je daná svým operačním kódem a parametry. Zkoumat typické sekvence instrukcí s konkrétními hodnotami parametrů může přinést zajímavé výsledky, ale nevede k nalezení obecných typických konstrukcí. Instrukce lze tak nahradit jejich zobecněnými protějšky. Zobecnění instrukce se skládá ze svou částí: zobecnění operačního kódu a zobecnění parametrů. Zobecnění operačního kódu lze dosáhnout odstraněním typové informace z názvu operace. Zobecnění parametrů může mít dvě úrovně. Na nejnižší úrovni je parametr zastoupený jen svým typem. Na vyšší úrovni je každé hodnotě parametru přiřazen typ a číselný identifikátor. Pokud se taková hodnota vyskytuje v jedné sekvenci vícekrát, přiřazený identifikátor se nemění. Lze tak obecně zkoumat práci s opakujícími se hodnotami parametrů.

V některých případech může být užitečné naopak rozšířit informace o instrukcích. Pracuje-li instrukce s lokální proměnnou a proměnná je jedním z parametrů metody, pak je vhodné nahradit označení proměnné klíčovým slovem `this`, nebo identifikátorem parametru metody. Díky tomu je možné pozorovat, jak se v metodě pracuje s jejími parametry a jak třída pracuje se svými metodami a členskými proměnnými. Seznam instrukcí je vhodné doplnit o návěští, která budou označovat místa skoků. Budou-li tato návěští součástí zkoumaných sekvencí, lze rozlišit jednotlivé základní bloky instrukcí.

Jako další možné rozšíření se nabízí práce s divokými kartami. V sekvencích instrukcí lze instrukce na libovolných pozicích nahradit zástupnými instrukcemi. Tyto instrukce se nazývají divoké karty a představují další formu zobecnění instrukcí. Každá divoká karta v sekvenci označuje pozici, na které se může vyskytovat jedna libovolná instrukce. Z pohledu stromové struktury jsou všechny divoké karty totožné. Sekvence s divokými kartami tak umožňují zkoumat i typické sekvence instrukcí, které nenásledují bezprostředně za sebou. K vytvoření všech sekvencí s divokými kartami pro danou sekvenci je třeba určit všechny

možné kombinace indexů instrukcí, které budou nahrazeny divokými kartami. Z praktických důvodů je vhodné zakázat náhradu instrukcí, které formují základní bloky.

4.3 Popis implementace

Program jsem implementovala v programovacím jazyce Java 8 s použitím knihoven ASM 5.0[1] a BCEL 6.0[2] a nástroje Gradle 2.7[3]. Vzhledem k tomu, že program `jbyca` některé třídy sdílí s programem `jbyco` z kapitoly 6, tak jsou třídy a balíčky obou programů umístěné v balíčku s názvem `jbyco`. V balíčku `jbyco.analysis` je deklarována třída `Application` s metodou `main`, která dle předaných parametrů spustí některý z nástrojů pro analýzu bajtkódu. Každý nástroj má definovanou vlastní metodu `main` a lze jej tedy spustit i samostatně.

4.3.1 Zpracování parametrů

Balíček `jbyco.lib` slouží jako knihovna užitečných tříd a funkcí. Jeho součástí jsou třídy `AbstractOption` a `AbstractOptions` sloužící k definování a zpracování argumentů příkazové řádky a třída `Utils` obsahující různé statické metody.

4.3.2 Práce se soubory

Balíček `jbyco.io` obsahuje třídy pro práci se soubory. Třída `CommonFile` reprezentuje soubor pomocí jeho absolutní i relativní cesty. Iterátor `CommonFilesIterator` rekurzivně prochází daný adresář a pro nalezené soubory vytváří a vrací instance třídy `CommonFile`. Rozšířením tohoto iterátoru je třída `ExtractedFilesIterator`, která prochází i `jar` soubory. Pomocí něj umožňuje třída `BytecodeCommonFiles` iterovat přes všechny nalezené `class` soubory. Soubory typu `jar` jsou pomocí třídy `JarExtractor` rozbaleny do dočasného adresáře a prohledávání pokračuje v tomto adresáři. Knihovna metod pro práci s dočasnými soubory je obsažena ve třídě `TemporaryFiles`. Třída `BytecodeFilesCounter` slouží v určení počtu `class` souborů v prohledávaném prostoru a nevyžaduje ke své práci dočasné soubory.

4.3.3 Analýza bajtkódu

Nástroje pro analýzu bajtkódu jsou umístěné v balíčku `jbyco.analysis`. Implementují rozhraní `Analyzer`, které deklaruje metody pro zpracování `class` souborů `processClassFile` a výpis získaných dat `writeResults`.

4.3.4 Výpis textové reprezentace bajtkódu

Převedení obsahu `class` souboru do textové reprezentace a její výpis na standardní výstup zajišťuje nástroj `BytecodePrinter` z balíčku `jbyco.analysis.content`. Tabulku konstant vypisuje prostřednictvím třídy `ConstantPoolWriter`.

[1] `http://asm.ow2.org/`
[2] `https://commons.apache.org/bcel/`
[3] `http://gradle.org/`

4.3.5 Sběr statistik

Nástroj pro získání souhrnných informací o zpracovaných `class` souborech je implementovaný ve třídě `StatisticsCollector` z balíčku `jbyco.analysis.statistics`. Nástroj prochází jednotlivé položky souboru a počty těchto položek aktualizuje v instanci třídy `StatisticsMap`. Po zpracování všech souborů se vypíše tabulka s typem položky, celkovým počtem výskytů tohoto typu a počtem výskytů přepočteným na jeden soubor.

4.3.6 Analýza velikosti

Přehled o velikostech jednotlivých položek v souborech poskytuje nástroj `SizeAnalyzer` z balíčku `jbyco.analysis.size`. Nástroj určuje velikosti zpracovávaných položek a informace o jejich velikostech udržuje v instanci třídy `SizeMap`. Výstupem je tabulka s typem položky, celkovou velikostí položek tohoto typu v souborech, průměrnou velikostí jedné položky tohoto typu a relativní celkovou velikostí vzhledem k celkové velikosti zpracovaných souborů.

4.3.7 Analýza lokálních proměnných

Data o využití lokálních proměnných jsou dostupná s nástrojem `VariablesAnalyzer` z balíčku `jbyco.analysis.variables`. Nástroj s pomocí třídy `VariablesMap` zaznamenává pro každou metodu počet formálních parametrů a lokálních proměnných a jejich použití v instrukcích metod. Výstupem tohoto nástroje jsou dvě tabulky: jedna pro formální parametry metody a jedna pro lokální proměnné. Každá z tabulek obsahuje index do tabulky lokálních proměnných, počet metod, které s danou proměnnou pracují, počet instrukcí pro načítání, ukládání a inkrementaci, součet všech instrukcí a průměrné hodnoty.

4.3.8 Analýza maxim

Ve třídě `MaxAnalyzer` z balíčku `jbyco.analysis.max` je implementovaný nástroj pro analýzu maximálního počtu lokálních proměnných a maximální hloubky zásobníku operandů v metodách. Nástroj zkoumá maxima v položkách *max_stack* a *max_variables* v atributech `Code`. Výstupem jsou tabulky s četnostmi těchto maxim ve zkoumaných metodách.

4.3.9 Analýza typických sekvencí instrukcí

Nástroj `PatternsAnalyzer` z balíčku `jbyco.analysis.patterns` umožňuje vyhledat typické sekvence instrukcí. Dle návrhu potřebuje ke své práci stromovou reprezentaci sekvencí a abstrahovanou reprezentaci instrukcí. K těmto účelům slouží dále uvedené balíčky. Výpis nalezených sekvencí zajišťuje třída `PatternsWriter`. Výstupem jsou délky sekvencí, jejich absolutní a relativní četnosti výskytu a řetězové reprezentace sekvencí. Výstup není nijak seřazený.

Hledání typických sekvencí probíhá ve třídě `PatternsAnalyzer` následovně. Každý `class` soubor je převeden na stromovou reprezentaci knihovny ASM. Seznam instrukcí každé metody je doplněn o návěští označující začátek a konec metody. Z těchto seznamů se postupně vygenerují všechny sekvence instrukcí dané délky. Ze sekvencí mohou být následně generované sekvence s daným počtem divokých karet. Každá vygenerovaná sekvence je pomocí třídy `Abstractor` převedena na sekvenci abstrahovaných instrukcí typu `AbstractInstruction`. Abstrahované instrukce jsou prostřednictvím třídy `Cache` nahrazeny instrukcemi typu `CachedInstruction`. Výsledná sekvence se předá instanci třídy

`TreeBuilder` a vloží do vytvářeného stromu. Po zpracování každého souboru se zkontroluje dostupná paměť a případně provede ořezání stromu. Nakonec se prostřednictvím třídy `PatternsWriter` vypíšou nalezené sekvence a jejich četnosti.

Reprezentace sekvencí instrukcí

V balíčku `jbyco.analysis.patterns.tree` jsou dostupné třídy pro stromovou reprezentaci sekvencí instrukcí. Uzel stromu je reprezentován třídou `Node` a nese položku a čítač četnosti výskytu této položky. Třída `Tree` reprezentuje strom. Budování stromu, vkládání sekvencí a ořezávání stromu zajišťuje třída `TreeBuilder`. Třída `TreeExporter` slouží pro výpis grafu ve formátu GML.

Reprezentace a abstrakce instrukcí

Balíček `jbyco.analysis.patterns.instructions` obsahuje rozhraní a třídy pro reprezentaci a abstrakci instrukcí. Abstrahování instrukcí umožňuje třída `Abstractor`. Třída implementuje rozhraní návštěvníka metody z knihovny ASM. Jednotlivé instrukce jsou abstrahovány a ukládány do seznamu instrukcí typu `AbstractInstruction`. Abstrakce instrukce spočívá v abstrahování operačního kódu, parametrů a návěští. Instance tříd pro abstrakci jednotlivých komponent jsou předány třídě `Abstractor` v konstruktoru. Z abstrahovaných komponent lze sestavit abstrahovanou instrukci typu `Instruction`. Součástí balíčku jsou i třídy `Cache` a `CachedInstruction`. Třída `Cache` spravuje mapu slabých referencí na abstraktní instrukce mapovaných na slabé reference na instrukce typu `CachedInstruction`. Pro každou abstrahovanou instrukci pak vrátí odpovídající instrukci z mapy. Pro různé objekty se shodnými instrukcemi tak `Cache` vrátí stejný objekt. Třídy slouží k úspoře paměti.

Reprezentace a abstrakce návěští

V balíčku `jbyco.analysis.patterns.labels` jsou rozhraní a třídy pro práci s návěštími. Abstrahované návěští je reprezentované rozhraním `AbstractLabel`. Abstrakci návěští popisuje rozhraní `AbstractLabelFactory`. Implementacemi těchto rozhraní jsou třídy `NumberedLabel`, `NamedLable` a `RelativeLabelFactory`. Třída `RelativeLabelFactory` každému návěští přiřazuje relativní identifikátor. Návěštím pro začátek a konec metody budou přiřazeny objekty typu `NamedLable` s identifikátory `begin` a `end`. Zbývajícím návěštím budou přiřazeny číselné identifikátory. Tyto identifikátory jsou relativní. Znamená to, že jsou jedinečné pouze v rámci aktuální sekvence.

Reprezentace a abstrakce operačních kódů

S operacemi se pracuje v balíčku `jbyco.analysis.patterns.operations`. Operace jsou definované pomocí výčtů implementujících rozhraní `AbstractOperation`. Každá operace reprezentuje skupinu operačních kódů. Výčet `GeneralOperation` obsahuje operace bez typových informací na rozdíl od výčtu `TypedOperation`. Výčty `GeneralHandleOperation` a `TypedHandleOperation` implementující rozhraní `AbstractHandleOperation` definují operace, které se mohou vyskytovat v položkách *constant_methodHandle*. Rozhraní pro abstrakci operačních kódů má název `AbstractOperationFactory`. Implementacemi tohoto rozhraní jsou třídy `GeneralOperationFactory` a `TypedOperationFactory`.

Reprezentace a abstrakce parametrů

Balíček `jbyco.analysis.patterns.parameters` umožňuje práci s parametry instrukcí. Každá třída reprezentující parametr implementuje rozhraní `AbstractParameter`. Těmito implementacemi jsou výčty `ParameterType` popisující typ parametru a `ParameterValue` sloužící k popisu hodnot `null` a `this`, třída `NumberedParameter` popisující parametr pomocí typu a relativního identifikátoru a třída `FullParameter` sestávající z typu parametru a pole hodnot. Abstrahování je deklarované v rozhraní `AbstractParameterFactory` a implementované v následujících třídách. Metody třídy `GeneralParameterFactory` vrací instance z výčtu `ParameterType`, metody třídy `NumberedParameterFactory` vrací instance třídy `NumberedParameter` a metody třídy `FullParameterFactory` vrací instance třídy `FullParameter`.

Generování sekvencí s divokými kartami

Třídy balíčku `jbyco.analysis.patterns.wildcards` slouží k vytváření sekvencí s divokými kartami. Divoká karta je reprezentovaná hodnotou `null`. Třída `CombinationIterator` je iterátorem přes všechny povolené kombinace indexů instrukcí v sekvenci, které budou nahrazené divokými kartami. Třída `WildSequenceGenerator` je iterátorem přes všechny sekvence s divokými kartami, které lze z dané sekvence vytvořit.

4.4 Překlad a spuštění

Nástroj `jbyca` je implementován v programovacím jazyce Java 8, proto je pro jeho překlad a spuštění vyžadovaná instalace Java JDK 8 a Java JRE 8. Zdrojové soubory jsou rozdělené do několika projektů. Soubory nástroje `jbyca` jsou součástí projektu `analysis`, zatímco knihovny pro práci se soubory a argumenty příkazové řádky jsou umístěné v projektu `common`. Projekt `examples` je užitečný pro generování testovacích `class` souborů. Překlad a instalaci těchto projektů zajišťuje skript `gradlew` vygenerovaný nástrojem Gradle. Nápovědu k tomuto skriptu lze vypsat příkazem `./gradlew tasks`.

Příkaz `./gradlew build` ve všech projektech do jejich adresářů `build` přeloží zdrojové soubory, stáhne potřebné knihovny a sestaví a zabalí výsledné distribuce. Po zadání příkazu `./gradlew installDist` se distribuce nainstalují do adresářů `build/install`. Výsledkem instalace projektu `analysis` je adresář `jbyca` se dvěma podadresáři `bin` a `lib` pro spustitelné soubory a knihovny. Nástroj lze spustit příkazem `./jbyca` z adresáře `bin`. Nápověda k nástroji se vypíše po uvedení přepínače `--help`. Dokumentaci k programovému rozhraní lze vygenerovat příkazem `./gradlew javadoc`. Dokumentace bude k dispozici v adresáři `build/docs`.

Součástí spustitelných souborů jsou dva nástroje pro vygenerování a zpracování výstupů programu. Příkaz `./jbyca-experiment data out` spustí sérii analýz souborů v adresáři `data` a získané výstupy uloží do adresáře `out`. Příkaz `jbyca-postprocessing out cls` zpracuje soubory s výstupy v adresáři `out` a vygeneruje sjednocené a seřazené výstupy do adresáře `cls`. Obě činnosti mohou být v závislosti na velikosti vstupních dat časově náročné.

Kapitola 5

Optimalizace velikosti bajtkódu

V této kapitole analyzuji obsah vybraného vzorku `class` souborů a na základě zjištěných poznatků navrhuji metody pro optimalizaci jejich velikosti.

5.1 Analýza bajtkódu

Pomocí nástroje `jbyca` jsem získala data reprezentující vybraný vzorek testovacích souborů a tato data následně zpracovala a vyhodnotila. Zkoumala jsem velikosti položek v `class` souborech, užití lokálních proměnných a parametrů metod a typické sekvence instrukcí. Testovací vzorek jsem vytvořila z `jar` souborů stažených z `http://mvnrepository.com`. Soubory jsem vybírala náhodně s ohledem na jejich velikost a četnost stahování. Zvolený vzorek se skládal z 95 souborů o celkové velikosti 102,4 MB a obsahoval 59 230 `class` souborů.

5.1.1 Velikost položek v souboru

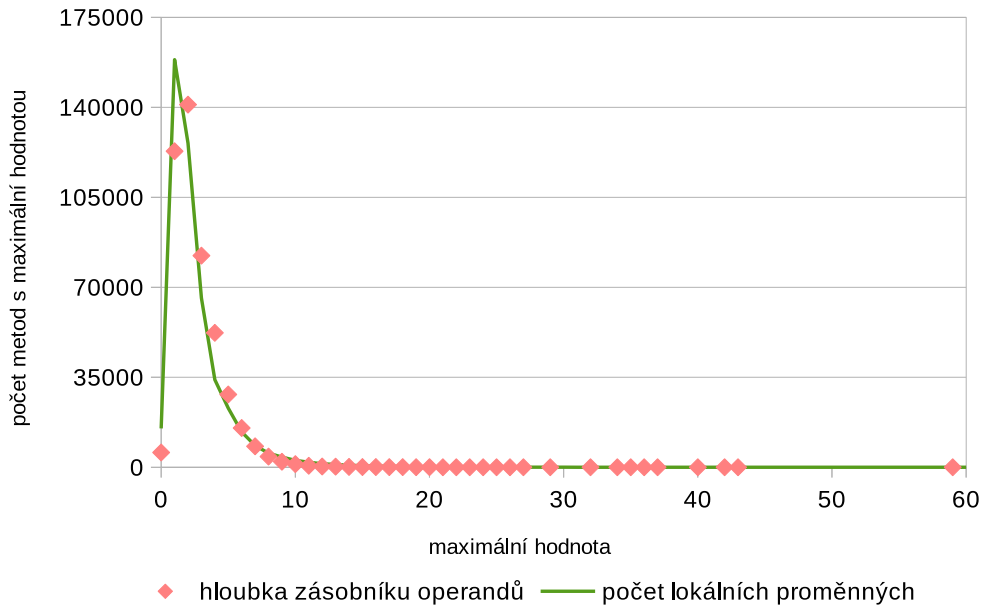
Typický `class` soubor z velkého vzorku obsahuje v průměru 117 konstant v tabulce konstant, 2 členské proměnné, 8 metod, 163 instrukcí a 29 atributů. Ze zkoumání celkové velikosti těchto položek vyplynulo, že konstanty tvoří 64% z celkové velikosti všech souborů, struktury pro členské proměnné 1%, struktury pro metody 2% a samotné instrukce 10%. Nejvýznamnějšími atributy jsou pak `Code` s velikostí 30%, informativní atributy s velikostí 14% a `StackMapTable` s velikostí 2%. Velikosti atributů vychází z délek atributů a velikosti konstant z tabulky konstant v nich nejsou zahrnuty.

Při bližším pohledu na velikosti konstant se ukázalo, že 89% z celkové velikosti konstant je tvořeno pouze konstantami typu *constant_utf8*. Tedy konstantami popisujícími řetězce v souboru. Z těchto řetězců pak 61% obsahuje názvy tříd, metod a členských proměnných a popisy jejich typů. Konstanty popisující číselné a řetězcové hodnoty tvoří 1% z celkové velikosti konstant a zbývající konstanty pro popisy tříd, metod a proměnných tvoří 10%.

Zkoumání instrukcí z hlediska jejich velikosti ukázalo, že prvních pět nejobjemnějších instrukcí tvoří 40% z celkové velikosti instrukcí. Jsou to instrukce pro volání metod, načtení hodnoty z členské proměnné a načtení reference na objekt z lokální proměnné s indexem 0. Na této pozici se často vyskytuje reference na aktuální objekt. Z instrukcí s proměnnou délkou má instrukce `tableswitch` v průměru velikost 107 B a instrukce `lookupswitch` velikost 43 B.

5.1.2 Počet lokálních proměnných a hloubka zásobníku

Průměrná maximální hloubka zásobníku operandů je 2,67. Průměrný maximální počet lokálních proměnných a parametrů metod je 2,71. Největší nalezená hloubka zásobníku pak byla 59 a největší nalezený počet lokálních proměnných 250. Data jsou znázorněna v grafu 5.1.



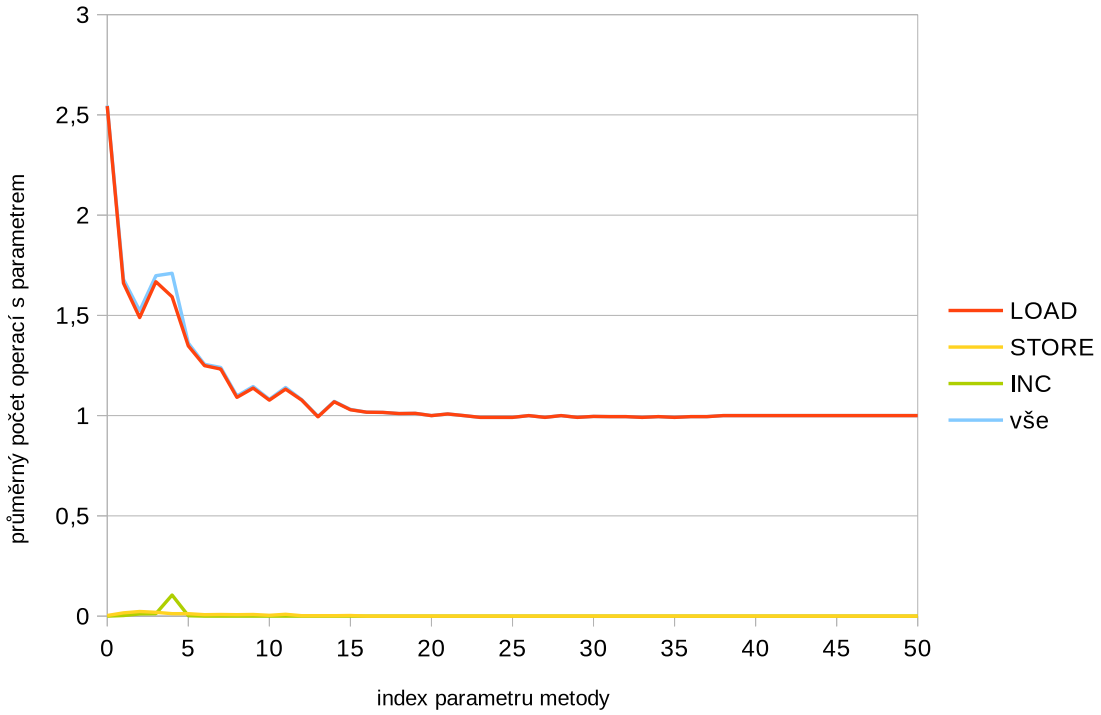
Obrázek 5.1: Maximální hloubky zásobníku operandů a maximální počty lokálních proměnných v metodách.

5.1.3 Užití lokálních proměnných a parametrů metody

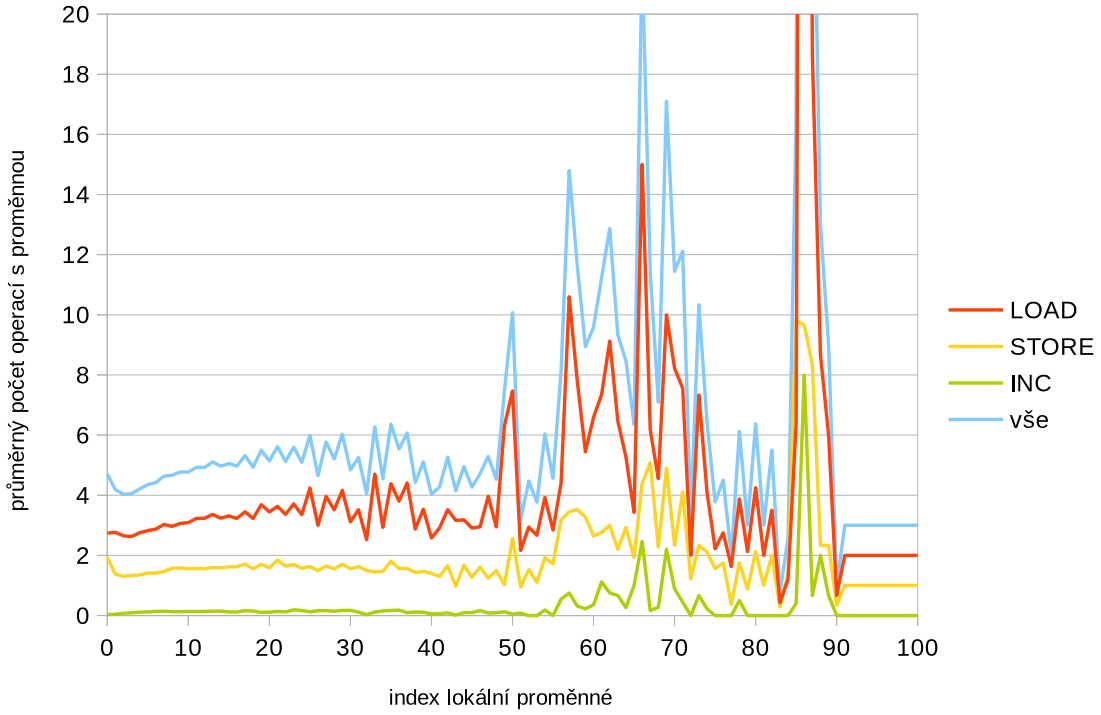
Data popisující užití parametrů a lokálních proměnných jsou znázorněna v grafech 5.2 a 5.3. Grafy znázorňují, kolikrát průměrná metoda pracuje se svým parametrem, respektive svojí lokální proměnnou na daném indexu. Z dat vyplývá, že z parametrů metody se jejich hodnoty načtou průměrně jedenkrát a dále se s nimi nepracuje. To lze z grafu vyčíst tak, že křivka pro instrukci `load` se téměř vždy překrývá s křivkou pro všechny typy instrukcí. Pro lokální proměnné platí, že s nižším indexem proměnné roste počet načítání jeho hodnoty až k počtu 2,5 pro index 0. V tomto indexu se u nestatických metod předává reference na aktuální objekt (`this`). S lokálními proměnnými, které neslouží k předávání parametrů, se průměrně provádí 4,2 operací načítání (`load`), 1,8 operací vkládání (`store`) a 0,3 operací inkrementace (`inc`).

5.1.4 Typické sekvence instrukcí

Procentuální ohodnocení sekvence instrukcí se vztahuje k počtu instrukcí a udává, kolik procent z celkového počtu instrukcí tvoří daná sekvence.



Obrázek 5.2: Průměrné počty operací s parametry metod.



Obrázek 5.3: Průměrné počty operací s lokálními proměnnými.

Typické operace

Z dat pro sekvence instrukcí délky jedna vyplývá, že 15,69% instrukcí tvoří instrukce pro volání metod objektu. Dalšími významnými instrukcemi jsou instrukce pro čtení hodnot z lokálních proměnných: čtení z parametru metody 10,34%, čtení z lokální proměnné 5,60% a čtení reference na aktuální objekt 9,00%. Na druhou stranu instrukcí pro ukládání hodnot do proměnných je výrazně méně: uložení do lokální proměnné 2,82%, uložení do parametru metody 1,39% a v 49 případech byla hodnota uložena do reference na `this`. To poukazuje opakované načítání neměnících se hodnot. Obdobně instrukce pro načtení a uložení hodnoty z členské proměnné tvoří 4,50% a 1,69%. Pro statické členské proměnné je to 1,15% a 0,29%.

Nejčastější instrukcí skoku je nepodmíněný skok 1,76%. Následují skoky s podmínkou rovnosti s nulou 1,21%, rovnosti s `null` 0,52% a nerovnosti s nulou 0,49%. Z hlediska práce se zásobníkem jsou zajímavé instrukce pro duplikaci a odebrání vrcholu, které tvoří 3,51% a 0,88%. Pro práci s polem pak instrukce pro uložení hodnoty 1,68%, načtení hodnoty 0,62% a zjištění délky pole 0,27%. Ukládání hodnot do pole je tedy častější operací než čtení hodnot z pole. U proměnných to bylo naopak. Instrukce pro návrat z metody tvoří 4,75% instrukcí a instrukce pro vytvoření nového objektu 1,96%.

Za zmínku dále stojí instrukce `swap` s 2 315 výskyty, `nop` s 364 výskyty a instrukce `lookupswitch` a `tableswitch`. Ve 21 případech instrukce `lookupswitch` obsahovala jen adresu výchozího bloku instrukcí, v 552 případech obsahovala jednu dvojici hodnota-adresa a v 998 případech dvě dvojice, což je nejčastější podoba této instrukce. V `tableswitch` se nejčastěji pracuje s rozsahem hodnot o délce tři a to ve 442 případech. Ve 150 případech byla délka rozsahu 1.

Typické parametry

Nejčastěji načítanými typy konstant jsou `int` 5,87%, řetězec 2,44%, reference na `null` 0,61%, `double` 0,24% a `long` 0,24%. Ze zkoumání konkrétních parametrů instrukcí pak vyplývá, že typickými konstantními hodnotami typu `int` jsou 0, 1, 2, 3, 4, -1, 8, 5, 6, 10, 7, 16 a 255. Pro typ `long` jsou to hodnoty 0, 1, -1, pro typ `double` 0, 1 a pro typ `float` 0, 1, 0,75 a 2. Typickou řetězcovou konstantou je prázdný řetězec.

Nejčastější volanou metodou je metoda `append` třídy `java.lang.StringBuilder` pro připojení řetězce. Následují metody `toString` a `<init>` téže třídy. Metoda `<init>` třídy `java.lang.Object` je až čtvrtá v pořadí. Dále se nejčastěji pracuje s metodami a objekty tříd `java.lang.StringBuffer`, `java.util.Iterator` a `java.lang.String`.

Typické je jednorozměrné pole typu `java.lang.Object`, `java.lang.String` nebo `byte`. Největší nalezená dimenze vícerozměrného pole byla 3. Typickými konstantními počty prvků v poli jsou hodnoty 0, 1, 2, 3. Nejčastěji se pracuje s nultým prvkem pole.

Typická těla metod

Celkem 39 317 ze 504 741 zkoumaných metod bylo abstraktních nebo byly metodami rozhraní. Dalších 5 687 metod obsahovalo pouze instrukci pro návrat z metody. Metody nejčastěji slouží k získání hodnoty z členské proměnné aktuálního objektu, volání metody aktuálního objektu a získání konstantní hodnoty. Ze zkoumání metod s konkrétními operačními kódy a parametry vyplývá, že 6 431 metod jsou implicitními konstruktory. Dalších 2 880 metod vrací hodnotu 0 typu `int` a 2 261 metod vrací hodnotu 1 typu `int`. Ve 1 474 případech pak metoda pouze vyvolá výjimku `UnsupportedOperationException`.

Výskyt	Sekvence	Popis sekvence
57 899	`astore` *var*; `aload` *var*;	Hodnota z vrcholu zásobníku se uloží do lokální proměnné a následně se načte zpět na zásobník.
26 694	`aload` *this*; `aload` *this*;	Za zásobník se dvakrát vkládá reference na aktuální objekt.
7 637	`pop`; `return`;	Manipuluje se se zásobníkem, ačkoliv na jeho stavu nezáleží.
4 023	`aload` *var*; `pop`;	Na zásobník se vloží hodnota a hned se odebere.
909	`aload` *var*; `aload` *this*; `swap`;	Instrukci `swap` lze aplikovat prohozením instrukcí.
527	`bipush` *i*; `bipush` *i*;	Za zásobník se dvakrát vkládá ta stejná hodnota.
222	`aload` *par*; `new` *type*; `dup_x1`; `swap`;	Sekvence typická pro balíček `groovy`.

Tabulka 5.1: Sekvence instrukcí manipulující se zásobníkem.

Zajímavé sekvence instrukcí

Z výstupů analýzy typických sekvencí instrukcí jsem vybrala některé zajímavé sekvence, ze kterých lze vycházet při návrhu metod pro optimalizaci velikosti bajtkódu. Společně s jejich popisem a četnostmi výskytu jsou uvedeny v několika následujících tabulkách.

Vybrané sekvence pro práci se zásobníkem z tabulky 5.1 poukazují na případy, kdy lze některou z instrukcí odstranit. Navíc se ukazuje, že se nedostatečně využívají specializované instrukce pro manipulaci se zásobníkem.

Z tabulky 5.2 vyplývá, že po vkládání číselných konstant se nemusí používat nejkratší varianty instrukcí. Dále je možné zjednodušit některé algebraické operace na základě jejich vlastností, ale žádné operace nad konstantními hodnotami v bajtkódu nalezeny nebyly. Stejně tak nebyly nalezeny žádné zjednodušitelné konverze hodnot.

Zajímavé sekvence instrukcí pracujících s objekty jsou uvedené v tabulce 5.3. Vzhledem k tomu, že se s objekty manipuluje prostřednictvím konkrétních členských proměnných a metod, tak se operace nad nimi špatně zobecňují.

Sekvence s instrukcemi pro podmíněné i nepodmíněné skoky jsou uvedené v tabulce 5.4. Sekvence ilustrují některé zbytečné operace, které se v bajtkódu vyskytovaly. Na druhou stranu bajtkód neobsahoval žádné snadno rozhodnutelné podmíněné skoky.

5.2 Metody pro optimalizaci velikosti bajtkódu

Výsledky analýzy `class` souborů jsem uplatnila při návrhu metod pro optimalizaci jejich velikosti. Inspirovala jsem se optimalizacemi, které navrhl Vašek [14] nebo které popisuje

Výskyt	Sekvence	Popis sekvence
1913	`ldc 0;`	Použití zbytečně veliké instrukce.
476	`iconst_0;` `iadd`	Celočíselné sčítání s nulou.
387	`l2i;` `i2b;`	Tuto sekvenci konverzí nelze nahradit jednou instrukcí. Sekvence, které by bylo možné nahradit, nalezeny nebyly.

291	`iload` *var*`;`
	`iconst_`*i*`;`
	`iadd;`
	`istore` *var*`;`
		Pro inkrementaci lokální proměnné existuje speciální instrukce `iinc`.

48	`iconst_0;` `ishl`	Posuv doleva o nula pozic.

Tabulka 5.2: Sekvence instrukcí s číselnými hodnotami a operacemi nad nimi.

Výskyt	Sekvence	Popis sekvence
1841	`checkcast` *type*`;` `checkcast` *type*`;`	Dvakrát se kontroluje, zda je ten stejný objekt toho stejného typu.
598	`putstatic` *class field*`;` `getstatic` *class field*`;`	Uložení hodnoty do statické členské proměnné dané třídy a opětovné získání této hodnoty.
282	`aconst_null;` `checkcast` *type*`;`	Kontrola, zda reference na `null` je daného typu.
0	`multianewarray 0` *type*`;`	Instrukce pro vytvoření nularozměrného pole objektů nebyla nalezena.

Tabulka 5.3: Sekvence instrukcí pracujících s objekty.

Výskyt	Sekvence	Popis sekvence
38 353	l_0: `goto` l_1;	Instrukce je počátkem bloku pro zpracování výjimky nebo cílovou instrukcí jiného skoku.
10 433	`goto` l_0; l_1: `iconst_`i; l_0: `ireturn;`	Nepodmíněný skok na instrukci pro návrat z metody.
2 273	`areturn;` `aconst_null;`	Před druhou instrukcí skoku chybí návěští. Tato instrukce se nemůže vykonat.
1 737	`ifne` l_0; `goto` l_1;	Při splnění i nesplnění podmínky dochází ke skoku.
841	`goto` l; l: ...	Nepodmíněný skok na následující instrukci.
39	`goto` l_0; `goto` l_1;	Před druhou instrukcí skoku chybí návěští. Tato instrukce se nemůže vykonat.
20	`if_cmpne` l; `goto` l;	Při splnění i nesplnění podmínky se skáče na stejnou adresu.

Tabulka 5.4: Sekvence instrukcí s podmíněnými a nepodmíněnými skoky.

Aho [1].

5.2.1 Optimalizace modifikující strukturu programu

Metody pro optimalizaci je vhodné rozdělit podle toho, zda optimalizace zasahují do struktury programu či nikoliv. Takový zásah představuje například přejmenování tříd, odstranění nepoužívaných metod či vkládání těla metody do těla jiné metody. Modifikace struktury programu nemusí být vždy žádoucí. Je třeba zvážit dva krajní případy. Pokud program slouží jako knihovna, ze které lze importovat balíčky a třídy, pak je třeba zachovat původní strukturu programu. Pokud je program konečnou aplikací zabalenou do `jar` souboru, pak je možné strukturu libovolně modifikovat tak, aby zůstala zachovaná sémantika.

Přejmenování balíčků, tříd, metod a členských proměnných

Více než polovinu z celkové velikosti souborů tvoří řetězce. Mezi tyto řetězce patří mimo jiné řetězcové konstanty, názvy atributů, tříd, metod, proměnných a popisy typů. Součástí popisu typu pak mohou být opět názvy tříd. Jako vhodná optimalizace se v této oblasti proto nabízí přejmenování balíčků, tříd, metod a členských proměnných za použití co nejkratších názvů. Tato forma optimalizace je obvykle vedlejším efektem obfuskace kódu [3].

Vkládání metod

Z analýzy typických těl metod vyplývá, že nezanedbatelná část metod má velmi primitivní funkcionalitu, jako je okamžitý návrat z metody či vrácení konstantní hodnoty. Volání takových metod je plýtvání časovými i paměťovými zdroji. Pokud je volání metody z hlediska velikosti dražší než provedení těla metody, pak je vhodné toto volání nahradit instrukcemi

z těla metody. Pokud se následně tato metoda nikde nevolá, je žádoucí její definici z příslušného `class` souboru zcela odstranit. Vkládání metod do kódu je často prováděno virtuálním strojem [11].

5.2.2 Optimalizace velikosti souboru

V této kapitole jsou uvedeny optimalizační metody, které nemodifikují strukturu programu a nejsou založené na nahrazování sekvencí instrukcí.

Odstranění zbytečných atributů

V kapitole 2.2.6 jsou mezi informativní atributy zařazeny ty, které poskytují informace vhodné například k ladění programů. Jsou to atributy `SourceFile`, `SourceDebugExtension`, `LineNumberTable`, `LocalVariableTable`, `LocalVariableTypeTable` a `Deprecated`. Z analýzy vyplynulo, že informativní atributy tvoří 14% z celkové velikosti zpracovaných `class` souborů. Vzhledem k tomu, že atributy nejsou důležité pro správnou interpretaci souboru, je žádoucí je ze souboru odstranit. Generování těchto atributů lze v překladači `javac` zabránit již za překladu volbou `-g:none`.

Realokace lokálních proměnných

Analýza užití lokálních proměnných a parametrů metod odhalila, že se s tímto paměťovým prostorem nepracuje optimálně. K úspornějšímu užívání proměnných nabádá i specifikace [9]. Proměnné lze realokovat a recyklovat tak, aby se častěji pracovalo s nižšími indexy proměnných. Nižší indexy pak umožňují používat kratší varianty instrukcí pro práci s proměnnými. Nejvíce užívanými indexy by proto měly být hodnoty 0 až 3, pro které existují specializované jednobajtové instrukce `load` a `store`.

5.2.3 Optimalizace sekvencí instrukcí

Z analýzy typických sekvencí instrukcí vyplynulo, že některé sekvence lze při zachování sémantiky nahradit kratšími sekvencemi. Tyto sekvence a jejich náhrady jsou popsané v této kapitole. Při nahrazování instrukcí je třeba dbát na zachování jejich vedlejších efektů. Pokud provedení instrukce může vyvolat výjimku, pak odstranění takové instrukce by mohlo změnit sémantiku programu. Totéž platí pro změnu pořadí instrukcí, které mohou vyvolat výjimku, neboť se tak změní i pořadí, ve kterém mohou být výjimky volány.

Optimalizace práce se zásobníkem

V některých sekvencích instrukcí lze instrukce pro práci se zásobníkem aplikovat na jiné instrukce. Například, je-li na zásobník vložena hodnota, která je v dalším kroku ze zásobníku zase odebrána, je možné instrukce pro vložení a odebrání hodnoty ze sekvence odstranit, pokud nemají vedlejší efekt. Výjimkou je instrukce pro duplikaci, která umožňuje zkrátit zápis sekvence instrukcí pro vložení více hodnot na zásobník. Je tedy výhodnější vyhledat dvojice instrukcí pro vložení shodných konstantních hodnot a jednu instrukci z dvojice nahradit instrukcí pro duplikaci. Konkrétní příklady jsou uvedené v tabulce 5.5.

Vzor	Náhrada	Popis optimalizace
`nop;`	-	Instrukce `nop` nemá žádný efekt. Lze ji proto odstranit.
`iconst_0;` `pop;`	-	Pokud instrukce před `pop` vkládá na zásobník hodnotu o délce jedné jednotky hloubky zásobníku a nemá žádný vedlejší efekt, lze obě instrukce odstranit. Obdobně pro `pop2`.
`iconst_0;` `pop2;`	`pop;`	Instrukci `pop2` lze aplikovat i částečně.
`bipush` x`;` `bipush` x`;`	`bipush` x`;` `dup;`	Pro duplikaci konstantních hodnot a hodnot z lokálních proměnných lze použít speciální instrukci `dup`.

`iload` x`;`
`iload` y`;`
`iload` x`;`
	`iload` y`;`
	`iload` x`;`
	`dup_x1;`
		Obdobně lze pro složitější duplikace používat instrukce `dup_x1`, `dup_x2`, `dup2`, `dup2_x1`, `dup2_x2`.

`dup;` `swap;`	`dup;`	Prohození dvou shodných hodnot nemá žádný efekt. Instrukci `swap` lze proto odstranit.

`iconst_0;`
`iconst_1;`
`swap;`
	`iconst_1;`
	`iconst_0;`
		Pokud obě instrukce před `swap` vkládají na zásobník hodnoty o délce jedné jednotky hloubky zásobníku a nemají žádný vedlejší efekt, lze je prohodit a instrukci `swap` odebrat.

`swap;` `return;`	`return;`	Pokud instrukce před `return` manipuluje se zásobníkem nebo lokálními proměnnými a nemá žádný vedlejší efekt, lze ji odstranit.

`iconst_0;`
`new` c`;`
`dup_x1;`
`swap;`
	`new` c`;`
	`dup;`
	`iconst_0;`
		Speciální případ aplikace instrukcí `swap` a `dup_x1`.

Tabulka 5.5: Příklady optimalizace práce se zásobníkem.

Vzor	Náhrada	Popis optimalizace
`iload` x; `iload` x;	`iload` x; `dup`;	Dvojí načtení hodnoty z téže lokální proměnné lze nahradit duplikací načtené hodnoty.
`iload` x; `istore` x;	-	Z lokální proměnné se načte hodnota a ihned se uloží do téže lokální proměnné. Taková operace nemá žádný efekt.
`istore` x; `iload` x;	`dup`; `istore` x;	Hodnota se uloží do lokální proměnné a ihned se z ní načte. Kratší alternativou je hodnotu na zásobníku duplikovat a kopii uložit do proměnné.
`istore` x; `istore` x;	`dup`; `istore` x;	Do téže lokální proměnné se dvakrát ukládá nějaká hodnota. První uložená hodnota je ihned přepsána druhou hodnotou, proto je zbytečné ji do proměnné vůbec ukládat.

Tabulka 5.6: Příklady optimalizace práce s lokálními proměnnými.

Optimalizace práce s lokálními proměnnými

Sekvence instrukcí, ve kterých se manipuluje jen s jednou lokální proměnnou, lze v některých případech nahradit kratšími sekvencemi. Ukázky těchto náhrad jsou uvedené v tabulce 5.6.

Optimalizace práce s objekty

Práci s objekty nelze bez hlubší analýzy příliš optimalizovat. Reference na objekt může být uchovávána ve více proměnných, s objekty může současně manipulovat více vláken programu a instrukce pro práci s objekty mohou generovat výjimky. Navržené metody popsané v tabulce 5.7 jsou proto pouze ty nejjednodušší.

Optimalizace konverze hodnot

V některých případech lze optimalizovat instrukce pro konverzi hodnot. Například, konverzi číselné konstanty lze nahradit konvertovanou číselnou konstantou a některé posloupnosti konverzí lze zjednodušit. Konkrétní příklady jsou uvedené v tabulce 5.8.

Zjednodušení algebraických výrazů

Některé sekvence instrukcí popisující výpočet algebraických výrazů lze zjednodušit nebo zcela nahradit hodnotou daného výrazu. Tato zjednodušení vychází z vlastností matematických operací a specifikace instrukcí. Jedná se například o celočíselné sčítání s nulou, rozdíl dvou shodných čísel či matematická operace se speciální hodnotou `NaN`. Konkrétní příklady jsou uvedené v tabulce 5.9.

Vzor	Náhrada	Popis optimalizace
`aconst_null;` `checkcast` *type*`;`	`aconst_null;`	Kontrola, zda reference na `null` je daného typu, nemá dle specifikace instrukce `checkcast` žádný efekt. Instrukci pro kontrolu typu lze proto odstranit.
`aconst_null;` `instanceof` *type*`;`	`iconst_0;`	Výsledkem určení, zda reference na `null` je daného typu, je dle specifikace instrukce `instanceof` hodnota nula typu `int`. Instrukce lze proto nahradit tímto výsledkem.
`checkcast` *type*`;` `checkcast` *type*`;`	`checkcast` *type*`;`	Pokud první instrukce vyvolá výjimku, pak se druhá instrukce neprovede, a pokud první instrukce nevyvolá výjimku, pak kontrola proběhla v pořádku a musí proběhnout v pořádku i ve druhé instrukci, neboť instrukce pracují se stejným typem i referencí. Druhá instrukce tedy nemá žádný efekt a lze ji odstranit.

`new` *exception*`;`
`dup;`
`invokespecial;`
`athrow;`
	`aconst_null;`
	`athrow;`
		Dle specifikace instrukce `athrow`, vyvolání výjimky `NullPointerException` z balíčku `java.lang` vytvořené bezparametrickým konstruktorem lze simulovat vyvoláním reference na `null`.

Tabulka 5.7: Příklady optimalizace práce s objekty.

Vzor	Náhrada	Popis optimalizace
`ldc 3.25;` `d2i`	`iconst_3;`	Dvojici instrukcí pro konverzi číselné konstanty lze nahradit instrukcí pro vložení konvertované číselné konstanty.
`i2l;` `l2i;`	`-`	Pokud sekvence konverzí převede hodnotu daného typu zpět na hodnotu téhož typu bez ztráty informace, lze takovou sekvenci odstranit. Obdobně pro sekvenci `f2d; d2f`.
`i2b;` `i2b;`	`i2b;`	Výsledkem konverze na typ `byte` je hodnota typu `int` v rozsahu typu `byte`. Opakovaná konverze je proto zbytečná.

Tabulka 5.8: Příklady optimalizace konverze hodnot.

Vzor	Náhrada	Popis optimalizace
`iconst_0;` `iadd;`	-	Přičtení nuly nemá žádný efekt. Instrukce lze proto smazat. Obdobně pro `ishr`, `ishl`, `iushr` a instrukce pro typ `long`.
`ldc NaN;` `fadd;`	`pop;` `ldc NaN;`	Výsledkem součtu s hodnotou `NaN` je `NaN`. Obdobně pro typ `double`.

`ldc` x`;`
`ldc` y`;`
`iadd;`
	`ldc` $x+y$`;`	Jsou-li operandy matematické operace číselné konstanty, pak lze určit výsledek operace. Instrukce je pak možné nahradit tímto výsledkem.

`iinc` i `0;`	-	Inkrementace lokální proměnné i o nulu nemá žádný efekt.

`iload` x`;`
`bipush` i`;`
`iadd;`
`istore` x`;`
	`iinc` x i`;`	Užití specializované instrukce `iinc`. Obdobně lze nahradit operaci `isub`.

Tabulka 5.9: Příklady zjednodušení algebraických výrazů.

Optimalizace konkatenace řetězců

Konkatenace řetězců je realizována pomocí instancí třídy `java.lang.StringBuilder`. Vzhledem k tomu, že její metoda `append` je dle výsledků analýzy bajtkódu jednou z nejčastěji volaných metod, je žádoucí počet volání této metody nějak redukovat. Příklady těchto redukcí jsou následující:

1. Volání metody `append` pro prázdný řetězec je zbytečná operace. Lze proto odstranit jak instrukci pro vložení prázdného řetězce na zásobník, tak instrukci pro volání metody.

2. Opakované volání metody `append` pro řetězcové konstanty lze nahradit jedním voláním metody pro konkatenaci těchto konstant. Taková sekvence tedy bude nahrazena jednou instrukcí `ldc`, která na zásobník vloží konkatenaci řetězcových konstant, a jednou instrukcí pro volání metody `append`.

3. Je-li objekt třídy `java.lang.StringBuilder` vytvořen bezparametrickým konstruktorem a parametrem prvního volání metody `append` je řetězec, pak lze tuto sekvenci nahradit voláním parametrického konstruktoru, jehož parametrem je parametr metody `append`.

4. Je-li objekt třídy `java.lang.StringBuilder` vytvořen parametrickým konstruktorem, kde parametrem je řetězcová konstanta, a parametrem prvního volání metody `append` je řetězcová konstanta, pak lze sekvenci nahradit voláním parametrického konstruktoru, kde parametrem je konkatenace řetězcových konstant.

Vzor	Náhrada	Popis optimalizace
`sipush 0;`	`iconst_0;`	Instrukce pro vkládání číselných proměnných lze někdy nahradit kratšími instrukcemi.
`ldc2_w 3.0;`	`iconst_3;` `i2d;`	Drahé instrukce pro vkládání číselným konstant lze nahradit kratšími instrukcemi pro jiný typ a přetypováním.

`ldc2_w NaN;`	`dconst_0;`
	`dup2;`
	`ddiv;`
		Instrukci pro vložení konstanty `NaN` lze nahradit instrukcemi pro dělení nulou s plovoucí řádovou čárkou. Ušetří se tak jedna položka v tabulce konstant.

Tabulka 5.10: Příklady substitucí číselných konstant.

Substituce číselných konstant

Číselné konstanty lze vkládat na zásobník více způsoby. Pro některé hodnoty existují krátké specializované instrukce, zatímco jiné hodnoty je nutné vyjádřit pomocí položky v tabulce konstant. Je tedy žádoucí každou instrukci pro vložení číselné konstanty nahradit co nejkratší alternativou. Například tříbajtovou instrukci `sipush 0` lze nahradit jednobajtovou instrukcí `iconst_0`.

Jednotlivé číselné datové typy jsou v instrukční sadě různě podporované. Například pro hodnotu 3 typu `int` existuje specializovaná jednobajtová instrukce `iconst_3`, ale pro hodnotu 3,0 typu `double` je třeba použít tříbajtovou instrukci `ldc2_w` a devítibajtovou položku v tabulce symbolů. Někdy proto může být vhodnější použít kratší instrukci s jiným typem a výsledek přetypovat, jak je ukázáno v tabulce 5.10.

Některé z uvedených substitucí nelze aplikovat před metodou zjednodušení algebraických výrazů a metodou optimalizace konverze hodnot, neboť by se zápis instrukcí opět zjednodušil. Současně by tyto substituce mohly bránit nalezení některých jiných vzorů. Je proto vhodné provést metodu substituce číselných konstant až jako poslední krok optimalizace.

Optimalizace podmíněných a nepodmíněných skoků

V rámci optimalizací podmíněných a nepodmíněných skoků je možné zjednodušit řízení toku programu. Toho lze dosáhnout eliminací zbytečných skoků a detekcí a odstraněním nedosažitelného kódu. Za nedosažitelný kód lze považovat instrukce, které se při běhu programu nemohou nikdy provést.

Příkladem zbytečného skoku je skok na bezprostředně následující instrukci nebo podmíněný skok, který při splnění i nesplnění podmínky skáče na stejnou adresu. Příkladem nedosažitelného kódu je pak instrukce, která bezprostředně následuje za instrukcí nepodmíněného skoku nebo instrukcí pro návrat z metody a není cílovou instrukcí nějakého skoku.

Někdy je také možné nahradit instrukce `tableswitch` a `lookupswitch` za ekvivalentní sekvence podmíněných skoků. Tato náhrada však nemá smysl v případech, kdy jsou porovnávané hodnoty příliš vysoké nebo skoky na instrukce příliš dlouhé. Další optimalizační metody jsou uvedené v tabulce 5.11.

Vzor	Náhrada	Popis optimalizace
`goto` l; l: ...	l: ...	Nepodmíněný skok na bezprostředně následující instrukci lze odstranit. Obdobně pro podmíněné skoky.
`goto` l_0; `iconst_0` l_1: ...	`goto` l_0; l_1: ...	Pokud instrukce bezprostředně následující za `goto` není cílem skoku z nějaké instrukce, případně počátkem bloku pro zpracování výjimky, pak je nedosažitelná a lze ji odstranit. Totéž platí pro instrukce pro návrat z metody.
`goto` l_0; ... l_0: `goto` l_1;	`goto` l_1; ... l_0: `goto` l_1;	Skok na instrukci nepodmíněného skoku lze nahradit přímým skokem na cílovou instrukci. Pokud po této úpravě bude instrukce pro meziskok nedosažitelná, lze ji odstranit. Metodu lze aplikovat i na podmíněné skoky, pokud výsledná relativní adresa nebude příliš velká.
`goto` l; ... l:`return`;	`return`; ... l:`return`;	Nepodmíněný skok na instrukci pro návrat z metody lze nahradit kratší instrukcí pro návrat z metody.
`iconst_1`; `ifgt` l;	`goto` l;	Pokud je možné vyhodnotit výsledek porovnání, lze dle výsledku podmíněný skok odstranit, nebo nahradit nepodmíněným skokem.
`dup`; `if_icmpeq` l;	`pop`; `goto` l;	Duplikovaná číselná hodnota je rovna své kopii. Výsledkem porovnání na rovnost je proto pravda. Lze aplikovat na všechny instrukce pro rovnost a nerovnost.
`ifeq` l; `goto` l;	`pop`; `goto` l;	Nezáleží na výsledku porovnání, neboť se v obou případech skáče na shodnou adresu. Podmíněný skok lze proto odstranit.
`iconst_0`; `if_icmpge` l;	`ifge` l;	Instrukci pro porovnání hodnoty typu `int` s nulou lze nahradit kratší variantou instrukce.
`lookupswitch` l;	`pop`; `goto` l;	Pokud je tabulka dvojic v instrukci `lookupswitch` prázdná, pak lze tuto instrukci nahradit skokem na výchozí adresu.

Tabulka 5.11: Příklady optimalizace skoků.

Kapitola 6

Nástroj pro optimalizaci velikosti bajtkódu

V této kapitole popisuji návrh a implementaci nástroje `jbyco` nebo-li Java Bytecode Optimizer určeného k optimalizaci velikosti `class` souborů. Nástroj implementuje optimalizační metody, které jsem navrhla v předchozí kapitole. Efektivita těchto metod je na základě výstupů programu vyhodnocena v kapitole 6.5.

6.1 Požadavky na program

Cílem programu je implementovat navržené optimalizační metody a na testovacím vzorku `class` souborů demonstrovat jejich efektivitu. Vstupem programu může být `class` soubor, `jar` soubor nebo adresář. Výstupem pak bude optimalizovaný `class soubor`, nebo kopie souborů s optimalizovanými `class` soubory. Každý `class` soubor se bude zpracovávat nezávisle na dalších souborech a informacích. Dále je potřeba umožnit vypisovat informace o provedených optimalizacích.

6.2 Návrh programu

Kapitola 5.2 je ve velké míře věnovaná optimalizačním metodám založeným na náhradě sekvencí instrukcí. Rozhodla jsem se proto na tyto metody zaměřit i při implementaci. Metody lze rozdělit na tzv. peephole optimalizace a optimalizace řízení toku programu. Peephole optimalizacím se dále věnuji v kapitole 6.2.1, zatímco optimalizacím řízení toku programu v kapitole 6.2.2.

K implementaci jsem se rozhodla použít knihovnu ASM. Avšak s ohledem na to, jakým způsobem je v knihovně bajtkód reprezentován, je výhodné hned na počátku optimalizací odstranit informativní atributy. Tyto atributy by jinak byly součástí seznamů instrukcí, což by zkomplikovalo vykonávání dalších optimalizací.

Dále jsem se rozhodla ve zjednodušené formě aplikovat realokaci lokálních proměnných. Toto zjednodušení spočívá v přečíslení indexů lokálních proměnných tak, aby byly proměnné očíslovány v pořadí prvního užití v instrukcích. Tímto přeskupením může vzniknout kompaktnější pole lokálních proměnných. Takový nástroj navíc poskytuje přímo knihovna ASM. Nad celkovým pořadím žádaných optimalizací se zamýšlím v kapitole 6.2.3.

6.2.1 Peephole optimalizace

Většina navržených optimalizačních metod v kapitole 5.2 je založených na nalezení vzorové sekvence instrukcí a náhradě této sekvence za optimalizované řešení. Tento způsob optimalizace se dle McKeemana [10] nazývá peephole optimalizace a je specifický tím, že se vždy zkoumá jen malý úsek kódu. Nezbytnou součástí návrhu programu `jbyco` je tedy i návrh rozhraní pro peephole optimalizace. Konkrétně je třeba navrhnout, jakým způsobem budou definovány vzorové sekvence, jak budou vzorové sekvence rozpoznávány v kódu určeném k optimalizaci a jak bude probíhat samotná úprava kódu.

Způsobů implementace peephole optimalizace je mnoho [2]. Jako přímé řešení se nabízí převádět vstupní bajtkód na řetězcovou reprezentaci. Vzorové sekvence by pak byly popsány regulárními výrazy a problematika nalezení a úpravy bajtkódu by se zjednodušila na prosté hledání a nahrazování regulárních výrazů v řetězci. Na závěr by bylo nutné řetězcovou reprezentaci opět převést na sekvenci instrukcí. Výhodou tohoto přístupu je snadné ladění, neboť řetězcová reprezentace kódu je snadno čitelná.

Efektivnější variantou by mohlo být hledání regulárními výrazy přímo nad instrukcemi bajtkódu. Regulární výrazy pro popis vzorových sekvencí by bylo možné předzpracovat a vygenerovat konečný stavový automat, který bude tyto výrazy rozpoznávat. Nalezené sekvence instrukcí pak mohou být nahrazeny za optimální. Pokud bude automat generován v době překladu, pak by takové řešení mělo být velice rychlé. Vyžaduje však implementovat nástroje pro lexikální a syntaktickou analýzu regulárních výrazů, nástroje pro převod regulárních výrazů na konečný automat a nástroje pro minimalizaci konečného automatu. Pokud se však sleví z požadavků na sílu jazyka pro popis vzorových sekvencí, lze tuto variantu zjednodušit.

Většinu vzorových sekvencí v navržených optimalizačních metodách tvoří posloupnost instrukcí konstantní délky. Vzorové instrukce jsou specifikovány svými operačními kódy a omezeními kladenými na jejich parametry. Jako možné řešení se proto nabízí definovat vzorové sekvence pomocí posloupností operačních kódů a teprve při nalezení odpovídající sekvence kontrolovat další omezení. Pokud jsou všechna omezení splněna, může být sekvence optimalizována. Ve své implementaci jsem zvolila tuto variantu.

Každá peephole optimalizace je tedy definována posloupností symbolů, které reprezentují množiny operačních kódů, a akcí, které provádí dodatečné kontroly a optimalizaci. Jedním z výstupů akce je informace o tom, zda se optimalizace provedla či neprovedla. Rozpoznávání posloupností symbolů lze realizovat jednoduchým konečným automatem, jehož stavy jsou indexy do posloupnosti symbolů. Počátečním stavem je nula a koncovým stavem je délka posloupnosti. Pokud operační kód instrukce na vstupu patří do množiny reprezentované symbolem na indexu určeném aktuálním stavem, automat provede přechod na následující index. Pokud automat přejde do koncového stavu, provede se pro nalezenou sekvenci instrukcí odpovídající akce.

6.2.2 Optimalizace řízení toku programu

Některé z navržených metod pro optimalizaci podmíněných a nepodmíněných skoků nelze implementovat pomocí peephole optimalizací, neboť vyžadují dodatečné informace o návěštích a adresách instrukcí. Některé z těchto optimalizací navíc mohou tyto informace zneplatnit, a proto je třeba je při každé možné změně aktualizovat. Adresy instrukcí stačí určit za použití heuristiky, kdy se adresy počítají pomocí největších možných velikostí instrukcí. Získané adresy tak popisují nejhorší možný případ.

6.2.3 Pořadí optimalizačních metod

U některých optimalizačních metod může záležet na pořadí, v jakém se aplikují. Například metodu pro substituci číselných konstant má smysl uskutečnit až jako poslední krok optimalizace, neboť jiné metody by mohly tuto substituci zvrátit. Na druhou stranu metoda pro odstranění zbytečných atributů může usnadnit další manipulaci s kódem a je proto vhodné ji aplikovat při načtení vstupního bajtkódu.

U optimalizací sekvencí instrukcí může docházet k tomu, že některé optimalizace budou vytvářet příležitosti pro další optimalizace. Vzhledem k uspořádání zásobníkového kódu pak může být výhodnější aplikovat metody pro sekvence instrukcí na kód v opačném směru, tedy od poslední instrukce k první. Nejprve se tak zkoumají operace a jejich parametry, předtím než se začnou zkoumat další vztahy mezi instrukcemi. Tento způsob práce s bajtkódem lze podpořit expanzí duplikací. Pokud je možné instrukci pro duplikaci aplikovat na úrovni instrukcí kódu, může tato aplikace vytvořit nové příležitosti pro optimalizace.

Na pořadí metod má vliv i způsob, jakým budou optimalizační metody implementované. Knihovna ASM umožňuje kombinovat dva přístupy pro práci z bajtkódem. Aby se zabránilo opakovanému generování stromové reprezentace, je vhodné optimalizaci rozdělit na tři fáze. V první fázi je vstupní pole bajtů převedeno na stromovou reprezentaci. Během tohoto převodu lze pomocí zřetězení návštěvníků již aplikovat některé optimalizace. Ve druhé fázi se optimalizuje stromová reprezentace. A ve třetí fázi je stromová reprezentace skrze zřetězené návštěvníky opět převedena na pole bajtů. V tomto okamžiku lze rozhodnout, zda optimalizace měly na vstupní kód nějaký efekt. Pokud došlo k redukci velikosti bajtkódu, lze všechny fáze zopakovat. Pokud k redukci nedošlo, proces optimalizace se ukončí. Validitu výsledného bajtkódu lze na závěr ověřit nástrojem z knihovny ASM. Konečný návrh na pořadí optimalizačních metod je následující:

1. Odstranění zbytečných atributů.
2. Expanze duplikací.
3. Zjednodušení kódu.
4. Redukce duplikací.
5. Realokace lokálních proměnných.
6. Substituce číselných konstant.

6.3 Popis implementace

Program jsem implementovala v jazyce Java 8 za použití knihovny ASM 5.0[1] a tříd, které vznikly v rámci implementace nástroje `jbyca`. Překlad a instalaci zajišťuje knihovna Gradle 2.7[2]. Program nepodporuje vytváření `jar` souborů z optimalizovaných souborů a kopírování souborů, které nejsou typu `class`, do výstupní struktury.

V balíčku `jbyco.optimization` je umístěná třída `Application` s hlavní metodou `main`. Tato metoda zpracuje parametry a vstupní soubor předá iterátoru `BytecodeFilesIterator` z balíčku `jbyco.io`. Pro každý nalezený `class` soubor se zavolá metoda `optimizeClassFile` třídy `Optimizer`. Ve třídě `Optimizer` jsou implementované všechny fáze optimalizace batjkódu. Ke sběru informací o provedených optimalizacích slouží třída `Statistics`. Třídy implementující optimalizační metody pro expanzi kódu jsou umístěné v podbalíčku `expansion`, třídy pro zjednodušení kódu jsou v podbalíčku `simplification` a třídy pro redukci kódu

[1] `http://asm.ow2.org/`
[2] `http://gradle.org/`

jsou obsažené v podbalíčku `reduction`. Různé obecné nástroje a třídy jsou umístěné v podbalíčku `common`.

Reprezentace vzorových sekvencí a akcí

Balíček `jbyco.optimization.peephole` obsahuje třídy pro implementaci a aplikaci peephole optimalizací. Rozhraní `Symbol` pro symboly reprezentuje množinu operačních kódů. Má jedinou metodu `match` s parametrem `AbstractInsnNode` z knihovny ASM, která vrací `true`, pokud operační kód dané instrukce patří do množiny reprezentované symbolem, jinak `false`. Toto rozhraní je implementováno výčtem `Symbols`, kde každý operační kód je reprezentován jedním symbolem a další symboly reprezentují skupiny operačních kódů se stejnými vlastnostmi. Vzorovou sekvenci lze definovat pomocí anotace `Pattern`, která obsahuje pole symbolů typu `Symbol`. Akce je reprezentovaná třídou implementující funkční rozhraní `PeepholeAction` a anotovanou anotací `Pattern`. Každá akce definuje metodu `replace` s parametry seznam instrukcí typu `InsnList` z knihovny ASM a pole instrukcí typu `AbstractInsnNode`. Seznam instrukcí je aktuální reprezentací těla zpracovávané metody. Pole instrukcí pak obsahuje instrukce ze seznamu instrukcí a tvoří nalezenou sekvenci instrukcí. Pomocí prvků tohoto pole lze snadno provádět dodatečné kontroly a modifikovat seznam instrukcí. Návratová hodnota je typu `boolean` a značí, zda došlo k modifikaci seznamu instrukcí. Třída `InsnUtils` z balíčku `jbyco.optimization.common` slouží akcím jako knihovna užitečných funkcí.

Hledání vzorových sekvencí a aplikace akcí

Třída `PeepholeRunner` z balíčku `jbyco.optimization.peephole` zapouzdřuje hledání vzorů a spouštění odpovídajících akcí. Její metoda `loadActions` načte akce a vzory a vytvoří pro ně konečné automaty reprezentované instancemi třídy `StateMachine`. Třída `StateMachine` obsahuje metody pro testování, zda lze přečíst vstupní instrukci, pro čtení vstupní instrukce, pro určení, zda je automat v konečném stavu, a pro vykonání příslušné akce.

Metoda `findAndReplace` třídy `PeepholeRunner` slouží k hledání vzorů a spouštění akcí. Dokud v daném seznamu instrukcí dochází ke změnám, tak se v něm opakovaně vyhledává. Pro aktuální instrukci se nejprve ověří, zda nemůže být přečtena některým z načtených konečných automatů. Pokud ano, je vytvořena kopie tohoto automatu a vložena do fronty běžících automatů. Tato fronta je následně dále zpracovávána. Pokud automat z fronty běžících automatů přečte aktuální instrukci, pak ve frontě zůstává, a pokud ji nepřečte, je z fronty odebrán. Přejde-li automat do koncového stavu, spustí se akce, a modifikuje-li akce seznam instrukcí, fronta běžících automatů se vyprázdní a vyhledávání vzorů se restartuje.

Aplikace optimalizací řízení toku programu

Třídy pro implementaci a aplikaci optimalizací řízení toku programu jsou umístěné v balíčku `jbyco.optimization.jump`. Obdobně jako u peephole optimalizací jsou optimalizační akce definované pomocí funkčních rozhraní `LabelAction`, `FrameAction`, `TableSwitchAction` a `LookupSwitchAction`. Tyto akce lze načíst a aplikovat pomocí tříd `LabelTransformer`, `FrameTransformer`, `TableSwitchTransformer` a `LookupSwitchTransformer` rozšiřujících abstraktní třídu `MethodTransformer` z balíčku `jbyco.optimization.common`. Bajtkód je optimalizován pomocí zřetězení těchto tříd a třídy `JumpTransformer`. Tato třída rozšiřuje abstraktní třídu `ClassTransformer` z balíčku `jbyco.optimization.common`. Data nezbytná pro provádění optimalizací se sbírají ve třídě `JumpCollector`. Informace o návěštích

Počet náhrad	Optimalizační metoda
352 972	Optimalizace práce se zásobníkem
167 242	Optimalizace podmíněných a nepodmíněných skoků
123 173	Expanze duplikace
34 934	Redukce duplikace
32 528	Optimalizace konverze hodnot
32 357	Optimalizace konkatenace řetězců
2 788	Optimalizace práce s objekty
1 881	Zjednodušení algebraických výrazů

Obrázek 6.1: Četnosti optimalizačních metod.

se uchovávají v instancích třídy `LabelNodeInfo` a informace o rámcích zásobníkových map ve `FrameNodeInfo`.

6.4 Překlad a spuštění

Nástroj je implementován v programovacích jazyce Java 8. K překladu a spuštění proto vyžaduje instalaci Java JDK 8 a Java JRE 8. Zdrojové soubory nástroje `jbyco` jsou součástí projektu `optimization`. Projekt lze přeložit a nainstalovat pomocí skriptu `gradlew` nástroje Gradle stejným způsobem, jako bylo popsáno v kapitole 4.4. Nástroj lze spustit příkazem `./jbyco`. Nápovědu vypíše volba `--help`.

6.5 Zhodnocení výstupů programu

Pomocí programu `jbyco` jsem optimalizovala vzorek dat, který jsem analyzovala v kapitole 5.1. Jak je ukázáno v tabulce 6.1, nejčastěji se prováděly optimalizační metody založené na náhradě sekvencí instrukcí pro práci se zásobníkem. Naopak nejméně se využívaly metody pro zjednodušování algebraických výrazů. V tabulce chybí optimalizační metody, které se prováděly vždy a jejich aplikaci tedy nelze nijak kvantifikovat.

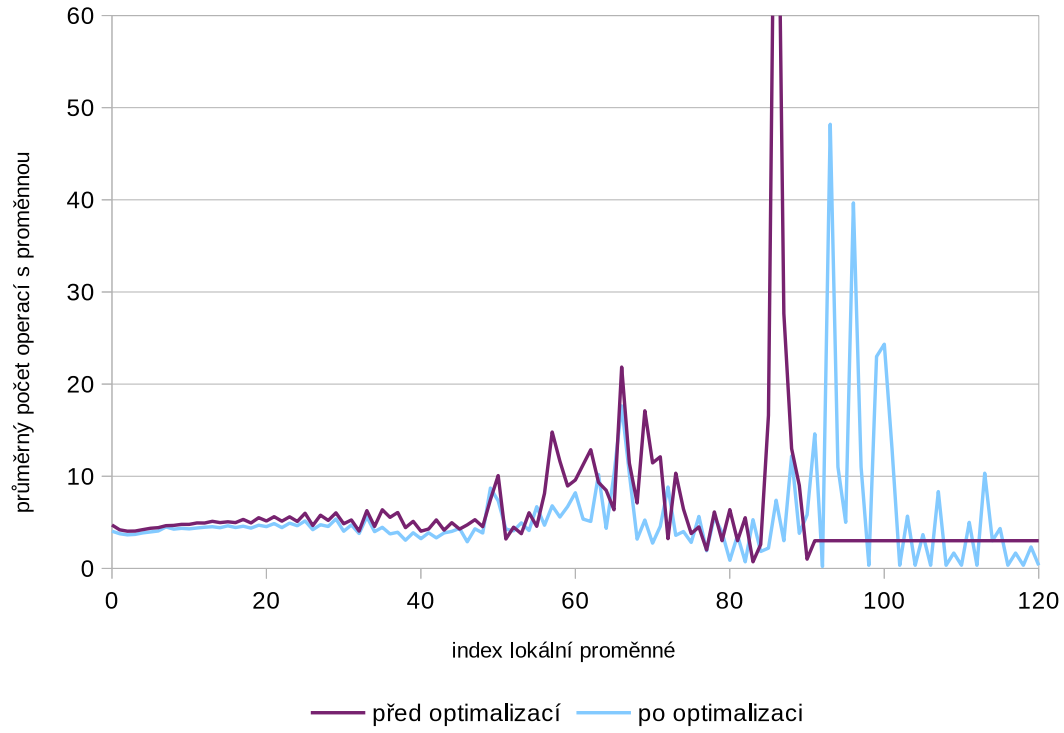
Vliv optimalizačních metod na celkovou velikost `class` souborů znázorňuje tabulka 6.2. Je patrné, že nejefektivnější metodou bylo odstranění zbytečných atributů. Aplikace této metody přinesla úsporu 22,487%. Z metod pro optimalizaci instrukcí byla nejefektivnější optimalizace skoků s 1,741%. Zajímavé je, že navzdory předpokladům má expanze duplikací negativní vliv na výslednou velikost. Při aplikaci optimalizací bez expanzí byla výsledná velikost souborů menší než při aplikaci optimalizací s expanzí. Na základě tohoto zjištění tedy lze fázi expanze z optimalizačního procesu odstranit.

Optimalizované `class` soubory jsem znovu analyzovala nástrojem `jbyca`. Optimalizace nijak výrazně neovlivnila maximální hloubky zásobníků operandů ani maximální počty lokálních proměnných. Na využití parametrů metody se optimalizace též nijak zásadně neprojevila. Zaznamenatelná změna však nastala u využití lokálních proměnných. Graf 6.3 zobrazuje, kolik operací se v metodách průměrně provedlo s lokální proměnnou na daném indexu před a po optimalizaci. Obecně lze říci, že počet operací s proměnnými klesl, neboť proměnné na indexech větších než 80 využívá méně než 10 metod. Zvýšení počtu operací u některých proměnných s indexy v rozsahu 80 až 120 lze vysvětlit nahrazením instrukce

Optimalizační metoda	Úspora v B	Úspora v %
Realokace lokálních proměnných	22 963	0,013
Substituce číselných konstant	141 781	0,075
Nahrazení sekvencí instrukcí	395 646	0,208
Optimalizace skoků	3 313 415	1,741
Odstranění zbytečných atributů	42 815 740	22,487
Všechny optimalizace	46 886 770	24,626
Všechny optimalizace bez expanze duplikací	46 888 858	24,627

Obrázek 6.2: Vliv optimalizačních metod na celkovou velikost `class` souborů.

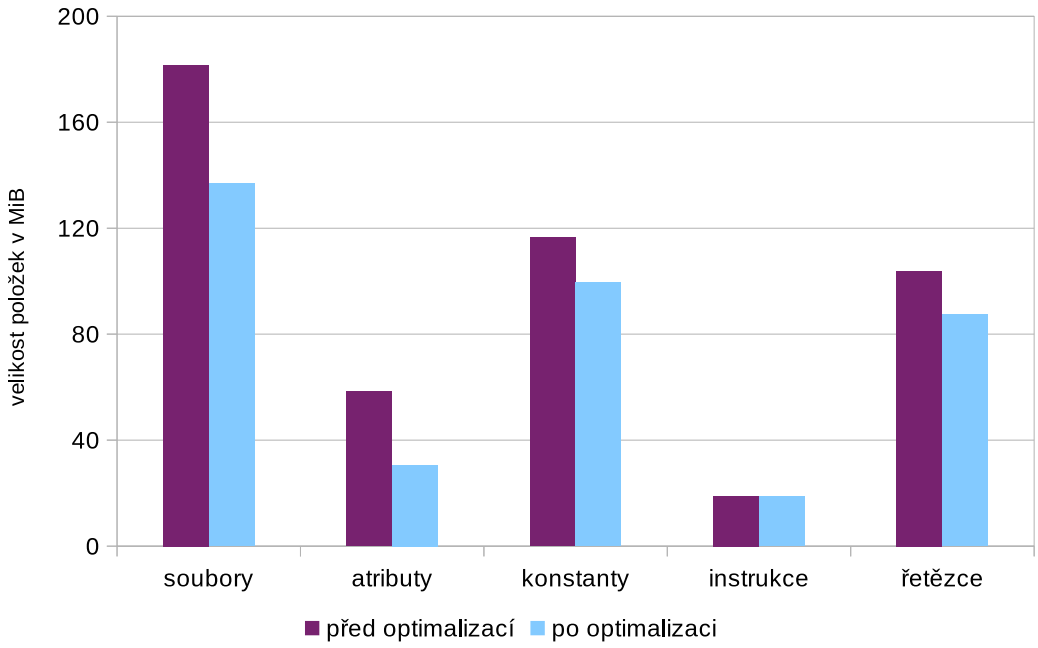
`tableswitch` nebo `lookupswitch` za sekvenci podmíněných skoků, kde se porovnávaná hodnota opakovaně načítá z lokální proměnné.



Obrázek 6.3: Využití lokálních proměnných před a po optimalizaci.

Na grafu 6.4 je znázorněna změna celkových velikostí položek ve zpracovaných `class` souborech. Graf potvrzuje, že optimalizace měla největší vliv na velikost atributů. Celkem se ušetřilo 44,7 MiB ze 181,6 MiB.

Dosažené výsledky nejsou zanedbatelné, ale rozšířením optimalizačních metod by se mohlo dosáhnout ještě větší úspory paměti. Výhodou nástroje `jbyco` je však zachování původních struktur programů a snadná optimalizace souborů bez složité konfigurace. Jiné nástroje pro optimalizaci velikosti [8] navíc ke své činnosti vyžadují vyřešené všechny závislosti mezi soubory, což nástroj `jbyco` nevyžaduje.



Obrázek 6.4: Velikosti položek `class` souborů před a po optimalizaci.

Kapitola 7

Závěr

Na základě specifikace [9] jsem popsala virtuální stroj Java Virtual Machine a formát jeho instrukčního souboru. Dále jsem se seznámila s nástroji pro manipulaci s bajtkódem BCEL, ASM a Javassist, uvedla jejich stručný popis a vzájemně je porovnala. Tyto knihovny jsem následně použila pro implementaci nástroje `jbyca` pro analýzu bajtkódu. Nástroj mi umožnil analyzovat velký vzorek `class` souborů a získat data vhodná k návrhu metod pro optimalizaci velikosti souborů. Konkrétně jsem se zabývala počty a velikostmi položek v souborech, využitím paměťového prostoru pro lokální proměnné a analýzou typických sekvencí instrukcí. Na základě poznatků z této analýzy jsem navrhla metody pro optimalizace velikosti bajtkódu. Většina navržených optimalizací je založena na náhradě sekvence instrukcí za kratší sekvenci. Další slouží k odstranění neužitečných informací ze souborů, nebo vedou k modifikaci struktury programu. Některé z navržených optimalizačních metod jsem následně implementovala v nástroji `jbyco` pro optimalizaci velikosti bajtkódu. Při návrhu nástroje jsem kladla důraz na snadnou modifikovatelnost a rozšiřovatelnost implementovaných optimalizací. Nakonec jsem s využitím tohoto nástroje optimalizovala testovací vzorek dat a výstupy opět analyzovala. Velikost dat se snížila o zhruba 25% se zachováním původních struktur programů.

Jako možné pokračování této práce se nabízí optimalizace nástroje pro hledání a nahrazování sekvencí instrukcí, jak je popsáno v kapitole 6.2. Dále by bylo vhodné implementovat a analyzovat ostatní navržené metody včetně těch, které modifikují strukturu programu. Je také možné rozšířit oblast aplikace optimalizačních metod z lokální úrovně na úroveň interprocedurální a intraprocedurální. Optimalizace na takové úrovni však již vyžadují vhodnější reprezentaci instrukcí a pokročilé analýzy kódu.

Literatura

[1] Aho, A. V.; Lam, M. S.; Sethi, R.; aj.: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, druhé vydání, 2007, ISBN 0-321-49169-6, 1009 s.

[2] Chakraborty, P.: Fifty years of peephole optimization. *Current Science*, ročník 108, č. 12, červen 2015: s. 2186–2190, ISSN 0011-3891.

[3] Chan, J.-T.; Yang, W.: Advanced Obfuscation Techniques for Java Bytecode. *J. Syst. Softw.*, ročník 71, č. 1-2, duben 2004: s. 1–10, ISSN 0164-1212.

[4] Chiba, S.: Javassist [online]. 2015 [cit. 2016-01-11].
Dostupné z: `http://jboss-javassist.github.io/javassist/`

[5] Clausen, L. R.; Schultz, U. P.; Consel, C.; aj.: Java Bytecode Compression for Low-end Embedded Systems. *ACM Trans. Program. Lang. Syst.*, ročník 22, č. 3, květen 2000: s. 471–489, ISSN 0164-0925.

[6] Engel, J.: *Programming for the Java Virtual Machine*. Addison-Wesley, 1999, ISBN 9780201309720, 488 s.

[7] Kazi, I. H.; Chen, H. H.; Stanley, B.; aj.: Techniques for Obtaining High Performance in Java Programs. *ACM Comput. Surv.*, ročník 32, č. 3, září 2000: s. 213–240, ISSN 0360-0300.

[8] Lafortune, E.: ProGuard [online]. 2015 [cit. 2016-05-16].
Dostupné z: `http://proguard.sourceforge.net/`

[9] Lindholm, T.; Yellin, F.; Bracha, G.; aj.: *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley, 2014, ISBN 013390590X, 600 s.

[10] McKeeman, W. M.: Peephole Optimization. *Commun. ACM*, ročník 8, č. 7, červenec 1965: s. 443–444, ISSN 0001-0782.

[11] Oracle Corporation: The Java HotSpot Performance Engine Architecture [online]. 2015 [cit. 2016-05-16].
Dostupné z: `http://oracle.com/technetwork/java/whitepaper-135217.html`

[12] OW2 Consortium: ASM [online]. 2015 [cit. 2016-01-11].
Dostupné z: `http://asm.ow2.org/`

[13] The Apache Software Foundation: BCEL [online]. 2014 [cit. 2016-01-11].
Dostupné z: `http://commons.apache.org/proper/commons-bcel/`

[14] Vašek, M.: *Optimalizace bajtkódu Javy s ohledem na jeho velikost*. Bakalářská práce, Masarykova univerzita, Fakulta informatiky, Brno, 2013.
Dostupné z: `http://theses.cz/id/yoo69v/`

[15] Venners, B.: *Inside the Java Virtual Machine*. McGraw-Hill Companies, druhé vydání, 2000, ISBN 0-07-135093-4, 703 s.

Přílohy

Seznam příloh

Příloha A

Obsah CD

Přiložené CD obsahuje následující soubory:

`xponco00.pdf`	Písemná zpráva ve formátu PDF.
`dp-xponco00.tar.gz`	Zdrojové texty písemné zprávy.
`app-xponco00.tar.gz`	Zdrojové soubory aplikací s manuálem.
`data-xponco00.tar.gz`	Analyzovaný vzorek dat.
`application`	Spustitelné soubory aplikací.