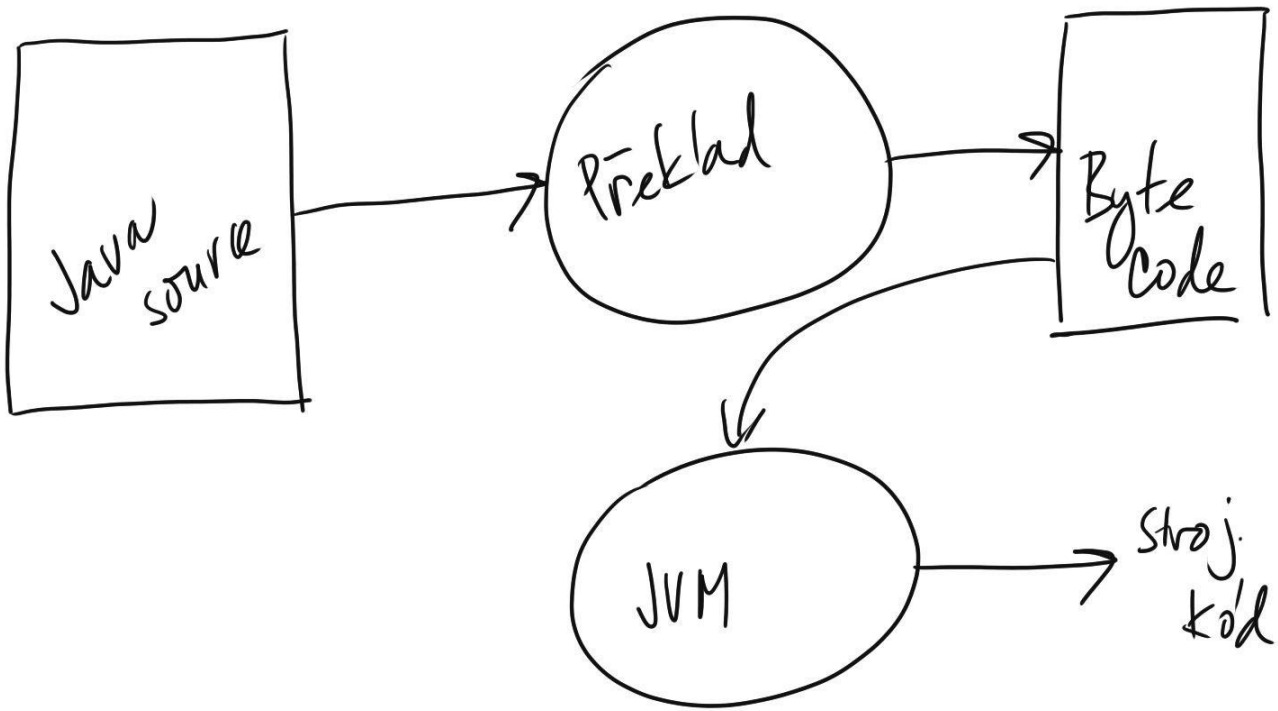
Java, překlad JIT a souvislosti

Překladače - přednášky, 2024 LS

Lenka Kosková Třísková, NTI TUL


Java source
Překlad
Byte code
JVM
Stroj. kód

Co je byte code?

- Opravdu blízké assemblerům a strojovým kódům
- Seznam instrukcí: https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings
- Ukázka konverze kódu:

  https://en.wikipedia.org/wiki/Java_bytecode

Co označujeme jako jazyky pro JVM?

- Jazyky, pro něž existuje převod do bytecode a jsou tedy interpretovatelné pomocí JVM
- A je jich FAKT hodně ;)
- https://en.wikipedia.org/wiki/List_of_JVM_languages
- Výstup překladu obsahuje tabulku symbolů a bytecode
- A i pro tyto jazyky lze využít výhody JIT

Co je JIT?

Just **I**n **T**ime compilation = překlad za běhu

JIT optimization = **Optimalizace za chodu**

V čem je výhodnější?

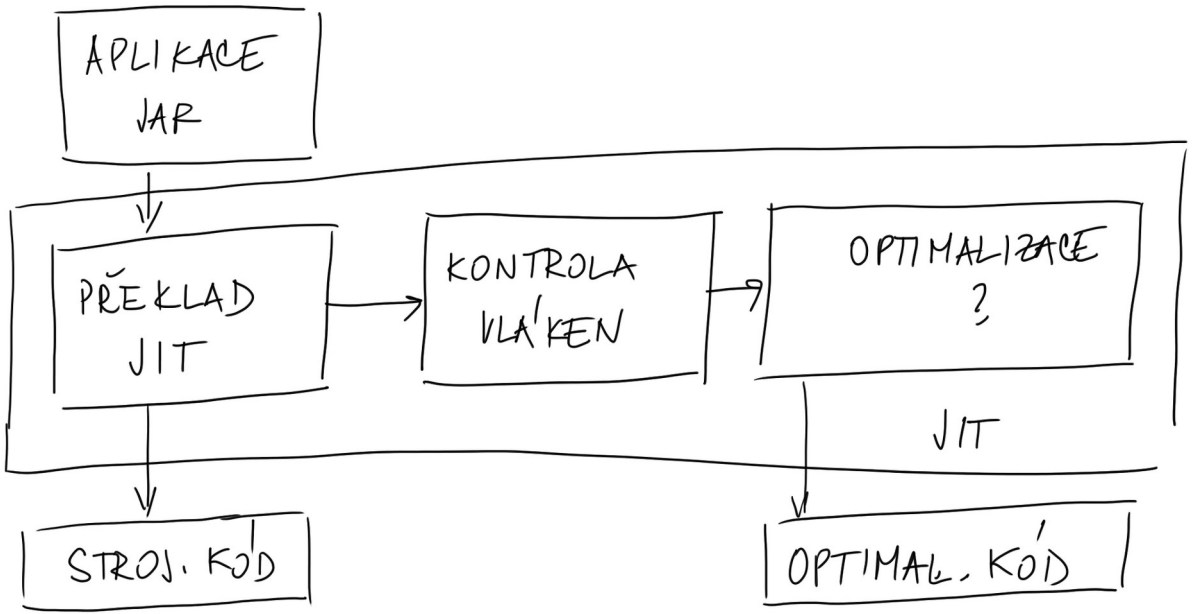
- Známe hodnodnoty proměnných
- Známe vstupní data
- Víme, jak často se volají určité úseky kódu

=> Můžeme ještě lépe optimalizovat!

Základní principy

- Můžeme ještě více provozovat “inlining”:
  - Známe počty opakování i u jiných cyklů, než for
  - Známe velikosti vektorů a matic
  - Zjistíme-li, že je krátká metoda volána často - inlning
- Můžeme kód pozorovat:
  - Je-li metoda volána více než cca 1000x => do strojového kódu
  - Tráví program v metodě hodně času? => “On stag replacement” => do strojového kódu
  - Aktivní vlákna => do strojového kódu

Java - JVM


APLIKACE
JAR
PŘEKLAD
JIT
KONTROLA
VLÁKEN
OPTIMALIZACE
?
JIT
STROJ. KÓD
OPTIMAL. KÓD

Ukázka: Zbavíme se zbytečného

```
class A {
B b;
public void foo() {
y = b.get();
něco dělej dál, ale neměň b dle
kontextu runtime;
z = b.get();
sum = y + z;
}
}
```

```
class A {
B b;
public void foo() {
y = b.value;
něco dělej dál, ale neměň b dle
kontextu runtime;
sum = y + y;
}
}
```

Vy to pište pořádně, ať je kód přehledný!
Optmializaci řeší překladač ;)

Ukázka: Eliminace skoků v runtime

```
private static int isOpt(int x, int y) {
int veryHardCalculation = 0;

if (x >= y) {
veryHardCalculation = x * 1000 + y;
}
else {
veryHardCalculation = y * 1000 + x;
}
return veryHardCalculation;
}
```

```
private static int isOpt(int x, int y) {
int veryHardCalculation = 0;

if (x < y) {
// Za chodu už vím, že x je většinou nebo skoro
vždy větší
veryHardCalculation = y * 1000 + x;
return veryHardCalculation;
}
else {
veryHardCalculation = x * 1000 + y;
return veryHardCalculation;
}
```

Ukázka: Expanze smyček

```
private static double[] loopUnrolling(double[][]
matrix1, double[] vector1) {
double[] result = new double[vector1.length];

for (int i = 0; i < matrix1.length; i++) {
for (int j = 0; j < vector1.length; j++) {
result[i] += matrix1[i][j] * vector1[j];
}
}

return result;
}
```

```
private static double[] loopUnrolling2(double[][]
matrix1, double[] vector1) {
double[] result = new double[vector1.length];

for (int i = 0; i < matrix1.length; i++) {
result[i] += matrix1[i][0] * vector1[0];
result[i] += matrix1[i][1] * vector1[1];
result[i] += matrix1[i][2] * vector1[2];
//Tohle tu mám místo vnořeného cyklu, už vím, že j
je malé (a nebo není, ale paměti je dost a počítá
se to často)
.
.
}

return result;
}
```

Klasika vs JIT

Klasický překlad a optimalizace	**JIT**
Může inlinovat, zná-li počty	Může skoro vždy inlinovat (zná počty)
Reflexe skoro není možná (neznáme počty průchodů a vazby)	Reflexe možná je
Není možná spekulativní optimalizace (neznáme data)	Spekulativní optimalizace možná
Celkový výkon (velká data) typicky nižší	Celkový výkon (velká data) vyšší
Plná rychlost od startu	V začátku pomalejší (probíhá měření a statistiky)
Žádná další zátěž pro CPU v runtime	Optimalizátor CPU samozřejmě dál zatíží

Reflexe: https://www.cs.ubc.ca/~murphy/jRMTool/doc/rma.htm

JIT - kroky

Fáze 1: Inlining

Fáze 2: Lokální optimalizace: Lokální data-flow, optimalizace pro registry

Fáze 3: Kontrola toku kódu:

- Pořadí kroků vzhledem ke statistice
- Otáčení smyček a podmínek
- Úpravy smyček
- Kontrola switch - velký switch nahradit rychlou volbou a potom switch pro méně časté

Fáze 4: Optimalizace globálně

Fáze 5: Převod do nativního kódu (“assembler”)

Vlastnosti JIT

Volba pro JVM (https://www.ibm.com/docs/en/sdk-java-technology/8?topic=options-xjit)

Lze vypnout, určit počet volání před překladem do assembleru, lze vynechat funkce z optimalizace apod.

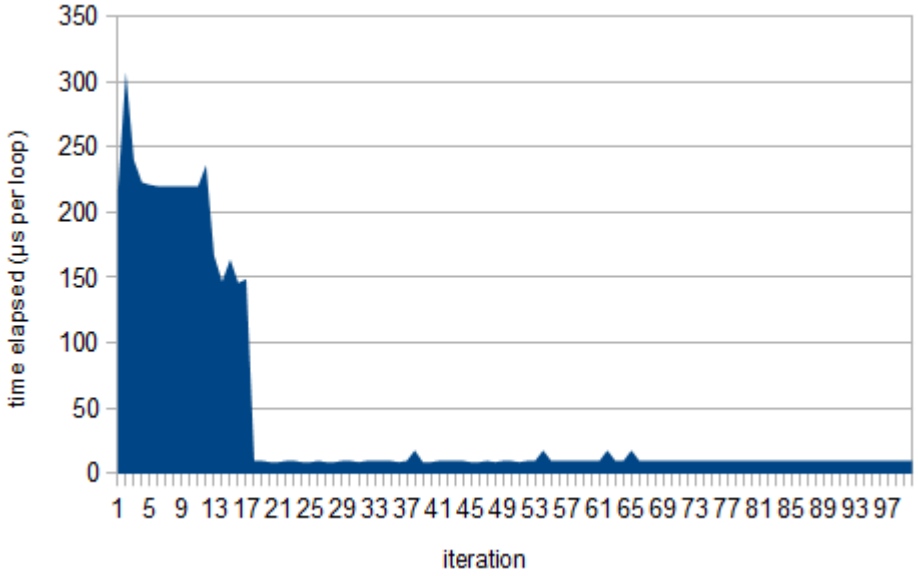
-XX:MaxInlineSize

-XX:FreqInlineSize

Měření výkonu

Jasně, jasně, jevilo by se, že optimalizátor sežere čas, který ušetří…

...jenže:



https://www.beyondjava.net/blog/a-close-look-at-javas-jit-dont-waste-your-time-on-local-optimizations/

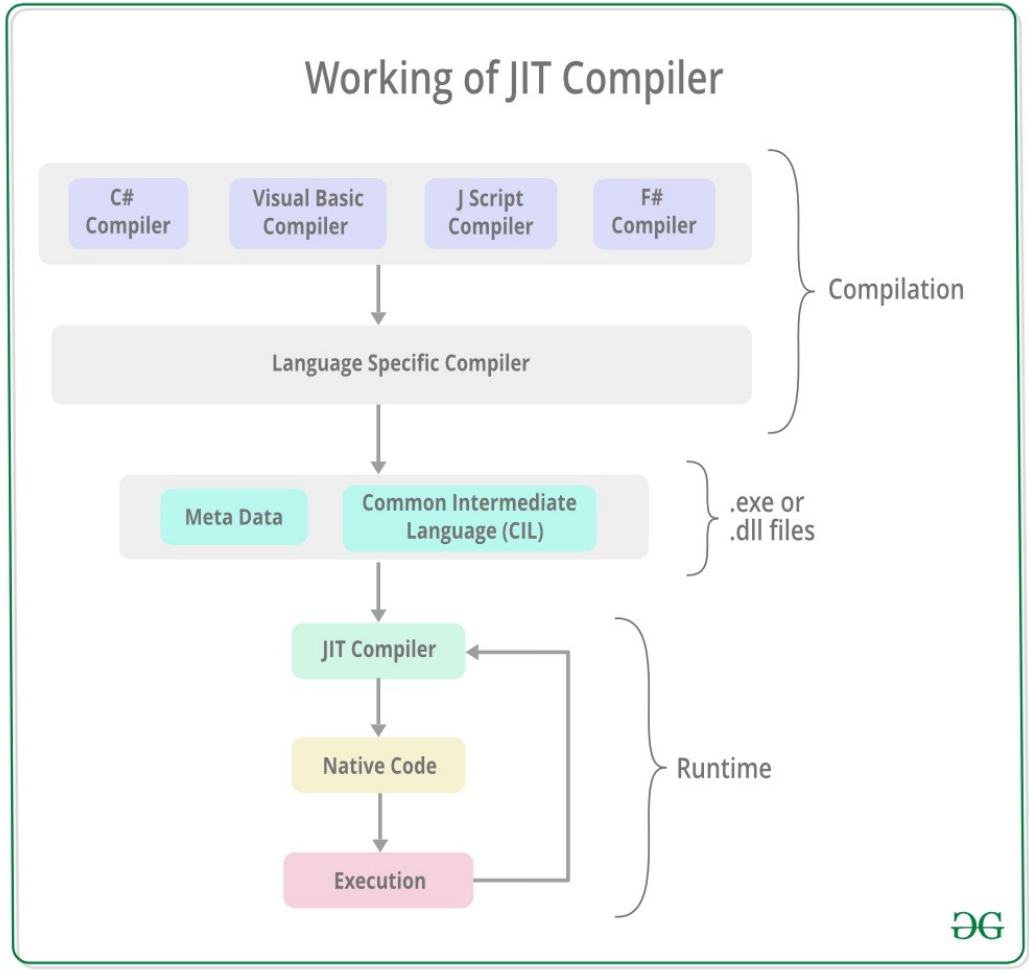
MSIL a JIT

MSIL = Microsoft Intermediate Language

CIL = Common Intermediate Language

.NET a Microsoft

Zdroj: https://www.geeksforgeeks.



Interpretované jazyky (Python)

Lexikální analýza → Syntaktická a. → Sémantická a. → Mezikód → Bytecode

Ale na rozdíl od Javy je předáno virtuálnímu stroji a ten vykoná bytecode.

Žádný strojový kód se **negeneruje**. (Otázka do pléna: kde se teda bere?)

Mnoho implementací (PyPy se snaží i o JIT).

Virtuální stroje pro Python

CPython – napsáno v C, „standard“ (.pyc)

JPython – Python do bytecode Javy, užívá JVM

IronPython – Python do MS IL (.NET) → CLR jako runtime

PyPy – runtime napsané v Pythonu

LLVM

=**L**ow **L**evel **V**irtual **M**achine

„Univerzální backend“; nejrůznější frontendy.

Vstupem je intermediální kód, výstupem strojový kód.

Silně podpořeno například od Apple, Open Source.

Vstupy: Ada, C, C++, Hasquel, Swift, ObjectC, Ruby, Python, R, Java Bytecode

Výstupy: ARM, Qualcomm, PowerPC, AMD TetraScale, Sparc, X86-32-64

LLVM - mezikód

Intermediate representation – blízké assembleru, přísně typové pro RISC

```
@.str = internal constant [14 x i8] c"hello, world\0A\00"

declare i32 @printf(i8*, ...)

define i32 @main(i32 %argc, i8** %argv) nounwind {
entry:
    %tmp1 = getelementptr [14 x i8]* @.str, i32 0, i32 0
    %tmp2 = call i32 (i8*, ...)* @printf( i8* %tmp1 ) nounwind
    ret i32 0
}
```

.NET

Překladač (minimální optimalizace) a JIT (většina práce)

Minimum Size – Maximum Speed – Vlastní

Podrobný popis:
https://docs.microsoft.com/en-us/dotnet/standard/managed-execution-process

MSIL: Microsoft intermediate language

Postaven na standardu ECMA (část III.):

https://www.ecma-international.org/publications-and-standards/standards/ecma-335/

JavaScript

- Celá řada interpretů (v jednotlivých prohlížečích)
- **V8:** Open Source investovaný Googlem
  - JIT (Crankshaft): Práce rozdělena do vláken, hlavní vlákno interpretuje, jiné vlákno optimalizuje
  - Výkon sledován profilerem, který dává pokyny pro optimalizaci
  - Jiná vlákna fungují jako Garbage collector
- Vlastní AST (Hydrogen)
- Hlavní optimalizace: inline kódu co to jen jde
- Po optimalizaci se překládá do strojového kódu