JVM Optimization 101
Sebastian Zarnekow
itemis
Static vs Dynamic Compilation

AOT vs JIT
JIT Compilation

- Compiled when needed
- Maybe immediately before execution
  - ...or when the VM decides it's important
  - ...or never?
JIT Compilation

- Makes bytecode fast
- Profiling scenarios at runtime
  - Invariants (known types, constants, null values)
  - Statistics (used branches, calls, polymorphism)
- Decisions based on executed code paths
- De-optimization if the world changes
Optimization Strategies
(Examples by Vladimir Ivanov)
Inlining

```c
int sum(int max) {
    int res = 0;
    for (int i = 0; i < max; i++) {
        res = plus(res, i);
    }
    return res;
}

int plus(int a, int b) {
    return a + b;
}
```
Inlining

```c
int sum(int max) {
    int res = 0;
    for (int i = 0; i < max; i++) {
        res = res + i;
    }
    return res;
}

int plus(int a, int b) {
    return a + b;
}
```
Inlining

- Gather Statistics
- Class Hierarchy Analysis
- Combine Caller and Callee (Call Site + Call)
- De-Virtualize and Optimize
public int constant() { return 3; }
public int method() {
    Tuple t = new Tuple(1, 2);
    return reduce(t);
}
public int reduce(Tuple t) {
    return t.first + getSecond(t);
}
public int getSecond(Tuple t) {
    return t.second;
}
public int method() {
    Tuple t = new Tuple(1, 2);
    return reduce(t);
}

public int reduce(Tuple t) {
    return t.first + getSecond(t);
}

public int getSecond(Tuple t) {
    return t.second;
}
public int method() {
    Tuple t = new Tuple(1, 2);
    return t.first + t.second;
}

public int reduce(Tuple t) {
    return t.first + getSecond(t);
}

public int getSecond(Tuple t) {
    return t.second;
}
public int method() {
    Tuple t = new Tuple(1, 2);
    return 3; t.first + t.second;
}

public int reduce(Tuple t) {
    return t.first + getSecond(t);
}

public int getSecond(Tuple t) {
    return t.second;
}
Escape Analysis

- Object Scope Analysis
- Avoid Unnecessary Objects on Heap
- Global Escape, Arg Escape, No Escape
- Eliminates Object Allocations and Unnecessary Locks
- May Eliminate Unnecessary Defensive Copies
On Stack Replacement (OSR)

- Methods with long running loops, back-branching
- Compile and replace while running
- Only rarely used in large systems
- Loops vs Benchmarks
public void foo(int[] arr, int a) {
    for(int i=0; i<arr.length; i++) {
        arr[i] += a;
    }
}

Loop Unrolling
public void foo(int[] arr, int a) {
    for(int i=0; i<arr.length; i+=4){
        arr[i] += a;
        arr[i+1] += a;
        arr[i+2] += a;
        arr[i+3] += a;
    }
    ...
}
Locks

cubic void m(Object newValue) {
    synchronized(this) {
        f1 = newValue;
    }
    synchronized(this) {
        f2 = newValue;
    }
}
public void m(Object newValue) {
    synchronized(this) {
        f1 = newValue;
    }
    synchronized(this) {
        f2 = newValue;
    }
}
public String m(Object newValue) {
    StringBuffer sb = new StringBuffer();
    sb.append(newValue);
    sb.append(newValue);
    sb.append(newValue);
    return sb.toString();
}
Intrinsics

This is not the code that you’re looking for.

- Known to the JIT compiler
- Bytecode is ignored
- Inserts optimized native code for current platform
- String::equals, String::indexOf, Object::hashCode, …
Optimization Strategies

- Inlining
- Escape Analysis
- Intrinsics
- Loop Unrolling
- Lock Fusion and Lock Elision
... and counting

compiler tactics
delayed compilation
tiered compilation
on-stack replacement
delayed reoptimization
program dependence graph representation
static single assignment representation

speculative (profile-based) techniques
optimistic nullness assertions
optimistic type assertions
optimistic type strengthening
optimistic array length strengthening
untaken branch pruning
optimistic N-morphic inlining
branch frequency prediction
call frequency prediction

proof-based techniques
exact type inference
memory value inference
memory value tracking
constant folding
reassociation
operator strength reduction
null check elimination
type test strength reduction
type test elimination
algebraic simplification
common subexpression elimination
integer range typing

flow-sensitive rewrites
conditional constant propagation
dominating test detection
flow-carried type narrowing
dead code elimination

language-specific techniques
class hierarchy analysis
devirtualization
symbolic constant propagation
autobox elimination
escape analysis
lock elision
lock fusion
de-reflection

memory and placement transformation
expression hoisting
expression sinking
redundant store elimination
adjacent store fusion
card-mark elimination
merge-point splitting

loop transformations
loop unrolling
loop peeling
safe-point elimination
iteration range splitting
range check elimination
loop vectorization

global code shaping
inlining (graph integration)
global code motion
heat-based code layout
switch balancing
throw inlining

control flow graph transformation
local code scheduling
local code bundling
delay slot filling
test graph-coloring register allocation
linear scan register allocation
live range splitting
copy coalescing
constant splitting
copy removal
address mode matching
instruction peepholing
DFA-based code generator
Diagnostics

- Print compiled methods
  -XX:+PrintCompilation

- Print info about inlining
  -XX:+UnlockDiagnosticVMOptions  -XX:+PrintInlining

- Print assembly code
  -XX:+PrintAssembly
Questions?
Tell me what you think.
Resources

- https://wikis.oracle.com/display/HotSpotInternals/PerformanceTechniques
- http://www.slideshare.net/iwanowww/jitcompiler-in-jvm-by
- http://www.slideshare.net/RafaelWinterhalter/an-introduction-to-jvm-performance
- http://www.slideshare.net/dougqh/jvm-mechanics-when-does-the
- https://wikis.oracle.com/display/HotSpotInternals/PerformanceTacticIndex
Off Topic

For Xtext users
Dear Xtext user,

please help us to better understand you and your needs, so we can improve the Xtext framework and our professional services accordingly. Answer as many questions as you like (the more the better) - they are all optional!

Thank you!

Gender

○ Female
○ Male

Age

Country