JVM JIT-compiler overview

Vladimir Ivanov
HotSpot JVM Compiler
Oracle Corp.

MAKE THE FUTURE JAVA

ORACLE

Agenda

- about compilers in general
  - … and JIT-compilers in particular
- about JIT-compilers in HotSpot JVM
- monitoring JIT-compilers in HotSpot JVM

Static vs Dynamic

AOT vs JIT

Dynamic and Static Compilation

Comparison

- Static compilation
  - “Ahead-Of-Time”(AOT) compilation
  - Source code → Native executable
  - Most of compilation work happens before executing

Dynamic and Static Compilation

Comparison

- Static compilation
  - “Ahead-Of-Time”(AOT) compilation
  - Source code → Native executable
  - Most of compilation work happens before executing
- Modern Java VMs use dynamic compilers (JIT)
  - “Just-In-Time” (JIT) compilation
  - Source code → Bytecode → Interpreter + JITted executable
  - Most of compilation work happens during application execution

Dynamic and Static Compilation

Comparison

- Static compilation (AOT)
  - can utilize complex and heavy analyses and optimizations

Dynamic and Static Compilation

Comparison

- Static compilation (AOT)
  - can utilize complex and heavy analyses and optimizations
    - … but static information sometimes isn’t enough
    - … and it’s hard to guess actual application behavior

Dynamic and Static Compilation

Comparison

- Static compilation (AOT)
  - can utilize complex and heavy analyses and optimizations
    - … but static information sometimes isn’t enough
    - … and it’s hard to guess actual application behavior
  - moreover, how to utilize specific platform features?
    - like SSE4.2 / AVX / AVX2, TSX, AES-NI, RdRand

Dynamic and Static Compilation

Comparison

- Modern Java VMs use dynamic compilers (JIT)
  - aggressive optimistic optimizations
    - through extensive usage of profiling data

Dynamic and Static Compilation

Comparison

- Modern Java VMs use dynamic compilers (JIT)
  - aggressive optimistic optimizations
    - through extensive usage of profiling data
    - … but resources are limited and shared with an application

Dynamic and Static Compilation

Comparison

- Modern Java VMs use dynamic compilers (JIT)
  - aggressive optimistic optimizations
    - through extensive usage of profiling data
    - … but resources are limited and shared with an application
  - thus:
    - startup speed suffers
    - peak performance may suffer as well (but not necessarily)

Dynamic and Static Compilation

Comparison

- Modern Java VMs use dynamic compilers (JIT)
  - aggressive optimistic optimizations
    - through extensive usage of **profiling** data
    - … but resources are limited and shared with an application
  - thus:
    - startup speed suffers
    - peak performance may suffer as well (but not necessarily)

Profiling

- Gathers data about code during execution
  - invariants
    - types, constants (e.g. null pointers)
  - statistics
    - branches, calls
- Gathered data can be used during optimization
  - Educated guess
  - Guess can be wrong

Optimistic Compilers

- Assume profile is accurate
  - Aggressively optimize based on profile
  - Bail out if they’re wrong
- ...and hope that they’re usually right

Profile-guided optimizations (PGO)

- Use profile for more efficient optimization
- PGO in JVMs
  - Always have it, turned on by default
  - Developers (usually) not interested or concerned about it
  - Profile is always consistent to execution scenario

Optimistic Compilers

Example

```
public void f() {
    A a;
    if (cond /*always true*/) {
        a = new B();
    } else {
        a = new C(); // never executed
    }
   a.m(); // exact type of a is either B or C
}
```

Optimistic Compilers

Example

```
public void f() {
    A a;
    if (cond /*always true*/) {
        a = new B();
    } else {
        toInterpreter(); // switch to interpreter
    }
   a.m(); // exact type of a is B
}
```

Dynamic Compilation in (J)VM

Dynamic Compilation (JIT)

- Can do non-conservative optimizations at runtime
- Separates optimization from product delivery cycle
  - Update JVM, run the same application, realize improved performance!
  - Can be "tuned" to the target platform

Dynamic Compilation (JIT)

- Knows a lot about Java program
  - loaded classes, executed methods, profiling
- Makes optimization based on that
- May re-optimize if previous assumption was wrong

JVM

- Runtime
  - class loading, bytecode verification, synchronization
- JIT
  - profiling, compilation plans
  - aggressive optimizations
- GC
  - different algorithms: throughput vs response time vs footprint

JVM: Makes Bytecodes Fast

- JVMs eventually JIT-compile bytecodes
  - To make them fast
  - compiled when needed
    - Maybe immediately before execution
    - ...or when we decide it’s important
    - ...or never?
  - Some JITs are high quality optimizing compilers

JVM: Makes Bytecodes Fast

- JVMs eventually JIT-compile bytecodes
- But cannot use existing static compilers directly
  - different cost model
    - time & resource constraints (CPU, memory)
  - tracking OOPs (ptrs) for GC
  - Java Memory Model (volatile reordering & fences)
  - New code patterns to optimize

JVM: Makes Bytecodes Fast

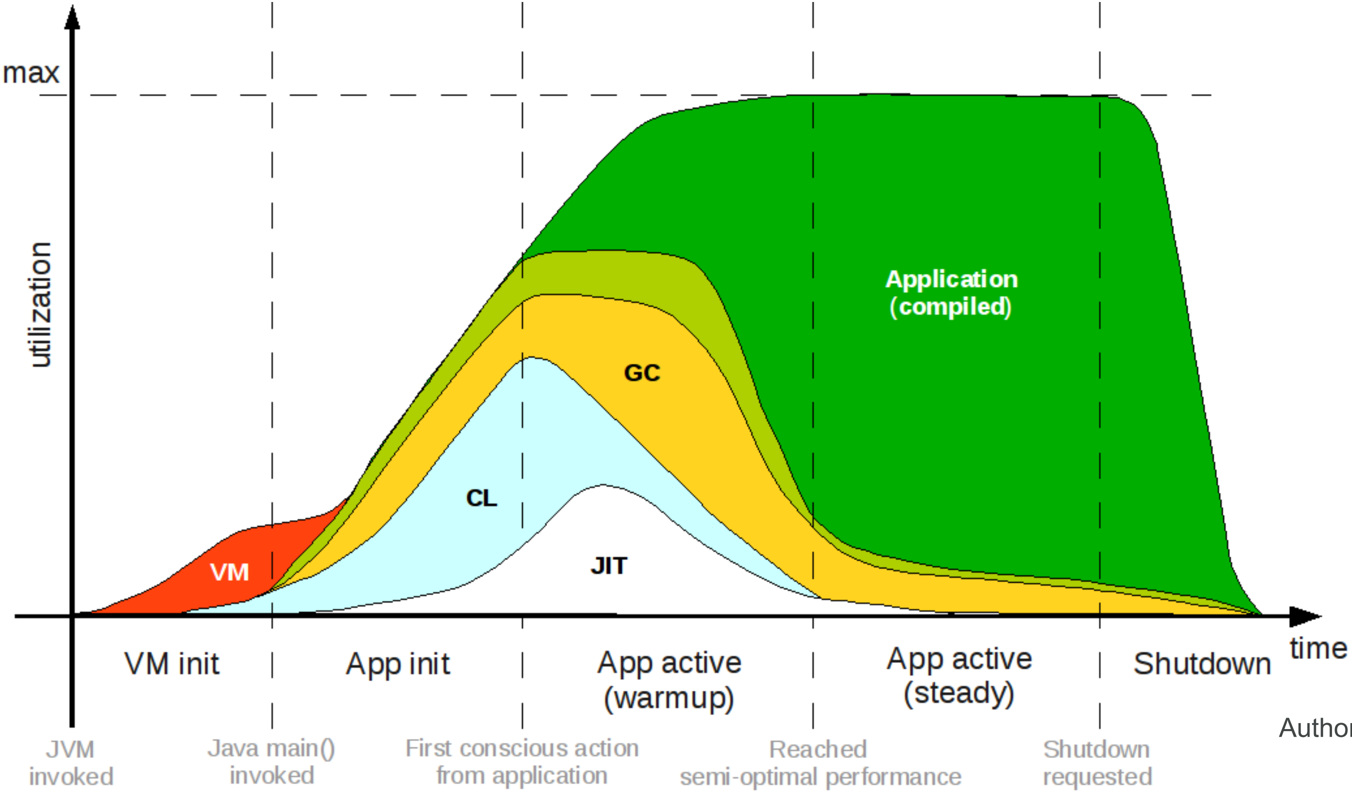
- JIT'ing requires Profiling
  - Because you don't want to JIT everything
- Profiling allows focused code-gen
- Profiling allows better code-gen
  - Inline what’s hot
  - Loop unrolling, range-check elimination, etc
  - Branch prediction, spill-code-gen, scheduling

Dynamic Compilation (JIT)

Overhead

- Is dynamic compilation overhead essential?
  - The longer your application runs, the less the overhead
- Trading off **compilation** time, not application time
  - Steal some cycles very early in execution
  - Done automagically and transparently to application
- Most of “perceived” overhead is compiler waiting for more data
  - ...thus running semi-optimal code for time being

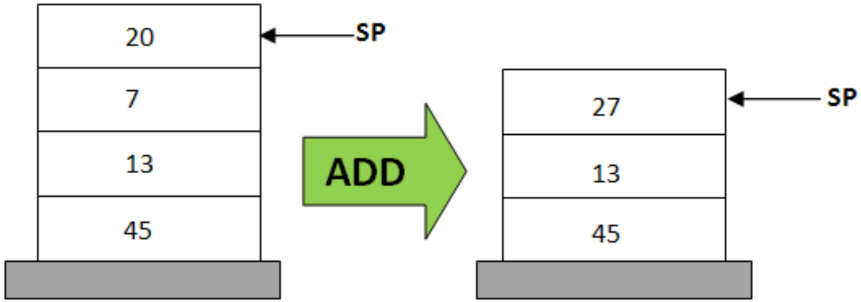
JVM



Author: Aleksey Shipilev

Mixed-Mode Execution

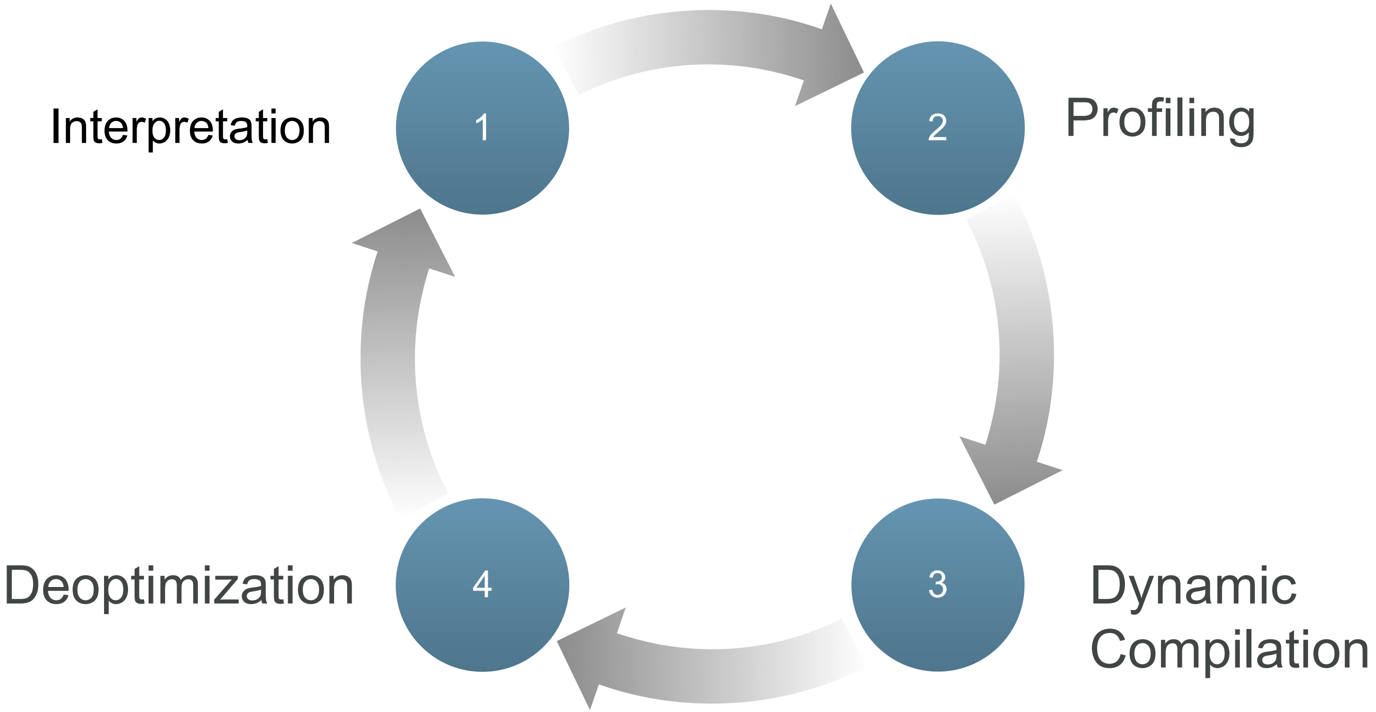
- Interpreted
  - Bytecode-walking
  - Artificial stack machine



- Compiled
  - Direct native operations
  - Native register machine

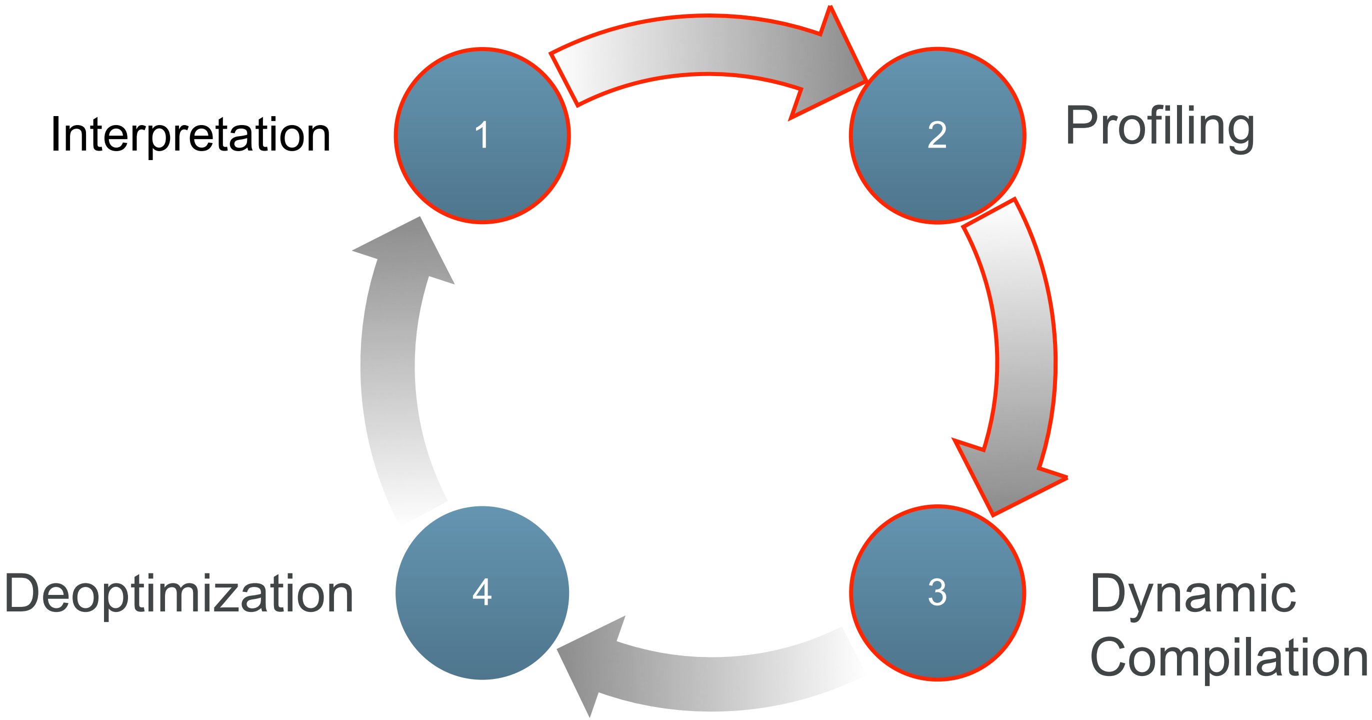
```
...
add     $0x7,%r8d
...
```

Bytecode Execution



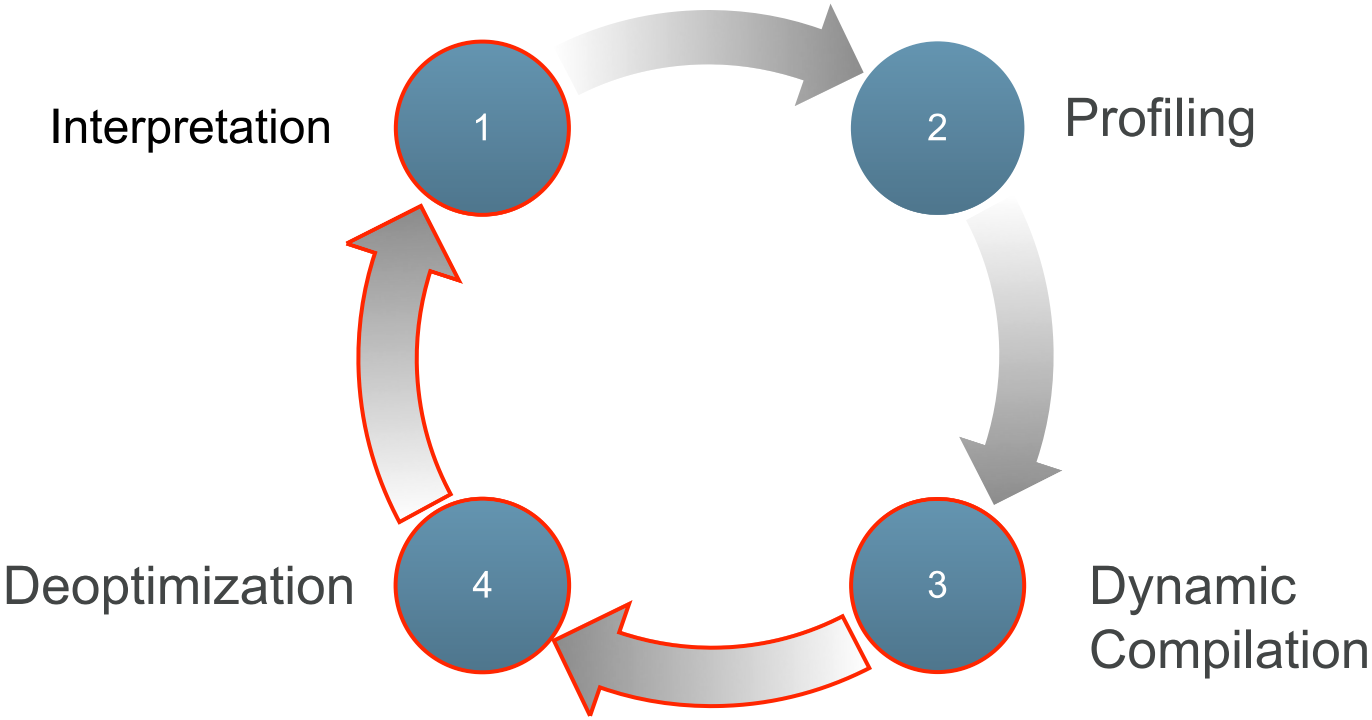
Bytecode Execution

Normal execution



Bytecode Execution

Recompilation

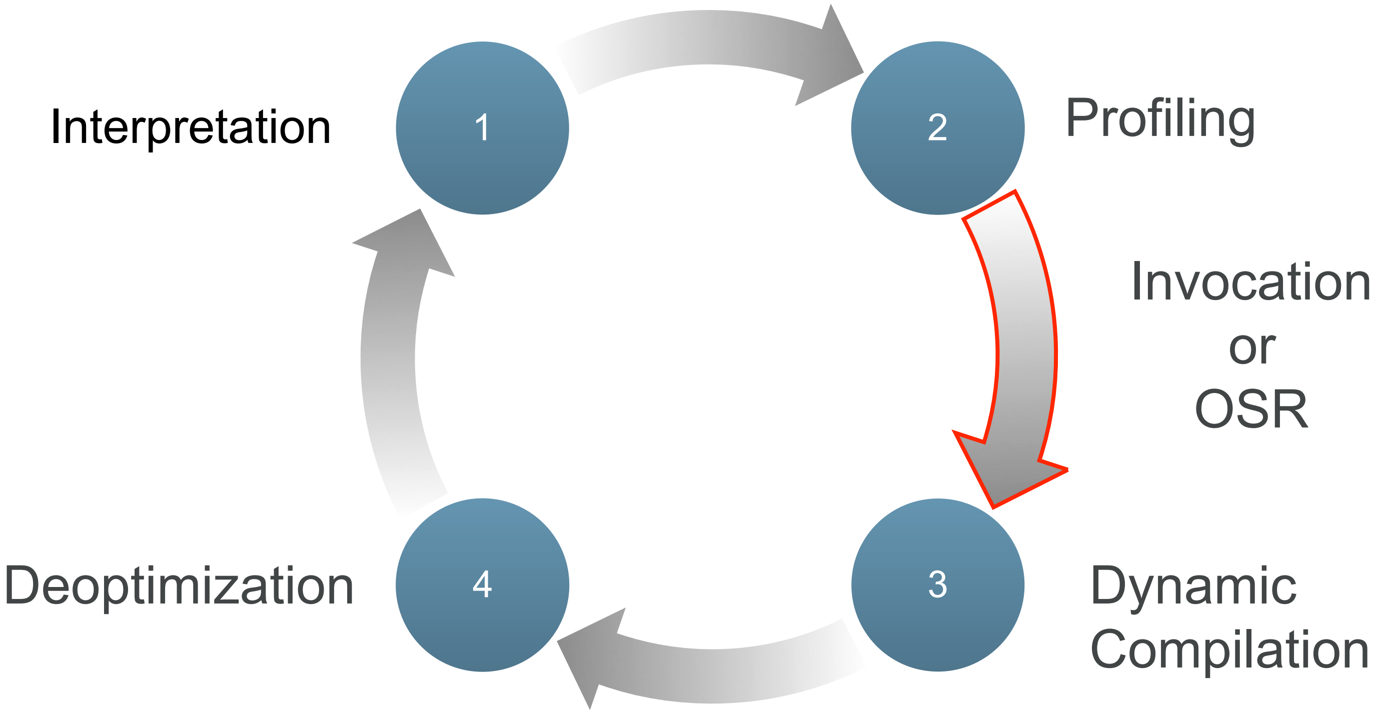


Deoptimization

- Bail out of running native code
  - stop executing native (JIT-generated) code
  - start interpreting bytecode
- It’s a complicated operation at runtime…
  - different calling conventions
  - different stack layout

Bytecode Execution

Interpretation => Native code execution



OSR: On-Stack Replacement

- Running method never exits? But it’s getting really hot?
  - Generally means loops, back-branching
- Compile and replace while running

- Not typically useful in large systems
  - … but looks great on benchmarks!

Optimizations

Optimizations in HotSpot JVM

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

JVM: Makes Virtual Calls Fast

- C++ avoids virtual calls
  - … because they are “slow”
  - … hard to see “through” virtual call

JVM: Makes Virtual Calls Fast

- C++ avoids virtual calls
- Java embraces them
  - … and makes them fast
  - both invokevirtual & invokeinterface

invokevirtual vs invokeinterface

```
class B extends A implements I, J, K { ... }
class C           implements I, J, K { ... }
```

invokevirtual A.m B

invokevirtual B.m B

invokevirtual C.m C

invokeinterface I.m B

invokeinterface I.m C

invokevirtual

```
 <+0>: mov     0x8(%rsi),%r10d                ; load Klass*
 <+4>: shl     $0x3,%r10                      ;

 <+8>: mov     0x10(%r8),%r11                 ; load vmindex

<+12>: mov     0x1c8(%r10,%r11,8),%rbx        ; load entry point address

<+20>: test    %rbx,%rbx
<+23>: je      <+32>
<+29>: jmpq    *0x48(%rbx)
<+32>: jmpq    <throw_AbstractMethodError_stub>
```

invokeinterface

```
 <+0>: mov    0x8(%rsi),%r10d
 <+4>: shl    $0x3,%r10
 <+8>: mov    0x20(%rdx),%eax
<+10>: shl    $0x3,%rax
<+15>: mov    0x48(%rax),%rax
<+19>: mov    0x10(%rdx),%rbx
<+23>: mov    0x128(%r10),%r11d
<+30>: lea    0x1c8(%r10,%r11,8),%r11
<+38>: lea    (%r10,%rbx,8),%r10
<+42>: mov    (%r11),%rbx
<+45>: cmp    %rbx,%rax
<+48>: je     <+71>
```

```
<+50>: 0x...f12: test   %rbx,%rbx
<+53>: 0x...f15: je     <+96>
<+59>: 0x...f1b: add    $0x10,%r11
<+63>: 0x...f1f: mov    (%r11),%rbx
<+66>: 0x...f22: cmp    %rbx,%rax
<+69>: 0x...f25: jne    <+50>
<+71>: 0x...f27: mov    0x8(%r11),%r11d
<+75>: 0x...f2b: mov    (%r10,%r11,1),%rbx
<+79>: 0x...f2f: test   %rbx,%rbx
<+82>: 0x...f32: je     <+91>
<+88>: 0x...f38: jmpq   *0x48(%rbx)
<+91>: 0x...f3b: jmpq   <throw_AME_stub>
<+96>: 0x...f40: jmpq   <throw_ICCE_stub>
```

JVM: Makes Virtual Calls Fast

- Well, mostly fast
  - Class Hierarchy Analysis (CHA)
  - profiling (exact types @ call sites)
- Fallback to slower mechanisms if needed
  - inline caches (ICs)
  - virtual dispatch

JVM: Makes Virtual Calls Fast

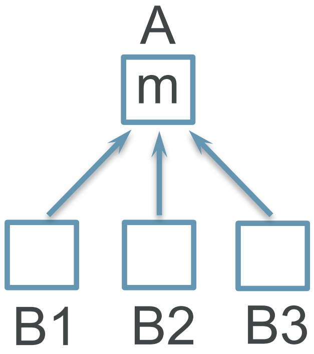
A a = new B1();

a.m();

invokevirtual A.m() B1

CHA: A.m()

Profile: B1 => A.m()



JVM: Makes Virtual Calls Fast

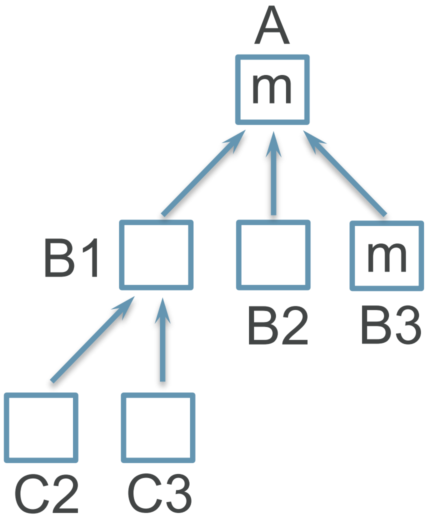
A a = new C2();

a.m();

invokevirtual A.m() C2

CHA: A.m() || B3.m() => failed

Profile: C2 => A.m()



JVM: Makes Virtual Calls Fast

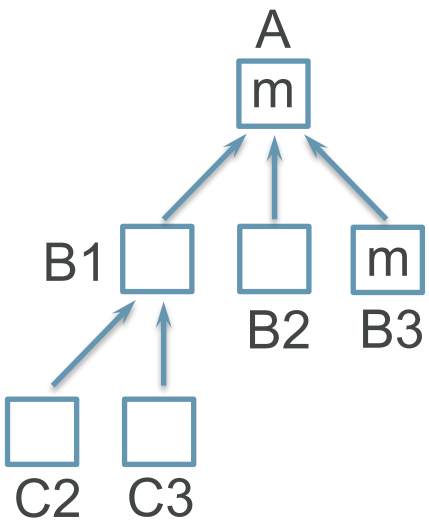
A a = (…) ? new C2() : new C3();

a.m();

invokevirtual A.m() C2/C3

CHA: A.m() || B3.m() => failed

Profile: C2, C3 => A.m()



JVM: Makes Virtual Calls Fast

- CHA & profiling turns most virtual calls into static calls
- Fallback to slower mechanisms
  - new classes loaded => adjusts CHA
  - uncommon traps
- When JVM fails to make the call static, use inline caches (ICs)
- When ICs fail, issue virtual call

Inlining

- Combine caller and callee into one unit
  - e.g. based on profile
  - … or proved using CHA (Class Hierarchy Analysis)
  - Perhaps with a type test (guard)
- Optimize as a whole (single compilation unit)
  - More code means better visibility

Inlining

Before

```
int addAll(int max) {
    int accum = 0;
    for (int i = 0; i < max; i++) {
        accum = add(accum, i);
    }
    return accum;
}

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

Inlining

After

```
int addAll(int max) {
    int accum = 0;
    for (int i = 0; i < max; i++) {
        accum = accum + i;
    }
    return accum;
}
```

Inlining and devirtualization

- Inlining is the most profitable compiler optimization
  - Rather straightforward to implement
  - Huge benefits: expands the scope for other optimizations
- OOP needs polymorphism, that implies virtual calls
  - Prevents naïve inlining
  - Devirtualization is required
  - (This does not mean you should not write OOP code)

Call Site

Flavors

- The place where you make a call
- Types
  - Monomorphic (**“one shape”**)
    - Single target class
  - Bimorphic (**“two shapes”**)
  - Polymorphic (**“many shapes”**)
  - Megamorphic (**“too many shapes”**)

Devirtualization in JVM

- Analyzes hierarchy of currently loaded classes (CHA)
- Efficiently devirtualizes all monomorphic calls
- Able to devirtualize polymorphic calls
- JVM may inline dynamic methods
  - Reflection calls
  - Runtime-synthesized methods
  - JSR 292

Devirtualization in JVM

- Class Hierarchy Analysis (CHA)
  – most of monomorphic call sites
- Type profiling
  – monomorphic, bimorphic & polymorphic call sites
- JVM may inline dynamic methods
  – Reflection calls, runtime-synthesized methods, JSR 292

Feedback multiplies optimizations

- Profiling and CHA produces information
  - ...which lets the JIT ignore unused paths
  - ...and helps the JIT sharpen types on hot paths
  - ...which allows calls to be devirtualized
  - ...allowing them to be inlined
  - ...expanding an ever-widening optimization horizon
- Result:

  Large native methods containing tightly optimized machine code for hundreds of inlined calls!

HotSpot JVM

Existing JVMs

- Oracle HotSpot
- Oracle JRockit
- IBM J9
- Excelsior JET
- Azul Zing
- SAPJVM
- …

HotSpot JVM

JIT-compilers

- client / C1
- server / C2
- tiered mode (C1 + C2)

HotSpot JVM

JIT-compilers

- client / C1
  - $ java –client
    - only available in 32-bit VM
  - fast code generation of acceptable quality
  - basic optimizations
  - doesn’t need profile
  - compilation threshold: 1,5k invocations

HotSpot JVM

JIT-compilers

- server / C2
  - $ java –server
  - highly optimized code for speed
  - many aggressive optimizations which rely on profile
  - compilation threshold: 10k invocations

HotSpot JVM

JIT-compilers comparison

- Client / C1

  \+ fast startup

  – peak performance suffers

- Server / C2

  \+ very good code for hot methods

  – slow startup / warmup

Tiered compilation

C1 + C2

- -XX:+TieredCompilation
  - since 7; default for –server since 8
- Multiple tiers of interpretation, C1, and C2
- Level0=Interpreter
- Level1-3=C1
  - #1: C1 w/o profiling
  - #2: C1 w/ basic profiling
  - #3: C1 w/ full profiling
- Level4=C2

Monitoring JIT

Monitoring JIT-Compiler

- how to print info about **compiled methods**?
  - -XX:+PrintCompilation
- how to print info about **inlining decisions**
  - -XX:+PrintInlining
- how to control **compilation policy**?
  - -XX:CompileCommand=…
- how to print **assembly code**?
  - -XX:+PrintAssembly
  - -XX:+PrintOptoAssembly (C2-only)

Print Compilation

- -XX:+PrintCompilation
- Print methods as they are JIT-compiled
- Class + name + size

Print Compilation

Sample output

```
$ java -XX:+PrintCompilation
  988   1       java.lang.String::hashCode (55 bytes)
 1271   2       sun.nio.cs.UTF_8$Encoder::encode (361 bytes)
 1406   3       java.lang.String::charAt (29 bytes)
```

Print Compilation

Other useful info

- 2043 470 % ! jdk.nashorn.internal.ir.FunctionNode::accept @ 136 (265 bytes)

  % == OSR compilation

  ! == has exception handles (may be expensive)

  s == synchronized method

- 2028 466 n java.lang.Class::isArray (native)

  n == native method

Print Compilation

Not just compilation notifications

- 621 160   java.lang.Object::equals (11 bytes)   made not entrant
  – don‘t allow any new calls into this compiled version
- 1807 160 java.lang.Object::equals (11 bytes)   made zombie
  – can safely throw away compiled version

No JIT At All?

- Code is too large
- Code isn't too «hot»
  - executed not too often

Print Inlining

- -XX:+UnlockDiagnosticVMOptions -XX:+PrintInlining
- Shows hierarchy of inlined methods
- Prints reason, if a method isn't inlined

Print Inlining

```
$ java -XX:+PrintCompilation -XX:+UnlockDiagnosticVMOptions -XX:+PrintInlining
   75    1            java.lang.String::hashCode (55 bytes)
   88    2            sun.nio.cs.UTF_8$Encoder::encode (361 bytes)
                      @ 14   java.lang.Math::min (11 bytes)   (intrinsic)
                      @ 139   java.lang.Character::isSurrogate (18 bytes)   never executed
  103    3             java.lang.String::charAt (29 bytes)
```

Print Inlining

```
$ java -XX:+PrintCompilation -XX:+UnlockDiagnosticVMOptions -XX:+PrintInlining
   75   1        java.lang.String::hashCode (55 bytes)
   88   2        sun.nio.cs.UTF_8$Encoder::encode (361 bytes)
                 @ 14   java.lang.Math::min (11 bytes)   (intrinsic)
                 @ 139   java.lang.Character::isSurrogate (18 bytes)   never executed
  103   3        java.lang.String::charAt (29 bytes)
```

Intrinsic

- Known to the JIT compiler
  - method bytecode is ignored
  - inserts “best” native code
- e.g. optimized sqrt in machine code
- Existing intrinsics
  - String::equals, Math::*, System::arraycopy, Object::hashCode, Object::getClass, sun.misc.Unsafe::*

Inlining Tuning

- -XX:MaxInlineSize=35
  - Largest inlinable method (bytecode)
- -XX:InlineSmallCode=#
  - Largest inlinable compiled method
- -XX:FreqInlineSize=#
  - Largest frequently-called method…
- -XX:MaxInlineLevel=9
  - How deep does the rabbit hole go?
- -XX:MaxRecursiveInlineLevel=#
  - recursive inlining

Machine Code

- -XX:+PrintAssembly
  - http://wikis.sun.com/display/HotSpotInternals/PrintAssembly
- Knowing code compiles is good
- Knowing code inlines is better
- Seeing the actual assembly is best!

-XX:CompileCommand=

- Syntax
  - “[command] [method] [signature]”
- Supported commands
  - **exclude** – never compile
  - **inline** – always inline
  - **dontinline** – never inline
- Method reference
  - class.name::methodName
- Method signature is optional

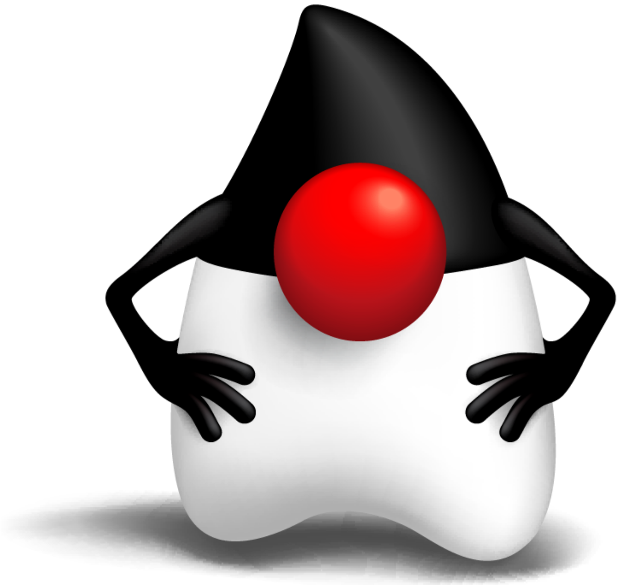
-XX:+LogCompilation

- Dumps detailed compilation-related info
  - info hotspot.log / hotspot_pid%.log (XML format)
- How to process
  - JITwatch
    - visualizes –XX:+LogCompilation output
  - logc.jar
    - http://hg.openjdk.java.net/jdk9/hs-comp/hotspot/share/tools/LogCompilation/

What Have We Learned?

- How JIT compilers work
- How HotSpot JIT works
- How to monitor the JIT in HotSpot

Questions?



vladimir.x.ivanov@oracle.com
@iwan0www

Optimizations

Loop Unrolling

Before

```
public void foo(int[] arr, int a) {
    for (int i = 0; i < arr.length; i++) {
        arr[i] += a;
    }
}
```

Loop Unrolling

After?

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

Loop unrolling

After!

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

    for (; i < arr.length; i++) {
        arr[i] += a;
    }
}
```

Loop unrolling

Machine code

```
0x...70: vmovdqu 0x10(%rsi,%r8,4),%ymm1
0x...77: vpaddd  %ymm0,%ymm1,%ymm1
0x...7b: vmovdqu %ymm1,0x10(%rsi,%r8,4)

0x...82: add     $0x8,%r8d

0x...86: cmp     %r9d,%r8d
0x...89: jl      0x...70
```

Lock Coarsening

Before

```
public void m(Object newValue) {
    synchronized(this) {
        field1 = newValue;
    }
    synchronized(this) {
        field2 = newValue;
    }
}
```

Lock Coarsening

After

```java
public void m(Object newValue) {
    synchronized(this) {
        field1 = newValue;
        field2 = newValue;
    }
}
```

Lock Elision

Before

```
public List<?> m() {
    List<Object> list = new ArrayList<>();
    synchronized (list) {
        list.add(someMethod());
    }
    return list;
}
```

Lock Elision

After

```java
public List<?> m() {
    List<Object> list = new ArrayList<>();
    list.add(someMethod());
    return list;
}
```

Escape Analysis

Before

```
public int m1() {
    Pair p = new Pair(1, 2);
    return m2(p);
}
public int m2(Pair p) {
    return p.first + m3(p);
}
public int m3(Pair p) { return p.second;}
```

Escape Analysis

After deep inlining

```java
public int m1() {
    Pair p = new Pair(1, 2);
    return p.first + p.second;
}
```

Escape Analysis

After

```java
public int m1() {
    return 3;
}
```

MAKE THE FUTURE JAVA

JavaOne™

ORACLE®