JavaOne™

JIT-compiler in JVM seen by a Java developer

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 Differences

- 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 executing

Dynamic and Static Compilation Differences

- Static compilation (AOT)
  - can utilize complex and heavy analyses and optimizations
  - … but static information sometimes isn't enough
  - … and it's hard to rely on profiling info, if any
  - moreover, how to utilize specific platform features (like SSE 4.2)?

Dynamic and Static Compilation Differences

- Modern Java VMs use dynamic compilers (JIT)
  – aggressive optimistic optimizations
    - through extensive usage of profiling info
  – … but budget is limited and shared with an application
  – startup speed suffers
  – peak performance may suffer as well (not necessary)

JIT-compilation

- Just-In-Time compilation
- Compiled when needed
- Maybe immediately before execution
  - ...or when we decide it's important
  - ...or never?

Dynamic Compilation in JVM

JVM

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

JVM: Makes Bytecodes Fast

- JVMs eventually JIT bytecodes
  - To make them fast
  - Some JITs are high quality optimizing compilers
- But cannot use existing static compilers directly:
  - Tracking OOPs (ptrs) for GC
  - Java Memory Model (volatile reordering & fences)
  - New code patterns to optimize
  - Time & resource constraints (CPU, memory)

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)

- Knows about
  - loaded classes, methods the program has executed
- Makes optimization decisions based on code paths executed
  - Code generation depends on what is observed:
    - loaded classes, code paths executed, branches taken
- May re-optimize if assumption was wrong, or alternative code paths taken
  - Instruction path length may change between invocations of methods as a result of de-optimization / re-compilation

Dynamic Compilation (JIT)

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

Profiling

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

Profile-guided optimization (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

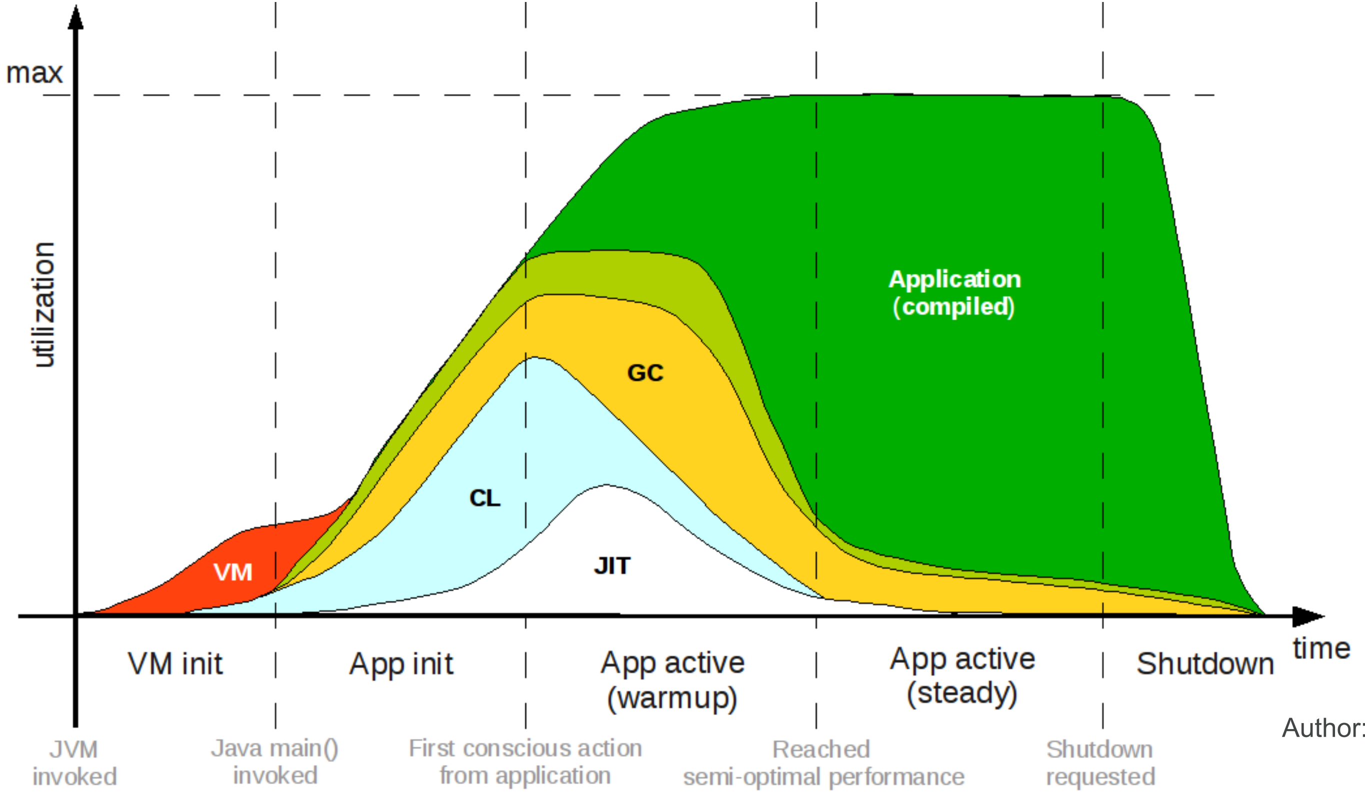
- Assume profile is accurate
  - Aggressively optimize based on profile
  - Bail out if we’re wrong
- ...and hope that we’re usually right

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

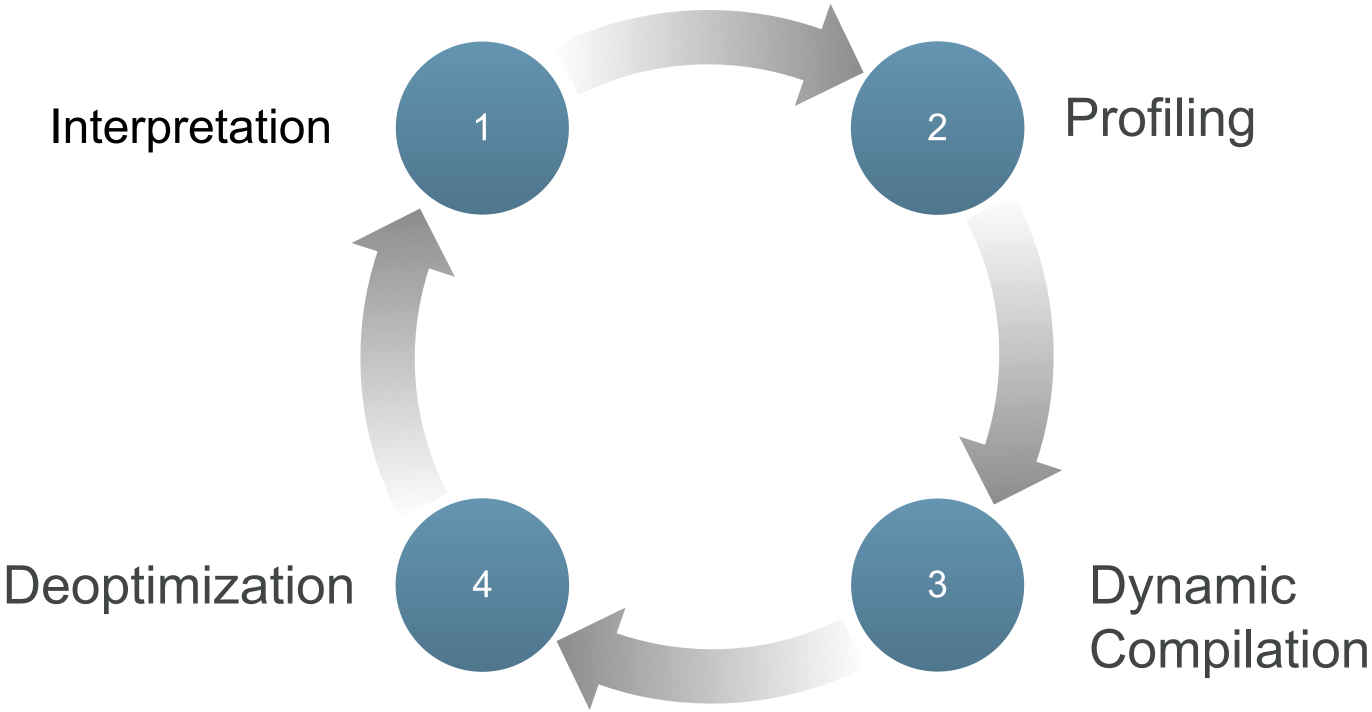


Author: Alexey Shipilev

Mixed-Mode Execution

- Interpreted
  – Bytecode-walking
  – Artificial stack machine
- Compiled
  – Direct native operations
  – Native register machine

Bytecode Execution



Deoptimization

- Bail out of running native code
  - stop executing native (JIT-generated) code
  - start interpreting bytecode
- It's a complicated operation at runtime…

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
- 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
- Java embraces them – and makes them fast
  - Well, mostly fast – JIT's do Class Hierarchy Analysis (CHA)
  - CHA turns most virtual calls into static calls
  - JVM detects new classes loaded, adjusts CHA
    - May need to re-JIT
  - When CHA fails to make the call static, inline caches
  - When IC's fail, virtual calls are back to being slow

Call Site

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

Inlining

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

Inlining

```
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

```
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)

JVM Devirtualization

- Developers shouldn't care
- Analyze hierarchy of currently loaded classes
- Efficiently devirtualize all monomorphic calls
- Able to devirtualize polymorphic calls
- JVM may inline dynamic methods
  - Reflection calls
  - Runtime-synthesized methods
  - JSR 292

Feedback multiplies optimizations

- On-line 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!

Loop unrolling

```
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=i+4) {
      arr[i] += a;   arr[i+1] += a;   arr[i+2] += a;   arr[i+3] += a;
  }
}
```

Loop unrolling

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

  for (int i = new_limit*4; i < arr.length; i++) {
     arr[i] += a;
  }}
```

Lock Coarsening

```
pubic void m1(Object newValue) {
    syncronized(this) {
        field1 = newValue;
    }
    syncronized(this) {
        field2 = newValue;
    }
}
```

Lock Coarsening

```
pubic void m1(Object newValue) {
    syncronized(this) {
        field1 = newValue;
        field2 = newValue;
    }
}
```

Lock Eliding

```
public void m1() {
  List list = new ArrayList();
  synchronized (list) {
    list.add(someMethod());
  }
}
```

Lock Eliding

```
public void m1() {
  List list = new ArrayList();
  synchronized (list) {
    list.add(someMethod());
  }
}
```

Lock Eliding

```
public void m1() {
  List list = new ArrayList();
  list.add(someMethod());
}
```

Escape Analysis

Initial version

```
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

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

Escape Analysis

Optimized version

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

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::*

HotSpot JVM

JVMs

- Oracle HotSpot
- IBM J9
- Oracle JRockit
- Azul Zing
- Excelsior JET
- Jikes RVM

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
- 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)
```

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

What Have We Learned?

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

Related Talks

“Quantum Performance Effects”

Sergey Kuksenko, Oracle

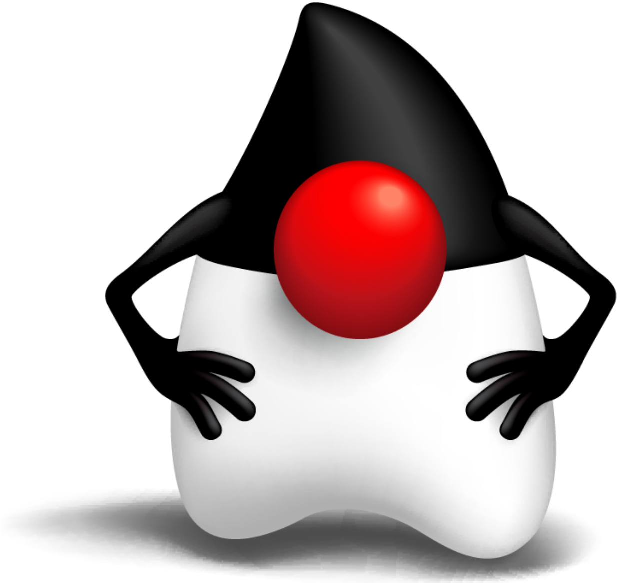
today, 13:30-14:30, «San-Francisco» hall

“Bulletproof Java Concurrency”

Aleksey Shipilev, Oracle

today, 15:30-16:30, «Moscow» hall

Questions?



vladimir.x.ivanov@oracle.com
@iwanowww

MAKE THE FUTURE JAVA

JavaOne™

ORACLE®