# AOT/Graal changes walkthrough

Igor Veresov
HotSpot Compiler Team

Java
Your
Next
(Cloud)

# Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

# Graal changes

- Indirect load of constants, including constant replacement
- Class initialization
- Profiling (tiered compilation support)
- Inlining

# Constants

- Constants embedded in the code (field offsets, GC barriers, etc) — need to be validated before any code is run.
- Global constants (heap top/end, card table base, etc) — eagerly initialized when the library is loaded. *GraalHotSpotVMConfigNode* represents these values. Folds to constants in JIT mode. Indirect loads in AOT.
- Local constants (classes, objects, method counters) — lazily initialized at runtime.

# Constant replacement

- Automatic — constants are replaced by nodes that provide indirection and handle lazy resolution if necessary: *ReplaceConstantNodesPhase.*
- Replaces classes, method counters and string constants with *ResolveConstantNode* and *ResolveMethodAndLoadCountersNode.*
- Some well-known class constants are eagerly resolved, replaced with *LoadConstantIndirectlyNode* (for example primitive array classes).
- Class mirror constants are replaced with indirections through class constants: *LoadJavaMirrorWithKlassPhase*.

# Constant replacement optimizations

- Currently single resolution for each constant (placed in a dominating block).
- Reuse of dominating class initialization nodes (*InitializeKlassNode).*
- Resolution of root method holder class and its superclasses can be omitted since it’s guaranteed to be initialized (and hence resolved). Replaced by *LoadConstantIndirectlyNode*.

# Constant replacement

Later all these nodes are lowered into snippets that do (for example for class constants):

```
KlassPointer result = LoadConstantIndirectlyNode.loadKlass(constant);
if (probability(VERY_SLOW_PATH_PROBABILITY, result.isNull())) {
    result = ResolveConstantStubCall.resolveKlass(constant,
                EncodedSymbolNode.encode(constant));
}
```

# Constant replacement

## Source files

- Globals: GraalHotSpotVMConfigNode.java
- Replace constant nodes with nodes that do the symbol to constant mapping: ReplaceConstantNodesPhase.java
- Nodes that appear in the graph after replacement: ResolveMethodAndLoadCounters.java, ResolveConstantNode.java, LoadConstantIndirectlyNode.java, LoadConstantIndireclyFixedNode.java
- Lowering: ResolveConstantSnippets.java
- ResolveConstantStubCall.java
- LIR: AMD64HotSpotConstantRetrievalOp.java, AMD64HotSpotLoadAddressOp.java

# Class initialization

- *InitializeKlassNode* is inserted at every initialization point through new parser plugin interface *ClassInitializationPlugin* (see *HotSpotClassInitializationPlugin* for implementation).
- Some optimizations are possible at parsing phase (type >=: holder, except for interfaces).
- Separate phase (*EliminateRedundantInitializationPhase*) with data flow analysis to remove redundant class initialization nodes. Good after loop peeling.
- Lowered into if-null-then-call-runtime diamond.

# Class initialization

## Source files

- Plugin interface: ClassInitializationPlugin.java
- Plugin the does node insertion: HotSpotClassInitializationPlugin.java
- Node that does class initialization: InitalizeKlassNode.java
- Plugin is used at initialization points: BytecodeParser.java
- Optimization: EliminateRedundantInitializationPhase.java
- Lowering: ResolveConstantSnippets.java
- LIR: AMD64HotSpotConstantRetrievalOp.java, AMD64HotSpotLoadAddressOp.java

# Tiered support

- Similar to level 2 profiling, higher thresholds.
- Counting invocation and back branches, calling back to runtime to re-JIT.
- Profiling nodes are inserted in the parser via new plugin interface *ProfilingPlugin* (see *HotSpotProfilingPlugin* for implementation).
- Later processed in *FinalizeProfileNodesPhase*. Assigns inlinee notification frequencies, and random sources (more about this later).

# Inserting profiling nodes

## Source files

- Plugin interface: ProfilingPlugin.java
- Plugin that does node insertion: HotSpotProfilingPlugin.java
- Plugin is used at profile points: BytecodeParser.java
- Base profile nodes: ProfileNode.java, ProfileWithNotificationNode.java
- Profile nodes: ProfileInvokeNode.java, ProfileBranchNode.java

# Kinds of profiling

- Profiling nodes are lowered in two ways. Normal profiling and probabilistic profiling.
- Normal profiling is what you'd expect (see *ProfileSnippets*).
- Probabilistic profiling tries to minimize cache line ping-ponging (see *ProbabilisticProfileSnippets*).

# Normal profiling & profile nodes optimizations

## Source files

- Optimization: ProfileNode.java (see ProfileNode.simplify())
- Frequency assignment: FinalizeProfileNodesPhase.java
- Lowering: ProfileSnippets.java

# Probabilistic profiling

- Threads executing same methods are competing for the same cache line, where the counters are.
- The idea is to not do increments every time, but do it with some predefined probability.
- Branch on random:  if (random() & ((1 << prob_log) - 1) == 0) { …. }

# Probabilistic profiling

```
if ((random() & ((1 << probLog) - 1)) == 0) { // branch on random
    int counterValue = counters.readInt(invocationCounterOffset) +
                       (invocationCounterIncrement << probLog);
    counters.writeInt(invocationCounterOffset, counterValue);
    if (freqLog >= 0) { // folds
        int probabilityMask = (1 << probLog) - 1;
        int frequencyMask = (1 << freqLog) - 1;
        int mask = frequencyMask & ~probabilityMask;
        if (counterValue & (mask << invocationCounterShift)) == 0)) {
            methodInvocationEvent(counters);
        }
    }
}
```

# Probabilistic profiling

- Start with a cheap random source.
- RandomSeedNode, which lowers to rdtsc on x64.
- For loops it's too expensive - so, inject linear congruential generators into loops: $X_{n+1} = a * X_n + c$
- Injection is also done in FinalizeProfileNodesPhase. Lowered with ProbabilisticProfileSnippets.
- Almost anything is better than a cache miss!
- Up to 30% speedup on NUMA machines.

# Probabilistic Profiling

## Source files

- Random generator insertion: FinalizeProfileNodesPhase.java
- Random seed node: RandomSeedNode.java
- Lowering: ProbabilisticProfileSnippets.java

# Inlining

- Special inlining policy. No profiling, allows for less depth.
- We don’t support CHA dependency validation, some CHA assumptions (single leaf) are converted to runtime checks.
- Otherwise inlining is only for exact types.

# Inlining

## Source files

- Policy: AOTInliningPolicy.java
- CHA workaround: InliningData.java (see getTypeCheckedAssumptionInfo())