State of Reachability Fence API support

java.lang.ref.Reference::reachabilityFence

Vladimir Ivanov

Senior Principal Member of Technical Staff

HotSpot JVM Compilers

Java Platform Group, Oracle Corp.

09 September 2025

GC-assisted Resource Management

- Native resource management in managed runtimes is challenging
  - Due to interactions with GC
- All types of native resources
  - File descriptors, native memory, …
- Track native resources as if they were on-heap Java memory
  - holder object represents an arbitrary resource
  - java.io.FileDescriptor, java.nio.DirectByteBuffer, …
- GC APIs in Java
  - Object::finalize()
  - java.lang.ref.Reference et al
  - java.lang.ref.Cleaner

GC-assisted Resource Management

- Main challenge: **non-determinism**
  - As early as next GC cycle, but not earlier
  - Delayed indefinitely (potentially)
  - **Premature reclamation** (JLS 12.6.1)
    - e.g., no guarantees a receiver of an instance method is kept alive for the duration of the call

- Remedies
  - Manage resources explicitly: AutoCloseable, try-with-resources
  - Control object liveness: **Reference::reachabilityFence** API

Reachability Fence API

public static void reachabilityFence(Object ref)

”Ensures that the given object remains strongly reachable. This reachability is assured regardless of any optimizing transformations the virtual machine may perform that might otherwise allow the object to become unreachable (see JLS 12.6.1). Thus, the given object is not reclaimable by garbage collection at least until after the invocation of this method. References to the given object will not be cleared (or enqueued, if applicable) by the garbage collector until after invocation of this method. Invocation of this method does not itself initiate reference processing, garbage collection, or finalization.

This method establishes an ordering for strong reachability with respect to garbage collection. It controls relations that are otherwise only implicit in a program -- the reachability conditions triggering garbage collection. This method is applicable only when reclamation may have visible effects, such as for objects that use finalizers or Cleaner, or code that performs reference processing.”

Reachability Fence API

java.lang.ref.Reference::reachabilityFence(Object ref)

- Introduced in JDK 9
  - JDK-8133348: ”Reference.reachabilityFence”
  - Conservative support in JVM
    - Implemented as a call (@DontInline)
  - Utilized in Cleaner API right away
    - JDK 9: JDK-8145459: “Cleaner - use Reference.reachabilityFence”

Reachability Fence API

java.lang.ref.Reference::reachabilityFence(Object ref)

- Improved in JDK 11
  - JDK-8199462: “Use Reference.reachabilityFence in direct ByteBuffer methods”
  - Peter Levart noticed that a bare call on bytecode level seems to be enough
    - piggyback on debug information tracking
  - @DontInline => @ForceInline

Reachability Fence API

java.lang.ref.Reference::reachabilityFence(Object ref)

- JDK 17: Spotted a bug with FFM API
  - Shared segments JVM support
    - JVM-assisted closure of shared memory segments
      - jdk.internal.misc.ScopedMemoryAccess::closeScope0
  - Workaround in FFM API implementation
    - Unconditionally deoptimize
  - Turned out Reference::reachabilityFence requires special support in C2 (at least)
  - JDK-8290892 tracks the bug

JDK-8290892: ”C2: Intrinsify Reference.reachabilityFence”

Overview

JDK-8290892

Example

```
public class MyOffHeapBuffer {
    final long address;

    public byte getByte(long offset) {
       return UNSAFE.getByte(null, address + offset);
    }
}
```

JDK-8290892

Example

```
public class MyOffHeapBuffer {
    final long address;
    final long limit;

    public byte getByte(long offset) {
        Preconditions.checkIndex(i, limit, …);
        return UNSAFE.getByte(null, address + offset);
    }
}
```

JDK-8290892

Example

```
public class MyOffHeapBuffer {
    final long address;
    final long limit;

    public byte getByte(long offset) {
      Preconditions.checkIndex(i, limit, …);
      try {
        return UNSAFE.getByte(null, address + offset);
      } finally {
        Reference.reachabilityFence(this);
      }
    }
}
```

JDK-8290892

Example

```
public class MyOffHeapBuffer {
        final long address;
        final long limit;

        public byte getByte(long offset) {
            …
        }
        MyOffHeapBuffer(long sizeInBytes) {
            …
            Cleaner.create().register(this, () -> {
                UNSAFE.freeMemory(address);
            });
        }
}
```

JDK-8290892

Example

```
MyOffHeapBuffer buffer = new MyOffHeapBuffer(sizeInBytes);

for (int i = 0; i < limit; i++) {
  byte b = buffer.getByte(i);
  …
}
```

JDK-8290892

Example

```
MyOffHeapBuffer buffer = new MyOffHeapBuffer(sizeInBytes); // field

for (int i = 0; i < limit; i++) {
  byte b = buffer.getByte(i);
  …
  // safepoint: doesn’t contain buffer in its JVM state
}
```

JDK-8290892

Example: After inlining

```
for (int i = 0; i < limit; i++) {
    MyOffHeapBuffer localBuffer = this.buffer;
    long localLimit = localBuffer.limit;
    if (i < 0 || i >= localLimit) ...
    long localAddress = localBuffer.address;
    byte b = UNSAFE.getByte(null, localAddress + i);
    // Reference.reachabiltiyFence(localBuffer)
    …
    // safepoint
}
```

JDK-8290892

Example: Loop strip-mining

```
for (long j = 0; j < limit + stride; j += stride) { // outer loop
  long innerLimit = Math.min(limit, j + stride);

  for (long i = j; i < innerLimit, limit); i++) { // counted loop
    MyOffHeapBuffer localBuffer = this.buffer;
    long localLimit = localBuffer.limit;
    if (i < 0 || i >= localLimit) { ... }
    long localAddress = localBuffer.address;
    byte b = UNSAFE.getByte(null, localAddress + i);
    // Reference.reachabiltiyFence(localBuffer)
    ...
  }

  // outer loop safepoint
}
```

JDK-8290892

Example: Loop-invariant code motion & loop predication

```
MyOffHeapBuffer localBuffer = this.buffer;
long localLimit = localBuffer.limit;
long localAddress = localBuffer.address;

if (limit < 0 || limit >= localLimit) { ... }

for (long j = 0; j < limit + stride; j += stride) { // outer loop
  long innerLimit = Math.min(limit, j + stride);

  for (long i = j; i < innerLimit, limit); i++) { // counted loop
    byte b = UNSAFE.getByte(null, localAddress + i);
    // Reference.reachabiltiyFence(localBuffer)
    …
  }
  // outer loop safepoint
}
```

JDK-8290892

GC at outer loop safepoint

```
MyOffHeapBuffer localBuffer = this.buffer;
long localLimit = localBuffer.limit;
long localAddress = localBuffer.address; // last usage of localBuffer oop

if (limit < 0 || limit >= localLimit) { ... }

for (long j = 0; j < limit + stride; j += stride) { // outer loop
  long innerLimit = Math.min(limit, j + stride);

  for (long i = j; i < innerLimit, limit); i++) { // counted loop
    byte b = UNSAFE.getByte(null, localAddress + i);
    // Reference.reachabiltiyFence(localBuffer)
    …
  }
  // outer loop safepoint: doesn’t keep localBuffer alive
}
```

JDK-8290892

Reachability Fence

```
MyOffHeapBuffer localBuffer = this.buffer;
long localLimit = localBuffer.limit;
long localAddress = localBuffer.address; // last usage of localBuffer oop
…
for (long j = 0; j < limit + stride; j += stride) { // outer loop
  …
  for (long i = j; i < innerLimit, limit); i++) { // counted loop
    byte b = UNSAFE.getByte(null, localAddress + i);
    ReachabilityFence(localBuffer); // extends live range of localBuffer
  }
  // outer loop safepoint: oop map contains localBuffer
}
```

Proposed Fix for JDK-8290892

https://github.com/openjdk/jdk/pull/25315

JDK-8290892: Proposed Fix

Attempt #1

- Intrinsify Reference.reachabilityFence call as ReachabilityFence ideal node
  - lightweight representation akin to Blackhole ideal node

- Pros
  - fixes the bug

- Cons
  - too heavy-weight: regresses peak performance
    - negatively affects loop opts

JDK-8290892: Proposed Fix

Attempt #2

- Directly attach referent to all interfering safepoints
  - add an auxiliary edge from safepoint to referent

- Pros
  - fixes the bug; low performance overhead
- Cons
  - the set of interfering safepoints can change during compilation
    - original IR shape (after parsring) is correct
    - but transformations can break the invariant
  - still requires tracking of OOPs with reachability fence
    - back to the problem with Attempt #1

JDK-8290892: Proposed Fix

Attempt #3

- 4 steps
  1. Parsing: **Insert ReachabilityFence (RF) nodes**
  2. Loop Opts: **Optimize RFs**
     - Eliminates redundant RFs (based on dominance info)
     - Move RFs with loop-invariant referents outside loops
  3. After Loop Opts: **Eliminate all RFs**
     - … and link their referents to interfering SafePoints (SP)
     - is_interfering(SP, REF, RF) = RF keeps REF alive across SP
  4. Final Graph Reshaping: **Rematerialize RFs from SPs**
     - for each referent edge on SP insert a RF after SP

JDK-8290892: Proposed Fix

Attempt #3: Motivation

- Parsing: **Insert ReachabilityFence (RF) nodes**
  - materializes reachability fences on IR level
- Loop Opts: **Optimize RFs**
  - minimizes interference with loop opts
- After Loop Opts: **Eliminate all RFs**
  - friendlier to subsequent safepoint elimination
  - if an interfering safepoint goes away during macro expansion, corresponding RF becomes redundant and should be eliminated
  - handled automatically with SP-attached representation

JDK-8290892: Proposed Fix

Attempt #3: Motivation (cont.)

- Final Graph Reshaping: **Rematerialize RFs from SPs**
  - SP-attached representation conflicts with existing IR shape
    - derived oops are represented as extra edges on SPs
  - a RF closely following interfering SP keeps the referent alive across it

Discussion

Question #1: What about constants?

- Connection between holder object and encapsulated state can be completely lost when (a) holder is a compile-time constant; and (b) it’s fields are trusted
  - dependent load is turned into 2 independent constants
  - no guarantees the constant for holder OOP is materialized in nmethod
- When constant folding happens, JVM explicitly trusts such cases
  - static finals, @Stable fields
  - but without explicit dependency
  - bugs in user code can manifest as crashes
- Should RF support conservatively cover such corner cases?
  - -XX:+PreserveReachabilityFencesOnConstants for now (turned OFF by default)

Question #2: Is the fix complete?

- Hard to say for sure
  - It does fix the problematic case and goes beyond that, but no guarantees yet

- An open question:
  - any guarantees that a SafePoint beyond RF can’t ever interfere with the referent?
    - in other words, any guarantees that users of referent’s state can’t be pushed beyond RF to interfere with other SPs?
  - RF as a memory barrier would guarantee it’s not possible, but such representation is prohibitively expensive

Wrong abstraction for JVM?

- It's very hard to reason about the effects of RFs in the context of optimizing compiler
  - the API is not friendly to JVM
    - the spec is very vague
    - too many unrelated details to care about
  - e.g., almost impossible to restore the connection between holder object and native resource, so JVM has to be conservative

- Is Reference::reachabilityFence() the right tool for the job?
  - Do we need to rethink the whole approach?

Java™

Thank you! Q&A

ORACLE