Instruction and Return Barriers

Code Integrity and Execution Correctness

Why?

- JIT performance
  - Profile based
    - Tracing
  - Speculation
  - Garbage collection
    - Managed Runtime Resource Reclamation
- JIT compilation costly
  - Compile as little as possible

Scenarios

- Instructions ahead are invalid
- Invalid Frame Data
  - Frame part of execution
  - Barrier may trigger GC/heap barrier
- Execution Must Be Stopped
- ?

Instructions ahead are invalid

- Stale Instructions
  - I/D Cache Inconsistency
  - Invalid Materialized Constants
  - Environmental Invalidation
  - Ineffective/Unwanted Instructions
- Missing instructions
  - Incomplete Compilation
    - Tracing JIT
  - Unavailable Data
    - Lazy loading
      - Linking without all referred object files

Invalid Frame Data

- Invalid
  - Stale references
    - Reference refer to moved objects
- Missing
  - Not mounted on native stack
    - Continuations

Execution Stoppage

- Exclusive access
  - E.g. object graph
- Quiescent state
  - As in RCU/QSBR
  - E.g. a JIT:ed method is reclaimed after a QS

Implementation Strategies

- Explicit Branching
  - Conditional jumps to stub routines.
- Trapping Instructions
  - Deliberately causing hardware traps (e.g., illegal instructions) to invoke slowpaths.
- Self- or Cross-Modifying Code
  - Patching instructions at runtime to redirect control flow.
- Destination Changes
  - Altering jump targets.
- Return Address Patching
  - Modifying the stack return address to hijack control flow upon function return.

Placement

- Function Entry
  - Entry Barrier, Entry Guard, Receiver Check
- Function Exit
  - Callee, Caller, Return address
- Guts
  - Deoptimization traps
    - E.g. Dead Code Resurrection
  - Execution Guards
    - Striped mined loops
  - Initialization Guards
    - Once barriers