Instruction and Return Barriers: Code Integrity and Execution Correctness

Instruction and return barriers are distinctly different from memory barriers (for memory ordering) and garbage collector (GC) barriers (for heap consistency). Instruction barriers and return barriers, however, serve to ensure the execution of correct and valid code. As executed code is highly dependent on stack state, frame data is considered part of this correctly executed instruction stream. While frame data is also tracked by the garbage collector and may be subject to GC barriers, heap memory and heap-related barriers are excluded in this context. However, barriers for halting the execution are included which is often utilized by garbage collectors for heap memory and stack exclusivity.

Barrier scenarios

1. Instructions ahead are invalid

The first scenario (invalid instructions) can be split into two cases: stale instructions or missing instructions.
This category addresses scenarios where the instruction stream is no longer safe to execute.

Stale Instructions

The instructions exist but are incorrect due to:

- **I/D Cache Inconsistency:** The instructions are valid in memory, but the *hart* (hardware thread) has a stale *Instruction Cache*.
- **Invalid Materialized Constants:**
    - The GC moved an object, invalidating the address encoded in the instruction.
    - The GC updated the generation, invalidating the encoded generation ID.
- **Environmental Invalidation:**
    - Class hierarchy changes (Class Graph).
    - Callee function changes.
- **Ineffective/Unwanted Instructions:**
    - Implicit null checks are trapping.
        - An explicit check is preferred.
    - Active debugging or tracing hooks.

Missing instructions

- **Incomplete Compilation**
    - Not all arguments or code paths (e.g., non-executed branches) were compiled.
- **Unavailable Data**
    - Dependencies, such as none-loaded classes, were missing at compile time.

2. Invalid Frame Data

The second scenario - frame data being invalid - can be further split into two sub-categories:

- **Frame is invalid**: E.g., an object pointer in a caller-saved register is not yet updated.
- **Frame is missing**: E.g., the continuation did not mount all frames when it was resumed.

3. Execution Guard

The third case, where execution must be stopped, can happen for various reasons. The two most common cases are:

- **Exclusive** access is required (e.g., for heap memory).
- **Quiescent state** is needed for a resource change to be visible.
    - Per thread rendezvous

Implementation Strategies

Barriers are implemented using techniques ranging from trapping instructions to self-modifying code. The strategy often depends on whether the barrier is **single-shot** or **rearmable**.

Common techniques include:

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

OpenJDK and RISC-V Implementation Examples

The following are examples of barriers used in OpenJDK and, where available, their RISC-V implementations.

Function Entry and Exit

Function Entry Barrier

This prevents threads from entering a function, typically paired with a return barrier to prevent returns into the function.

- **OpenJDK Generic:** Often implemented as a `nop` patched atomically to a long jump.
- **RISC-V Specifics:** Atomic patching of `nop` to `jump` is difficult on RISC-V. Instead, RISC-V utilizes a **Function Entry Guard**.
    - *Note:* A theoretical alternative on RISC-V would be patching `nop` to `ill` to trap execution or to use `auipc` + `nop` to `auipc` + `jalr.`

Function Entry Guard

This mechanism supports **lazy cross-modifying code**. Every thread and the global system maintains an **instruction stream epoch**.

1. If the code stream changes, the global epoch increments.
2. Threads cannot enter the function without comparing their local epoch to the global epoch.
3. On mismatch the thread:
    a. Updates the instruction stream if needed.
    b. Updates its epoch (and synchronizes I/D caches) before proceeding.

```
None
  0x000076426b5a03ce:   auipc t0,0x0
  0x000076426b5a03d2:   lwu t0,74(t0) # 0x000076426b5a0418
  0x000076426b5a03d6:   lwu t1,32(s7)
  0x000076426b5a03da:   beq t0,t1,0x000076426b5a03e2
  0x000076426b5a03de:   jal zero,0x000076426b5a040a
```

Function Entry Receiver Check

Optimized polymorphic callsites may speculatively compile as direct calls, skipping vtable lookups. This barrier validates that the actual object receiver matches the expected class. If the check fails, the callsite gets re-resolved.

```
None
0x0000789922fe7cb4:   lwu   t1, 8(a1)
0x0000789922fe7cb8:   lwu   t2, 8(t0)
0x0000789922fe7cbc:   beq   t1, t2, 0x0000789922fe7cc8
0x0000789922fe7cc0:   auipc t1, 0xfff81
0x0000789922fe7cc4:   jalr  zero, 1728(t1) # 0x0000789922f69380
```

Function Return Barriers

Return barriers ensure caller and caller frame is still valid. A return barrier can be implemented in several, OpenJDK uses three methods:

1. **Caller Check:** The caller verifies validity immediately after the call returns.
    a. Deoptimization
2. **Callee Check:** The callee checks safety before executing the return instruction.
    a. Execution guard
    b. Frame valid
3. **Return Address Patching:** The return address on the stack is overwritten to redirect the thread.
    a. Frame not mounted on stack
    b. Deoptimization

Caller Check:

```
None
0x00007e6e2ef896ac:   jalr   ra, 1498(t1)
0x00007e6e2ef896b0:   addi   zero, zero, 0  // <===== overwritten
with ILL if not valid
0x00007e6e2ef896b4:   lui    zero, 0x0      // Register map
0x00007e6e2ef896b8:   addiw  zero, zero, 0  // Register map
```

Callee Execution Guard - Safepoint Polling (Non-trapping) with frame check

```
None
0x000073917b6d3dea:   ld     t0, 40(s7)  // s7 is a native pointer
to data structure representing current thread
0x000073917b6d3dee:   bgeu   t0, sp, 0x000073917b6d3df6 // Per
thread allowed depth check
```

```
0x000073917b6d3df2:   jal    zero, 0x000073917b6d3df8
```

Callee Execution Guard - Safepoint Polling (Non-trapping)

```
None
0x00007899236d097e:   ld     t0, 40(s7) // s7 is a native pointer to
data structure representing current thread
0x00007899236d0982:   bexti  t0, t0, 0x0
0x00007899236d0986:   beq    t0, zero, 0x00007899236d099e
```

Instruction Barriers Not Related to Entry or Exit - In The Guts

Compiled function missing constant

This occurs when a caller is compiled before the callee (and its associated constants) is fully loaded.

- **Concept:** A jump targets a stub that serializes execution, looks up the constant, and patches the original jump to a `mov` instruction.
- **RISC-V Note:** This is not implemented on RISC-V due to limitations in cross-modifying code sequences (converting a jump to a move atomically is complex).

Incomplete Compilation

Class Init Barrier (loading, non trapping)

Used when a code path requires a class that may not be initialized or loaded (Java loads classes lazily).

```
None
0x0000789922f6b258:   lbu    t0, 304(t1)
0x0000789922f6b25c:   fence  r, rw
0x0000789922f6b260:   c.addi t0, -4
0x0000789922f6b262:   beq    t0, zero, 0x0000789922f6b276
0x0000789922f6b266:   ld     t0, 312(t1)
0x0000789922f6b26a:   beq    s7, t0, 0x0000789922f6b276
```

Trapping, deoptimizing to interpreter

Usually emitted as a branch or trap.

- **Null or Zero Violations**: The code encountered an unexpected null object or a division by zero.
- **Unexpected Non-Null**: The compiler assumed a value would be null (or zero) based on profiling, but a real value appeared.
- **Range Check Failure**: An attempt was made to access an array index that was out of bounds.
- **Type Mismatches**: A runtime checkcast failed, or the wrong type was stored into an object array.
- **Intrinsic Failure**: An optimized intrinsic method received an input argument it could not handle.
- **Polymorphic Thrashing**: An inlined method call site encountered a new object type not seen previously (e.g., a 3rd type at a bimorphic site).
- **Dead Code Resurrection**: A branch that the compiler marked as "never taken" (dead) was actually entered.
- **Broken Loop Predicates**: Assumptions made to optimize loops (like moving checks outside the loop) turned out to be false.
- **Failed Speculations:** Aggressive optimizations based on previous profiling (speculating an object will always be Type X or never be null) were proven wrong.
- **Loop Limit Violations**: An unrolled loop exceeded the safety limits set by the compiler.
- **Unloaded Classes**: The code referenced a class or constant pool entry that has since been unloaded from memory.
- **Uninitialized State**: The code attempted to access a class that is not yet fully initialized.
- **Code Age/Tenure**: The compiled code has existed for too long or has trapped too many times and needs to be refreshed.
- **Vectorization Failure**: Speculative attempts to use SIMD (Single Instruction, Multiple Data) instructions failed.

Implicit Null Check

Used when a code path requires none null.

```
None
  0x000079eba765a1f0:   lwu t2,12(a1)                    ; implicit
exception: dispatches to 0x000079eba765a338
```

Execution guard

Halting execution is required to manage state consistency. This can range from a **Global Safepoint**, where all Java threads are paused to establish an **exclusive access**, to a **quiescent state**, where all Java threads are past a rendezvous point. A single thread can also be requested to pause, e.g. for exclusive access to stack. data

Safepoint Polling (Non-trapping)

```
None
0x000073917b6d4738:   ld     t0, 40(s7) // s7 is a native pointer to
data structure representing current thread
0x000073917b6d473c:   bgeu   t0, s0, 0x000073917b6d4744
0x000073917b6d4740:   jal    zero, 0x000073917b6d47d4
```

Safepoint Polling (Trapping)

Typical in strip-mined outer loops.

```
None
0x000073917b5a292c:   ld     t2, 48(s7) // s7 is a native pointer to
data structure representing current thread
0x000073917b5a2930:   lwu    zero, 0(t2)
```

Callee changes

During the lifetime of a JIT-compiled function, the methods it calls (callee) may change state.

- Dynamic Resolution: Calls do not block. Instead, they are re-resolved dynamically as the target changes (e.g., transitioning from the interpreter to JIT-compiled code).
- Unlimited Updates: There is no limit to the number of times a callsite may be resolved and updated.
- Stale Callers: To handle stray (stale) callers, function entry barriers are used to force the caller to re-resolve the callsite.
- Implicit Barrier: Essentially, the target address of the callsite acts as the barrier itself.