Concurrent thread-stack processing in the Z Garbage Collector

Erik Österlund
Consulting Member of Technical Staff
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, timing, and pricing of any features or functionality described for Oracle’s products may change and remains at the sole discretion of Oracle Corporation.
Contents

• What is ZGC
• Goals
• Technical Overview
• JEP 376: ZGC: Concurrent Thread-Stack Processing
• Goals
• Technical Overview
• Evaluation
What is ZGC

- Low latency
- Scalable
- Easy to Use

- Concurrent
- Tracing
- Compacting
- Region-based
- NUMA-aware
- Load barriers
Z Garbage Collector Goals

Max GC pause time: 10 ms JDK < 16
Max heap size: 16 TB
Max CPU overhead: 15%

Easy to Tune!
Z Garbage Collector Goals

Max GC pause time
- 10 ms  JDK < 16
- <1 ms  JDK >= 16

Max heap size
- 16 TB

Max CPU overhead
- 15 %

Easy to Tune!
GC pause times **do not** increase with the heap or live-set size
JDK < 16

GC pause times do increase with the root-set size
JDK >= 16

GC pause times **do not** increase
with the root-set size
ZGC Phases
JDK < 16

- Mark Start
- Concurrent Mark/Remap
- Mark End
- Concurrent prepare for Relocation
- Relocate Start
- Concurrent Relocation
ZGC Phases
JDK < 16

Mark Start

Concurrent Mark/Remap

Mark End

Concurrent prepare for Relocation

Relocate Start

Concurrent Relocation

Scan Thread Stacks
ZGC Phases
JDK < 16

Mark Start
Concurrent Mark/Remap
Scan Thread Stacks

Walk Object Graph

Mark End
Concurrent prepare for Relocation

Relocate Start
Concurrent Relocation
ZGC Phases
JDK < 16

Mark Start

Mark End

Relocate Start

Mark Start
Concurrent Mark/Remap
Scan Thread Stacks

Concurrent prepare for Relocation
Synchronization Point

Concurrent Relocation
ZGC Phases
JDK < 16

Walk Object Graph
- Mark Start
- Concurrent Mark/Remap
- Scan Thread Stacks
- Synchronization Point

Reference Processing
- Class Unloading
- Relocation Set Selection

Concurrent Mark/Remap
- Concurrent prepare for Relocation
- Concurrent Relocation

Relocation Set Selection
- Relocate Start
ZGC Phases
JDK < 16

- Mark Start
- Mark End
- Relocate Start

- Walk Object Graph
- Scan Thread Stacks
- Synchronization Point
- Scan Thread Stacks

- Concurrent Mark/Remap
- Concurrent prepare for Relocation
- Concurrent Relocation

- Reference Processing
- Class Unloading
- Relocation Set Selection

- Mark Start
- Mark End
- Relocate Start
ZGC Phases
JDK >= 16

Walk Object Graph

Mark Start

Concurrent Mark/Remap

Synchronization Point

Concurrent prepare for Relocation

Concurrent Relocation

Relocate Start

Reference Processing
Class Unloading
Relocation Set Selection

Concurrent prepare for Relocation

Synchronization Point

Class Unloading
JEP 376: ZGC: Concurrent Thread-Stack Processing
JEP Goals

- Remove thread-stack processing from ZGC safepoints
- Make stack processing lazy, cooperative, concurrent, and incremental

- Remove all other per-thread root processing from ZGC safepoints
- Provide a mechanism by which other HotSpot subsystems can lazily process stacks
Stack Watermarks

Overview

- Need to process object references in stack frames
- Keep track of what frames have been processed
- Ensure that the top 2 frames are always processed
  - Let’s call them “caller” and “callee”
  - Catch violating frame unwinding
  - Process 3 frames when leaving a safepoint
- Use per-thread lock to coordinate processing with GC
Stack Processing Overview

What we want to do

Mark Start

Concurrent Mark/Remap

Stack Processing

Snapshot
grow
top
Stack Processing Overview

What we want to do

Mark Start

Concurrent Mark/Remap

Snapshot

Initial Processing

caller
callee

Stack Processing
Stack Processing Overview
What we want to do

Mark Start

Concurrent Mark/Remap

Stack Processing

Snapshot
Initial Processing
Return

caller
callee
caller
callee
Stack Processing Overview

What we want to do

Mark Start

Concurrent Mark/Remap

Stack Processing

Snapshot

Initial Processing

Return

Return

caller
callee

caller
callee

caller
callee
Stack Processing Overview

What we want to do

Mark Start

Concurrent Mark/Remap

Stack Processing

Snapshot

Initial Processing

Return

Return

Call

caller
callee
caller
callee
caller
callee
Stack Processing Overview

What we want to do

Mark Start

Concurrent Mark/Remap

Stack Processing

Snapshot  | Initial Processing  | Return  | Return  | Call  | GC
---|---|---|---|---|---
| | | | | |
Stack Processing Overview

What we want to do

Mark Start
Concurrent Mark/Remap

Stack Processing

Snapshot
Initial Processing
Return
Return
Call
GC
GC

caller
callee
caller
callee
caller
callee
caller
callee

caller
callee

caller
callee

caller
callee

Java

Copyright © 2021, Oracle and/or its affiliates
Stack Watermark Barrier

Initial Processing

- At the GC safepoint, we flip the current GC phase, and continue
- When Java thread wakes up from safepoint, it detects the phase change
  - Detects stack and thread roots are invalid
- Trigger initial processing
  - Process three frames
  - Process other thread roots
- Update GC phase of stack watermark
Stack Processing Overview

What we want to do

Mark Start

Concurrent Mark/Remap

Snapshot

Initial Processing

caller
callee

Stack Processing
Stack Watermark Barrier
Intercepting dangerous returns

• Before returning, check if stack watermark invariant will break (at least 2 frames processed)
• Compare if frame pointer > thread-local “poll” value
  • Top frame grows and shrinks compared to snapshot, but frame pointer is constant
• Poll value set to stack pointer of “callee” frame
Stack Processing Overview

What we want to do

Mark Start

Concurrent Mark/Remap

Snapshot

Initial Processing

Return

Return

Call

GC

GC
Stack Watermark Barrier
Intercepting dangerous returns

- Adding instructions per compiled call can be performance sensitive
- Ideal to incorporate check into existing checks
- Stack watermark barrier replaces previous method epilog safepoint poll
Compiled Method Epilog

Overview

- Frame pointer is *not* available
- Poll happens after unwinding frame
- Stack pointer available instead
  - Frame pointer - 8
Compiled Method Epilog
x86_64 assembly

```
movq rtmp, 0x330(r15)
testb rax, 0x0(rtmp)
```
Compiled Method Epilog
x86_64 assembly

```assembly
movq rtmp, 0x330(r15)
testb rax, 0x0(rtmp)
cmpq rsp, 0x338(r15)
ja slow_path
```
Overview

- Frame pointer is available
- Poll happens *before* unwinding
  - This is why initial processing needs 3 frames
  - A new GC phase shift can happen in the unwind handler, before top frame is unwinded
  - Expectation after unwinding is at least top 2 frames are processed
Interpreter Method Epilog

x86_64 assembly

```
cmpq rbp, 0x338(r15)
ja slow_path
```

Diagram:
- `rbp` (callee frame pointer)
- Poll value
- Return PC
- Previous RBP
- Caller
- Callee
Loop safepoint polls

Overview

- Compiled methods: same as before (indirect load)
- Interpreter: check if low order bit is set
Loop polls
x86_64 assembly

Interpreter:

```
testq 0x338(r15), 0x1
jnz slow_path
```

Compiled:

```
movq rtmp, 0x330(r15)
testb rax, 0x0(rtmp)
```
**Stack Watermark Barrier**
Intercepting dangerous returns

<table>
<thead>
<tr>
<th>Thread-local poll value</th>
<th>Stack Watermark</th>
<th>Safepoint</th>
<th>Thread-local handshake</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFFFFFFFFFFFFE</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>0x1</td>
<td>None</td>
<td>Pending</td>
<td>None</td>
</tr>
<tr>
<td>0x1</td>
<td>None</td>
<td>None</td>
<td>Pending</td>
</tr>
<tr>
<td>0x1</td>
<td>None</td>
<td>Pending</td>
<td>Pending</td>
</tr>
<tr>
<td>0x1</td>
<td>Invariant breaks when unwinding some frame</td>
<td>Pending</td>
<td>None</td>
</tr>
<tr>
<td>0x1</td>
<td>Invariant breaks when unwinding some frame</td>
<td>None</td>
<td>Pending</td>
</tr>
<tr>
<td>0x1</td>
<td>Invariant breaks when unwinding some frame</td>
<td>Pending</td>
<td>Pending</td>
</tr>
<tr>
<td>$callee_sp</td>
<td>Invariant breaks when unwinding frame for $callee_sp</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>
Stack Processing Overview

What we want to do

Mark Start

Concurrent Mark/Remap

Snapshot

Initial Processing

Return

Return

 caller
callee
caller
callee
caller
callee
Stack Watermark Barrier
Intra-thread interactions

• GC thread wants to process frame of Java thread
• Take lock
• Process caller of “caller”
• Update caller/callee relationship one frame up in the stack
• Release lock
• Thread-local poll value may only be updated by the thread itself
  • Need to perform an acquire to make concurrent stack changes visible
Stack Processing Overview

What we want to do

Snapshots: caller callee

Initial Processing: caller callee

Return: caller callee

Return: caller callee

Call: caller callee

GC: caller callee

GC: caller callee

Mark Start

Concurrent Mark/Remap
Stack Watermark Barrier
Intra-thread interactions cont.

• Non-GC threads may also access thread internals of a remote thread
• Initial processing done under the lock
  • Can be triggered by other threads
• Used by thread-local handshakes to make remote thread access safe
Stack Watermark Barrier

Stack walkers

- JVM has stack walkers, reading object references deeper into the stack
- Can’t read object references from stack with load barriers
  - Object-internal pointers break
- JVM-internal stack walking API hooks to process frames in the stack watermark
  - When asking for frame caller, ensure it is processed before exposing it
- Must initialize processing first
- Any thread may initialize processing on any other thread
- Must grab hold of thread to walk its stack
  - Current thread: always initialized
  - Thread-local handshake: forces initialization
  - Safepoint: forces initialization (for random GC-unaware safepoints)
  - Async call trace: inaccurate; can’t read object references
  - JFR stack sampling: inaccurate; can’t read object references
- ...
Stack Watermark Barrier
Exception handling

• So we figured out returns, but what about `throw`?
• Every type of frame has an exception handler, invoked when its callee unwinds into the caller
• Exception handler invoked `after` unwinding
Stack Watermark Barrier
Exception handling

Upcall
Deoptimized c1
Interpreter
c1
c2

Caller
Callee
Stack Watermark Barrier

Exception handling

- upcall
- deoptimized
c1
- interpreter
c1
- c2

 caller
Exception handling

Stack Watermark Barrier

Process frame in c2 exception handler **after** unwinding
Stack Watermark Barrier
Exception handling

- caller
- c1
- interpreter
- deoptimized
- upcall
Stack Watermark Barrier
Exception handling

Process frame in c1 exception handler after unwinding
Stack Watermark Barrier

Exception handling

- upcall
- deoptimized c1
- interpreter

caller
Stack Watermark Barrier

Exception handling

- upcall
- deoptimized c1
- interpreter

caller
Stack Watermark Barrier
Exception handling

Process frame in interpreter exception handler after unwinding
Stack Watermark Barrier
Exception handling

```
upcall

deoptimized
c1
caller
```
Stack Watermark Barrier
Exception handling

Process frame in depot with exception handler after unwinding
Stack Watermark Barrier
Exception handling

Process frame in upcall exception handler after unwinding
Stack Watermark Barrier
Exotic unwinding

- On-stack-replacement
  - Unwind interpreter frame, replace with compiled frame
- Deoptimization
  - Unwind compiled frame, replace with interpreted frame
- JVMTI pop frame
- JVMTI force early return
- Unwinding from native code
Stack Watermark Barrier
How to process a frame

• Need to find object references, and fix them
  • During relocation phase
    • Relocate objects that need to move
    • Remap references to objects that have been relocated
    • Fix pointer colors (to “remapped” color)
  • During marking phase
    • Lazily remap object references
    • Mark objects as live
    • Fix pointer colors (to some “marked” color)
Stack Watermark Barrier
How to process a frame

• Object references can be found embedded in the frame
• Can also be found embedded in compiled method of frame
  • Lazily apply compiled method entry barriers
    • Usually invoked on first call to a method, per GC phase
    • Update embedded object references
    • Disarm compiled method entry barrier
Stack Watermark Barrier

Extensibility

- All this logic is hidden in a shared HotSpot framework
- Specific user of the API only needs to specify
  - how to process a frame
  - how to detect GC phase change if applicable
- Shenandoah adopted this
- Discussions about using it for optimized JFR stack sampling
Evaluation

GC pause times

Average GC Pause Times on SPECjbb2015 128G

- ParallelGC
- G1GC
- ZGC

JDK15  JDK17
Evaluation
GC pause times

Average GC Pause Times on SPECjbb2015 128G

- ParallelGC
- G1GC
- ZGC

JDK15  JDK17
Questions
The End