JEP 376: ZGC: Concurrent Thread-Stack Processing

Owner	Erik Österlund
Type	Feature
Scope	Implementation
Status	Closed / Delivered
Release	16
Component	hotspot / gc
Discussion	hotspot dash gc dash dev at openjdk dot java dot net
Effort	S
Duration	S
Reviewed by	Mikael Vidstedt, Per Liden, Stefan Karlsson
Endorsed by	Mikael Vidstedt
Created	2020/02/21 09:12
Updated	2021/03/07 19:41
Issue	8239600

Summary

Move ZGC thread-stack processing from safepoints to a concurrent phase.

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.

Non-Goals

- It is not a goal to implement concurrent per-thread processing of non-GC safepoint operations, such as class redefinition.

Success Metrics

- The throughput cost of the improved latency should be insignificant.
- Less than one millisecond should be spent inside ZGC safepoints on typical machines.

Motivation

The ZGC garbage collector (GC) aims to make GC pauses and scalability issues in HotSpot a thing of the past. We have, so far, moved all GC operations that scale with the size of the heap and the size of metaspace out of safepoint operations and into concurrent phases. Those include marking, relocation, reference processing, class unloading, and most root processing.

The only activities still done in GC safepoints are a subset of root processing and a time-bounded marking termination operation. The roots include Java thread stacks and various other thread roots. These roots are problematic, since they scale with the number of threads. With many threads on large machine, root processing becomes a problem.

In order to move beyond what we have today, and to meet the expectation that time spent inside of GC safepoints does not exceed one millisecond, even on large machines, we must move this per-thread processing, including stack scanning, out to a concurrent phase.

After this work, essentially nothing of significance will be done inside ZGC safepoint operations.

The infrastructure built as part of this project may eventually be used by other projects, such as Loom and JFR, to unify lazy stack processing.

Description

We propose to address the stack-scanning problem with a *stack watermark barrier*. A GC safepoint will logically invalidate Java thread stacks by flipping a global variable. Each invalidated stack will be processed concurrently, keeping track of what remains to be processed. As each thread wakes up from the safepoint it will notice that its stack is invalid by comparing some epoch counters, so it will install a *stack watermark* to track the state of its stack scan. The stack watermark makes it possible to distinguish whether a given frame is above the watermark (assuming that stacks grow downward) and hence must not be used by a Java thread since it may contain stale object references.

In all operations that either pop a frame or walk below the last frame of the stack (e.g., stack walkers, returns, and exceptions), hooks will compare some stack-local address to the watermark. (This stack-local address may be a frame pointer, where available, or a stack pointer for compiled frames where the frame pointer is optimized away but frames have a reasonably constant size.) When above the watermark, a slow path will be taken to fix up one frame by updating the object references within it and moving the watermark upward. In order to make returns as fast as they are today, the stack watermark barrier will use a slightly modified safepoint poll. The new poll not only takes a slow path when safepoints (or indeed thread-local handshakes) are pending, but also when returning to a frame that has not yet been fixed up. This can be encoded for compiled methods with a single conditional branch.

An invariant of the stack watermark is that, given a callee which is the last frame of the stack, both the callee and the caller are processed. To ensure this, when the stack watermark state is installed when waking up from safepoints, both the caller and the callee are processed. The callee is armed so that returns from that callee will trigger further processing of the caller, moving the armed frame to the caller, and so on. Hence processing triggered by frame unwinding or walking always occurs two frames above the frame being unwound or walked. This simplifies the passing of arguments that have to be owned by the caller yet are used by the callee; both the caller and the callee frames (and hence the extra stack arguments) can be accessed freely.

Java threads will process the minimum number of frames needed to continue execution. Concurrent GC threads will take care of the remaining frames, ensuring that all thread stacks and other thread roots are eventually processed. Synchronization, utilizing the stack watermark barrier, will ensure that Java threads do not return into a frame while the GC is processing it.

Alternatives

When it comes to dealing with stack walkers, we considered the alternative solution of sprinkling load barriers across the VM where object references are loaded from the stack. We dismissed this because it fundamentally could not guarantee that root processing of internal pointers into objects are processed correctly. The base pointer of an internal pointer must always be processed after an internal pointer, and stack walkers would risk violating that invariant. Therefore we chose the approach of processing the whole frame, if not already processed, via stack walking.

Testing

The main code paths affected by this work are paths that other tests already stress to a great degree, so stress testing with the existing testing infrastructure should be sufficient.