JEP 423: Region Pinning for G1

Author	Hamlin Li
Owner	Thomas Schatzl
Type	Feature
Scope	Implementation
Status	Closed / Delivered
Release	22
Component	hotspot / gc
Discussion	hotspot dash gc dash dev at openjdk dot java dot net
Effort	L
Duration	M
Reviewed by	Thomas Schatzl, Vladimir Kozlov
Endorsed by	Vladimir Kozlov
Created	2021/10/28 08:05
Updated	2024/02/05 16:37
Issue	8276094

Summary

Reduce latency by implementing region pinning in G1, so that garbage collection need not be disabled during Java Native Interface (JNI) critical regions.

Goals

- No stalling of threads due to JNI critical regions.
- No additional latency to start a garbage collection due to JNI critical regions.
- No regressions in GC pause times when no JNI critical regions are active.
- Minimal regressions in GC pause times when JNI critical regions are active.

Motivation

For interoperability with unmanaged programming languages such as C and C++, JNI defines functions to get and then release direct pointers to Java objects. These functions must always be used in pairs: First, get a pointer to an object (e.g., via `GetPrimitiveArrayCritical`); then, after using the object, release the pointer (e.g., via `ReleasePrimitiveArrayCritical`). Code within such function pairs is considered to run in a *critical region*, and the Java object available for use during that time is a *critical object*.

When a Java thread is in a critical region, the JVM must take care not to move the associated critical object during garbage collection. It can do this by *pinning* such objects to their locations, essentially locking them in place as the GC moves other objects. Alternatively, it can simply disable GC whenever a thread is in a critical region.

The default GC, G1, takes the latter approach, disabling GC during every critical region. This has a significant impact on latency: If a Java thread triggers a GC then it must wait until no other threads are in critical regions. The severity of the impact depends upon the frequency and duration of critical regions. In the worst cases users report critical sections blocking their entire application for minutes, unnecessary out-of-memory conditions due to thread starvation, and even premature VM shutdown. Due to these problems, the maintainers of some Java libraries and frameworks have chosen not to use critical regions by default (e.g., JavaCPP) or even at all (e.g., Netty), even though doing so can adversely affect throughput.

With the change that we propose here, Java threads will never wait for a G1 GC operation to complete.

Description

Background

G1 partitions the heap into fixed-size *memory regions* (not to be confused with *critical* regions). G1 is a generational collector, so any non-empty region is a member of either the young generation or the old generation. In any particular collection operation, objects are *evacuated* (i.e., moved) from only a subset of the regions to some other subset.

If G1 is unable to find space to evacuate an object during a minor (i.e., young-generation) collection then it leaves the object in place and marks both it and its containing region as having *failed evacuation*. After evacuation, G1 fixes up the failed regions by promoting them from the young generation to the old generation, potentially keeping them ready for subsequent evacuation.

G1 is already capable of pinning objects to their memory locations during major (i.e., full) collection operations, simply by not evacuating the regions that contain them. For example, G1 pins *humongous* regions, which contain large objects. It also pins, for the duration of a single collection, any region that exceeds a specified liveness threshold.

G1 cannot pin arbitrary regions during minor collection operations, though it does exclude humongous regions from such collections.

Pinning regions during minor collection operations

We aim to achieve the above goals by extending G1 to pin arbitrary regions during both major and minor collection operations, as follows:

- Maintain a count of the number of critical objects in each region: Increment it when a critical object in that region is obtained, and decrement it when that object is released. When the count is zero then garbage-collect the region normally; when the count is non-zero, consider the region to be pinned.
- During a major collection, do not evacuate any pinned region.
- During a minor collection, treat pinned regions in the young generation as having failed evacuation, thus promoting them to the old generation. Do not evacuate existing pinned regions in the old generation.

Once we have done this then we can implement JNI critical regions — without disabling GC — by pinning regions that contain critical objects and continuing to collect garbage in unpinned regions.

Alternatives

The JNI specification suggests two other ways to implement critical regions:

- At the start of a critical region, copy the critical object to the C heap, where it will not be moved; at the end of the critical region, copy it back.

  This is very inefficient in both time and space. In G1 we could do this only for critical objects in regions that cannot be pinned. Those regions are in the young generation, however, in which most object use and modification typically occurs, so we do not expect that this would help much.

- Pin objects individually.

  G1 can only evacuate whole regions, so a single pinned object in a region would prevent the collection of that region. The end result would be little different from what we propose above except that it would have higher overhead, since tracking individual pinned objects is more costly than maintaining per-region counts of critical objects.

Testing

Aside from functionality tests, we will do benchmarking and performance measurements to collect the performance data necessary to ensure that our goals are met.

Risks and Assumptions

We assume that there will be no changes to the expected usage of JNI critical regions: They will continue to be used sparingly, and they will be short in duration.

There is a risk of heap exhaustion when an application pins many regions at the same time. We have no solution for this, but the fact that the Shenandoah GC pins memory regions during JNI critical regions and does not have this problem suggests that it will not be a problem for G1.