Abstract

Performance regression was observed with a warmup process that does class loading and initialization with CDS (non-JDK default) enabled. Investigation found the effect was correlated to the runtime Java heap region size and was caused by increased GC overhead (particularly young-gen GC). The overhead was worse and more pronounced with enlarged region sizes. The issue was due to the lack of populated G1BlockOffsetTablePart for ‘open’ archive regions.

Micro-benchmark

A standalone test/benchmark implemented a tight loop for loading and initializing a list of classes (about ~100,000 classes) by invoking Class.forName(). The test could be executed in single thread mode (loading and initializing all classes sequentially within one thread) and multi-threaded mode (loading and initializing classes concurrently using a specified number of threads).

Performance Comparisons

All measurements were performed on a local Linux machine using JDK 11.

Execution Time

Each result was the average of the test execution time from 10 runs reported by `perf` tool.

NOTE: Enlarged G1 region size is used by default (if region size is not explicitly specified) in the following measurements.

Heap region size: 16M (no -Xms -Xmx setting in command-line)

	Enable CDS	CDS off
32-threads	11.900s	7.2396s
16-threads	12.136s	7.3831s
8-threads	11.7148s	8.8131s
4-threads	12.517s	11.9189s
2-threads	11.7897s	14.779s
1-thread	10.6274s	12.2889s

-Xmx5625M -Xms5625M

	Enable CDS	CDS off
32-threads	8.0970s	6.9260s
16-threads	8.9857s	7.3448s
8-threads	8.9707s	8.4970s
4-threads	9.3818s	11.9134s
2-threads	8.9602s	14.6877s
1-thread	7.9756s	12.3731s

-Xmx800M -Xms800M

	Enable CDS	CDS off
32-threads	6.0635s	7.2391s
16-threads	6.4316s	7.5044s
8-threads	6.4890s	8.7262s
4-threads	7.2306s	12.1763s
2-threads	7.5984s	14.928s
1-thread	6.8995s	12.1305s

The above performance results indicated that the Java heap size appeared to have a large impact on the performance of the warmup process when CDS archive was enabled. Using a smaller Java heap produced better results compared to using a larger Java heap with CDS archive enabled. When CDS archive was disabled, there appeared to be no significant performance impact from using different Java heap sizes for the warmup process.

CPU Cycles

	With CDS	Without CDS

-Xms5625M -Xmx5625M, 8-threads	412,078,054,544	203,027,047,699
-Xms5625M -Xmx5625M, 4-threads	365,228,947,898	174,465,171,482
-Xms600M -Xmx600M, 8-threads	211,443,168,149	307,696,590,372

With 5625M Java heap, there was a significant CPU overhead (2x) observed when CDS was enabled. Larger Java heap had higher CPU overhead.

Young Gen Collection Pauses and Overhead

When running the test with CDS enabled and in multi-threaded mode, there were noticeable pauses during the test execution. Analysis using `perf` record/report and `-Xlog:gc*` found the CPU overhead and noticeable pauses were caused by young gen collection activities.

With CDS enabled, only 2 young gen evacuations were observed during a test execution according to the gc log. However, each young gen GC had large pause:

```
[3.953s][info][gc,start     ] GC(0) Pause Young (Normal) (G1
Evacuation Pause)
[3.955s][info][gc,task      ] GC(0) Using 18 workers of 18 for
evacuation
[6.311s][info][gc,phases    ] GC(0)   Pre Evacuate Collection Set:
0.0ms
[6.311s][info][gc,phases    ] GC(0)   Evacuate Collection Set:
2353.5ms
[6.311s][info][gc,phases    ] GC(0)   Post Evacuate Collection Set:
1.8ms
[6.311s][info][gc,phases    ] GC(0)   Other: 2.0ms
[6.311s][info][gc,heap      ] GC(0) Eden regions: 32->0(28)
[6.311s][info][gc,heap      ] GC(0) Survivor regions: 0->4(4)
[6.311s][info][gc,heap      ] GC(0) Old regions: 0->6
[6.311s][info][gc,heap      ] GC(0) Archive regions: 3->3
[6.311s][info][gc,heap      ] GC(0) Humongous regions: 3->3
[6.311s][info][gc,metaspace ] GC(0) Metaspace:
13869K->13869K(1064960K)
[6.311s][info][gc           ] GC(0) Pause Young (Normal) (G1
Evacuation Pause) 1216M->495M(20480M) 2357.513ms
[6.311s][info][gc,cpu       ] GC(0) User=41.91s Sys=0.44s Real=2.36s
```

When CDS was disabled, there were 7 young gen evacuations observed during a test execution. More young gen GC activities were seen in this case because there were more Java object allocations due to the lack of pre-populated heap data. The evacuation pauses were much smaller in this case. Below was the largest observed:

```
[4.590s][info][gc,start      ] GC(12) Pause Young (Concurrent Start)
(Metadata GC Threshold)
[4.590s][info][gc,task       ] GC(12) Using 18 workers of 18 for
evacuation
[4.745s][info][gc,phases     ] GC(12)   Pre Evacuate Collection Set:
0.2ms
[4.745s][info][gc,phases     ] GC(12)   Evacuate Collection Set:
153.0ms
[4.745s][info][gc,phases     ] GC(12)   Post Evacuate Collection Set:
1.3ms
[4.745s][info][gc,phases     ] GC(12)   Other: 0.8ms
[4.745s][info][gc,heap       ] GC(12) Eden regions: 36->0(122)
[4.745s][info][gc,heap       ] GC(12) Survivor regions: 8->12(48)
[4.745s][info][gc,heap       ] GC(12) Old regions: 4->4
[4.745s][info][gc,heap       ] GC(12) Archive regions: 0->0
[4.745s][info][gc,heap       ] GC(12) Humongous regions: 3->3
[4.745s][info][gc,metaspace  ] GC(12) Metaspace:
501279K->501279K(1505280K)
[4.745s][info][gc            ] GC(12) Pause Young (Concurrent Start)
(Metadata GC Threshold) 1606M->565M(20480M) 155.444ms
[4.745s][info][gc,cpu        ] GC(12) User=1.74s Sys=0.09s Real=0.16s
[4.745s][info][gc            ] GC(13) Concurrent Cycle
[4.745s][info][gc,marking    ] GC(13) Concurrent Clear Claimed Marks
[4.745s][info][gc,marking    ] GC(13) Concurrent Clear Claimed Marks
0.120ms
[4.745s][info][gc,marking    ] GC(13) Concurrent Scan Root Regions
```

Region Size Impact

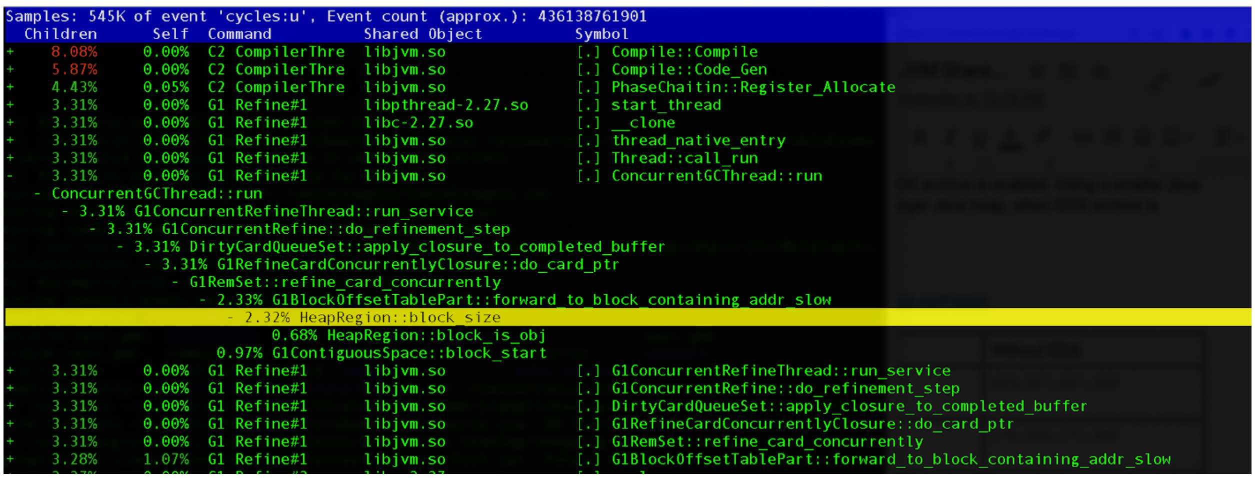
Further investigation found the actual underlying cause of the overhead was due to larger G1 heap region size. The performance impact was higher with enlarged default G1 region size.

	cycles:u	time elapsed
5G Java heap, 2M region size	194,826,729,554	6.6889 +- 0.0675 seconds

5G Java heap, 8M region size	414,257,073,171	8.9408 +- 0.0588 seconds

Top JVM functions Reported by perf

`Perf` report produced from test run with archive enabled showed GC related work was among the top CPU consumers. When archive was disabled, GC was not reported as the top CPU consumers by `perf`. The difference between the two profiling reports indicated the overhead observed with enabling the archive was related to GC. The perf call graph showed `G1BlockOffsetTablePart::forward_to_block_containing_addr_slow->HeapRegion::block_is_obj` was where the overhead came from.



```
Samples: 545K of event 'cycles:u', Event count (approx.): 436138761901
  Children      Self  Command          Shared Object          Symbol
+    8.08%     0.00%  C2 CompilerThre  libjvm.so              [.] Compile::Compile
+    5.87%     0.00%  C2 CompilerThre  libjvm.so              [.] Compile::Code_Gen
+    4.43%     0.05%  C2 CompilerThre  libjvm.so              [.] PhaseChaitin::Register_Allocate
+    3.31%     0.00%  G1 Refine#1      libpthread-2.27.so     [.] start_thread
+    3.31%     0.00%  G1 Refine#1      libc-2.27.so           [.] __clone
+    3.31%     0.00%  G1 Refine#1      libjvm.so              [.] thread_native_entry
+    3.31%     0.00%  G1 Refine#1      libjvm.so              [.] Thread::call_run
-    3.31%     0.00%  G1 Refine#1      libjvm.so              [.] ConcurrentGCThread::run
   - ConcurrentGCThread::run
      - 3.31% G1ConcurrentRefineThread::run_service
         - 3.31% G1ConcurrentRefine::do_refinement_step
            - 3.31% DirtyCardQueueSet::apply_closure_to_completed_buffer
               - 3.31% G1RefineCardConcurrentlyClosure::do_card_ptr
                  - G1RemSet::refine_card_concurrently
                     - 2.33% G1BlockOffsetTablePart::forward_to_block_containing_addr_slow
                        - 2.32% HeapRegion::block_size
                             0.68% HeapRegion::block_is_obj
                       0.97% G1ContiguousSpace::block_start
+    3.31%     0.00%  G1 Refine#1      libjvm.so              [.] G1ConcurrentRefineThread::run_service
+    3.31%     0.00%  G1 Refine#1      libjvm.so              [.] G1ConcurrentRefine::do_refinement_step
+    3.31%     0.00%  G1 Refine#1      libjvm.so              [.] DirtyCardQueueSet::apply_closure_to_completed_buffer
+    3.31%     0.00%  G1 Refine#1      libjvm.so              [.] G1RefineCardConcurrentlyClosure::do_card_ptr
+    3.31%     0.00%  G1 Refine#1      libjvm.so              [.] G1RemSet::refine_card_concurrently
+    3.28%     1.07%  G1 Refine#1      libjvm.so              [.] G1BlockOffsetTablePart::forward_to_block_containing_addr_slow
```

Cause

`G1BlockOffsetTable` divides the covered space (Java heap) into “N”-word subregions (“N” is from 2^”LogN”). It uses an `_offset_array` to tell how far back it must go to find the start of a block that contains the first word of a subregion. Every G1 region (is a `G1ContiguousSpace`) owns a `G1BlockOffsetTablePart` (associates to part of the `_offset_array`), which covers space of the current region.

For a pre-populated open archive heap region, its `G1BlockOffsetTablePart` is never set up at runtime. `G1BlockOffsetTablePart::block_start(const void* addr)` always does lookup from the start (bottom) of the region when called (for an open archive region) at runtime, regardless if the given '`addr`' is near the bottom, top or in the middle of the region. The lookup becomes linear, instead of `O(2^LogN)`. Therefore, large heap region size makes the situation worse and young-gen pauses longer.

Note that the overhead occurs to all young gen GCs throughout the execution, not just during startup.

Solution

Populate `G1BlockOffsetTableParts` and associated `G1BlockOffsetTable::_offset_array` entries for 'open' archive regions at runtime.